Documentation

PROJECT PROPOSAL

For our culminating project, we will be creating a program that contains four different games. It will provide the users a choice of which game they want to play. The options will be checkers, battleship, connect four, and tic-tac-toe. Each program will implement similar logic, GUI, and object oriented programming, but will have differences in gameplay and how the user interacts with them.

For the checkers game, it will be player vs. player, with one controlling the red checkers and one controlling the black checkers. The game will have optional jumping, with logic to check when a jump is possible. When a piece fully crosses the game board, it becomes a king, and is able to move both ways. The players will be able to move their pieces through a dialogue window that asks for the coordinates of the piece they wish to move, and then the square they wish to move it to. The program will then use built-in logic to check if the move is legal. As with normal checkers, the game ends when one player captures all of the other player's pieces.

For the Battleship game, it will be player vs AI. The game will have two screens, one with the player's ships that shows where the opponent has shot and the other with a grid showing either hit or miss. At the beginning of the game, the player will place their ships, and then go back and forth with the computer trying to sink all the opponent's ships before their own ships are sunk. The player will be able to guess a square to shoot at by entering the coordinates of that square. After each shot, the grid will refresh to show an update on hits and misses. The game ends when either the computer or the player sinks all of their opponent's ships.

For the Connect 4 game, it will be player vs player. One player will control the red tiles and one will control the blue tiles. The game will have one screen and will ask the user to input a column number to place the tile, and the tile will fall until it is blocked by another tile. The game goes back and forth between players until one gets four in a row, either horizontally, vertically, or diagonally.

For the tic, the tac, and the toe game (not to be confused with the popular candy Tic Tacs), it will be player vs. player. One player will control the Xs and one player will control the Os. In the first game, X will play first, and every game thereafter, the winner of the previous game will play first. In the case of a tie, the player who plays first will be the one who has most recently won a game. The game is won by placing three pieces in a row on the 3x3 game board. This can be done vertically, horizontally, or diagonally. To implement this game in Java, the user will be able to enter the square they wish to place their piece in on each turn, and the logic will check if that square is occupied by another piece. After each turn, the computer will check if a player has won the game. If they have won, the game will end and the program will display a win message.

