FRUSTRATION

A SOFTWARE ENGINEERING DESIGN PROJECT

ASSIGNMENT 2017: GROUP PROJECT

GROUP
Rubab Ramzan - S00162293
Brain Mc Gowan - S00165159
Bryan Kerruish - S00173160

Table of Contents

Project Overview

For this project, we decided upon developing an application to replicate the board game Frustration. Frustration is a simple board game in which players compete to be the first to send four pieces all the way around a board and is a version of the game Ludo.

This game has a concise set of rules and this documentation allowed for precise and accurate requirements gathering.

From the game's documentation, we were able to generate a series of user stories and use case templates.

To further ensure we had captured the requirements fully, we developed a program flow chart and a number of process flow charts to cover the main methods.

Part 1:

Draft Class Diagram

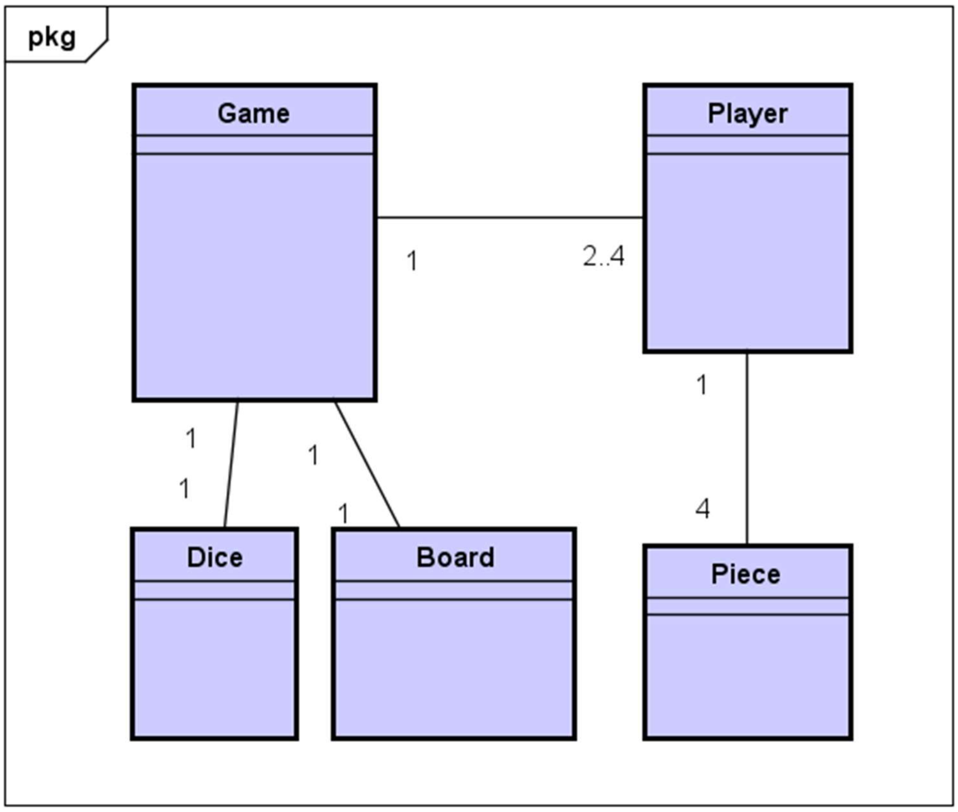


Figure 1 - Draft Class Diagram

Draft Class Diagram Notes

Game Class:

This class will contain the game logic.
It will instantiate two to four players depending on user selection; a board and a dice.

- The game will have one board.
- The game will have two to four players.
- The game will have one dice.

Board Class:

The board will contain the game logic for the board positions, home position and the pieces' locations.

- The game will have one board.

Player Class:

The player class will contain the logic of where the player starts on the board and the pieces they have. The player will instantiate a list containing four pieces.

- The game will have two to four players.
- The player will have four pieces.

Piece Class:

The piece will contain the logic for moving around the board and its state within the game.

- Each player will have four pieces.

Dice Class:

The dice will contain the logic to generate a random number in the range of one to six.

- The game will have one dice.

State Diagrams