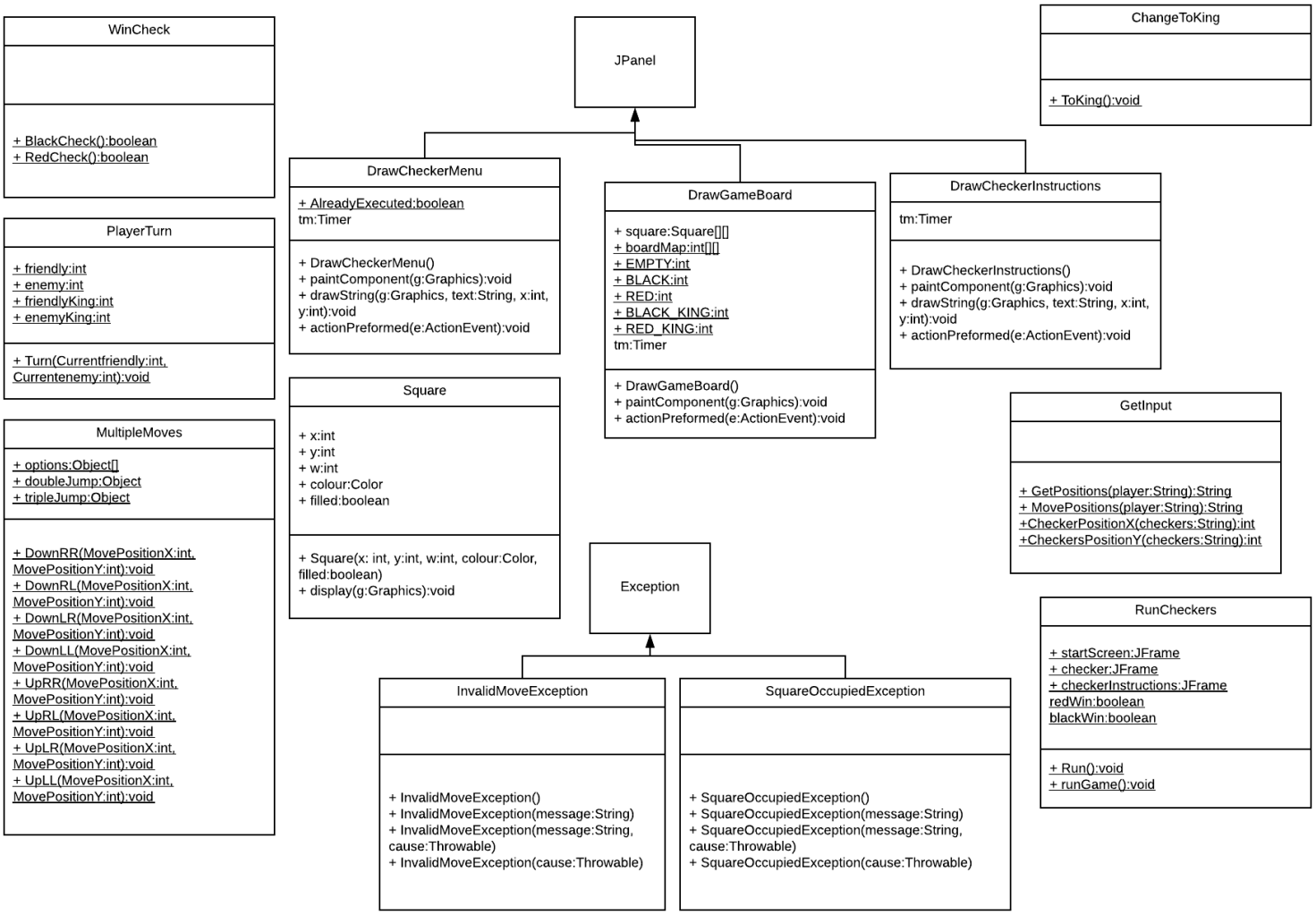
CHECKERS

Description:

Checkers is a game where each player (one black and one red) controls twelve circular pieces on an 8 by 8 board. On any given turn, a piece can only move diagonal, meaning that the game is played entirely on one colour square of the board. A legal move is either a move towards the other end of the board, or a jump over an opposing piece in the same direction. If a piece reaches the end of the board, it becomes a “king” which means that it can move in both the forwards and backwards direction. The game is won by one player when all of the opposing pieces are captured.

The game we have designed is a player vs player game. Each player will tell the computer where they want to move their piece by inputting coordinates in the format A1. Then, through a series of if statements, the computer will check if the move is legal according to the rules of checkers. If it is not legal, it will throw an exception. This exception will cause a dialogue box to pop up, showing that the move is illegal. Then, the user will be able to re enter the coordinates of their desired move. After each turn, the game logic will check if one player has no pieces left. If this is the case, the program will stop running and the opposing player wins the game.

UML Class Diagram:



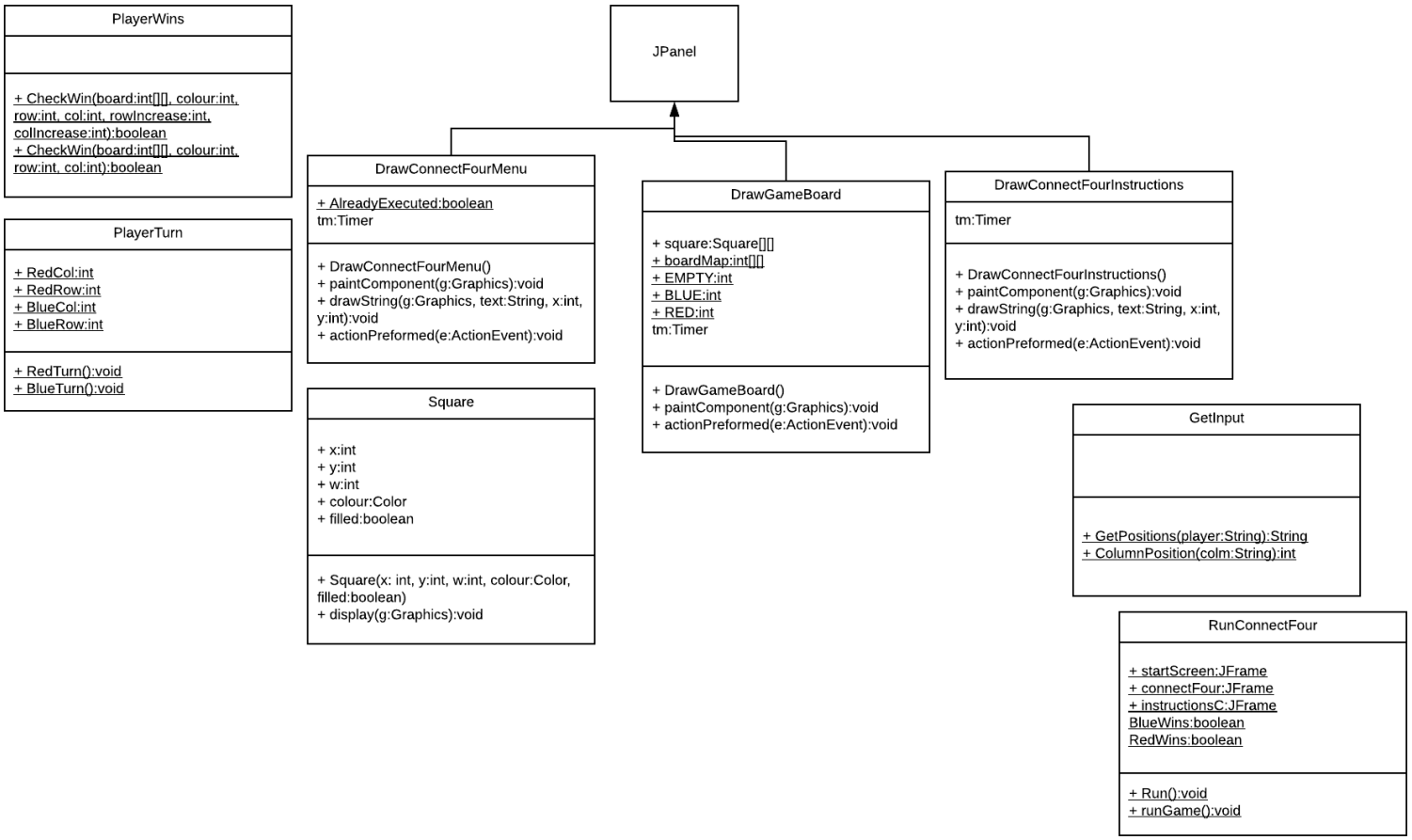
CONNECT FOUR

Description:

Connect Four is a game played by dropping tokens into columns of a board from above. The turns alternate between red and blue until one player is able to place 4 of their pieces in a row. In the game we designed, there are 6 rows and 7 columns. This leads to 42 possible squares for pieces to occupy. A player can get four in a row horizontally, vertically, or diagonally.

The game we designed is a player vs player game. A dialogue box opens, asking the player to enter to column in which they want to put their piece. Then, the game logic will check if the column is valid (between 1 and 7 inclusive). If it is not, it will prompt the user to enter a column again. If the column the user picks is already full of pieces, it will create an full column exception and prompt the user again. After each red move, the computer will check if red has won the game. If they have not, blue will be prompted to move. After each blue move, the computer will check to see if blue has won. If not, the whole sequence repeats. The program stops running when one player wins the game.

UML Class Diagram:



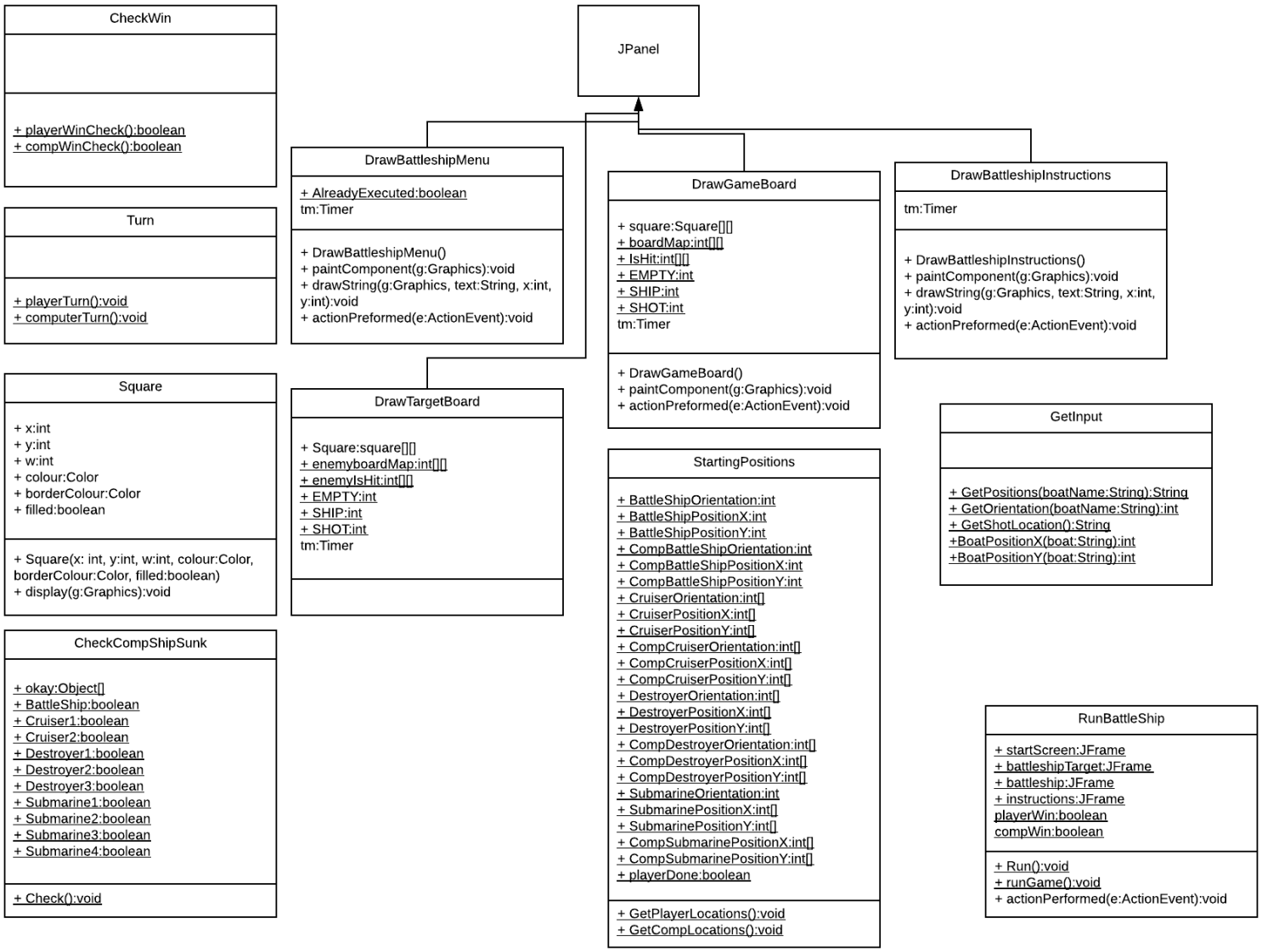
BATTLESHIP

Description:

Battleship is a game played on a 10x10 grid. Each player places their ships on the board (1 battleship - 4 tiles, 2 cruisers - 3 tiles, 3 destroyers - 2 tiles, and 4 submarines - 1 tile), either horizontally or vertically, and then take turns guessing where the enemy ships are located. If the player guesses a location of a ship, it is recorded as a hit. When all the tiles of enemy ships are hit, the player wins the game.

The game we designed is a player vs computer game. The program, for each ship, will ask the user if they want to place the ship horizontally or vertically, and then the coordinates of where the ship will be placed. Once all the ships are placed, the game starts. The computer's ships will be randomized. Then, the user will begin guessing the locations of the computer's ships. Their hits and misses will show up on a second game board. For each user turn, the computer will randomly guess a square on the user's board. If it is a hit, that section of the ship on the user's board will turn red. On each guess, the game logic will check if it is firstly on the board and secondly a square that has not been guesses previously. The game stops until one player has no ships left.

UML Class Diagram:

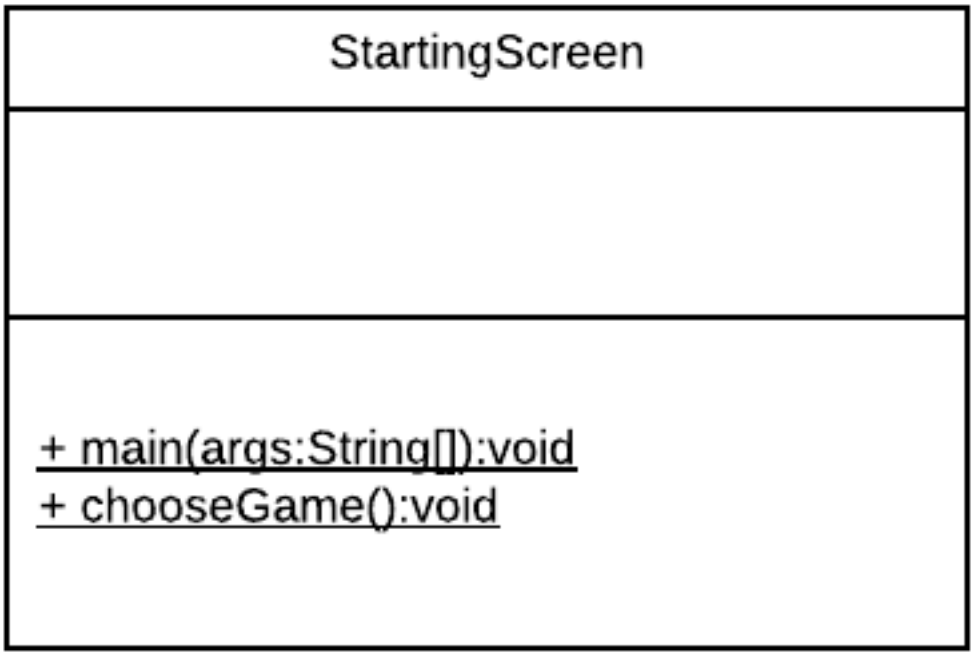


GAME ENGINE

Description:

The three games mentioned above will be tied together using a game engine. This will be a menu-style screen that allows the user to choose one of the games to play. From there, it will bring the used to a start screen for the game they choose, which will then prompt the user to start the game.

UML Class Diagram:



IPO CHART

Input	Processing	Output
• Buttons to start the game • In each game, the coordinates of pieces/shot locations	• Computer generated locations in battleship • Logic to check if moves are valid in all games • Storing game data in a board map	• Menu screens, buttons GUI • Game board, user display • Pieces once placed by the user or computer • Win message

CONSTANTS AND VARIABLES

Checkers

Constants (knowns)	Variables (unknowns)
• Starting positions of pieces • Board size • First move (black) • Window size	• Winner • Player decisions (which pieces to move)

Battleship

Constants (knowns)	Variables (unknowns)
• Board size • Window size	• User placement of ships • Orientation of ships (user and computer) • Computer placement of ships • Winner • Player decisions (where to shoot)

Connect Four

Constants (knowns)	Variables (unknowns)
• Board size • Window size	• Winner • Player decisions (where to place pieces)

PSEUDO-CODE

1. Run starting screen class
2. Accept input from the user
3. Bring user to start menu of their choice (checkers, battleship, connect four) by calling Draw_____Menu

4. Run game, open instructions, or close program (dependent on user choice)
5. Create a new instance of DrawGameBoard from the correct package to draw the board
6. Call RunGame from the correct package to run the game
7. Repeatedly ask the user for input until a win condition is met
8. Display user win message, provide option to close the program

Checkers Logic

1. Ask user for input (piece they want to move)
2. Check if the square entered:
    A. Is on the board
    B. Contains a piece of the correct colour
    C. Can legally make a move
3. If any are false, ask for input again (back to step 1)
4. If all three are true, ask user for input (square they want to move to)
5. Check if the square entered is a valid move
6. If no, ask for input again (back to step 4)
7. If yes, move the piece there by updating the board map
8. Check if any pieces can be converted to kings
9. Check if the player who has just moved has won the game (done by checking if the opposing player has more than zero pieces left)
10. If they have not won, switch players
11. Repeat until a player has won
12. Stop the game (win condition met)

Battleship Logic

1. Ask user to choose horizontal or vertical
2. Ask user to input (square to place the ship)
3. Check if the square entered:
    A. Is on the board
    B. Is empty

    C. Can contain a ship of the desired size and orientation
4. If any are false, ask for input again (back to step 2)
5. If all three are true, move on to the next ship
6. Repeat until all ships are placed
7. Once all ships are placed, ask the user for input (where they want to fire)
8. Check if the square entered:
    A. Is on the board
    B. Has not been previously shot at
9. If either is false, ask for input again (back to step 7)
10. If both are true and the square is empty, draw a “miss” symbol
11. If both are true and the square contains a ship, draw a “hit” symbol
12. If a hit and sinks a ship, display a message showing which ship was sunk
13. Check if the player has won (player has hit all ships)
14. Computer randomly guesses a square on the user’s board
15. Check if the computer has won (computer has hit all ships)
16. Repeat until either the player or computer has won
17. Stop the game (win condition met)

<u>Connect Four Logic</u>

1. Ask user for input (column they want to place piece)
2. Check if the number entered:
    A. Is between 1 and 7 inclusive
    B. Is not full
3. If either is false, ask for input again (back to step 1)
4. If both are true, place piece in that column
5. Check if the player who has just moved has won the game (four in a row in any direction)
6. If they have not won, switch players
7. Repeat until a player has won
8. Stop the game (win condition met)

TESTING AND DEBUGGING

The testing for our program was done during the process of programming. However, there are certain techniques we used to test various aspects of our program.

For testing loops, if statements, try-catch blocks, and other control structures, we used system.out.println(). This allowed us to see when the program reached certain sections of code. In times when the wrong if statement was being executed, this allowed us to pinpoint exactly where we needed to make changes. Using console output is also useful for testing the current value of a variable, which can often indicate why a certain portion of code is running differently than expected.

Since our program consists of three games, we had a variety of cases that needed to be tested. The most common was for checkers. In checkers, the player has the opportunity to double and triple jump opponents, which had to be tested thoroughly. However, it is inconvenient to run the entire game and play through a real checkers match to test one case. So, we made edits to the way the game board was initialized to specifically check cases. We made use of the same technique in battleship. Since it would take too long to randomly guess the location of computer ships, we set them to become visible for the purpose of testing the pop-up messages that tell the user which ship has been sunk. This made it much easier to test each ship individually, as we could clearly see where the ships were located.

In addition to testing very specific aspects of code, we also did testing casually, running through games on a regular basis to discover the multitude of bugs that we fixed. We would enter random coordinates in all three games to see how the computer would react. Sometimes, our program was already engineered to handle situations. Other's it would abruptly crash, which was frustrating, but allowed us to discover what parts we needed to focus on fixing.

MAINTENANCE/REFLECTIONS

Working with a partner was a new experience for both of us. However, it allowed us to discover all the benefits of coding with another individual. Firstly, we were able to conquer twice as much at once. Ethan often handled a lot of the logic-based, or back end, aspects of our program, while Nathan often dealt with the appearance and bug testing, or front end aspects. Secondly, each partner was able to work to his personal strengths and leave their weaker parts for the other person to do. This came in handy numerous times. Thirdly, having a set of fresh eyes helped solved countless numbers of bugs. Often, one of us would be working on something for hours, and it is impossible to discover the problem. The other person would then take a look at the program for two minutes, and discover a 1 in place of a 0, which had been causing all the problems. In the end, it was a very beneficial partnership that allowed us to produce a much better program.

In terms of project maintenance, there is one consistent bug that we have been unable to fix. On the starting screen for all three games, there is a chance that clicking a button will do the wrong thing. For example, clicking instructions will start the game, or sometimes even close the program. We are unable to discover why this is the case, as we know our if statements have sound logic. It is likely that it has to do with the mouseListener interface, but we are not sure. However, this has been the only problem, and the games themselves all work seamlessly.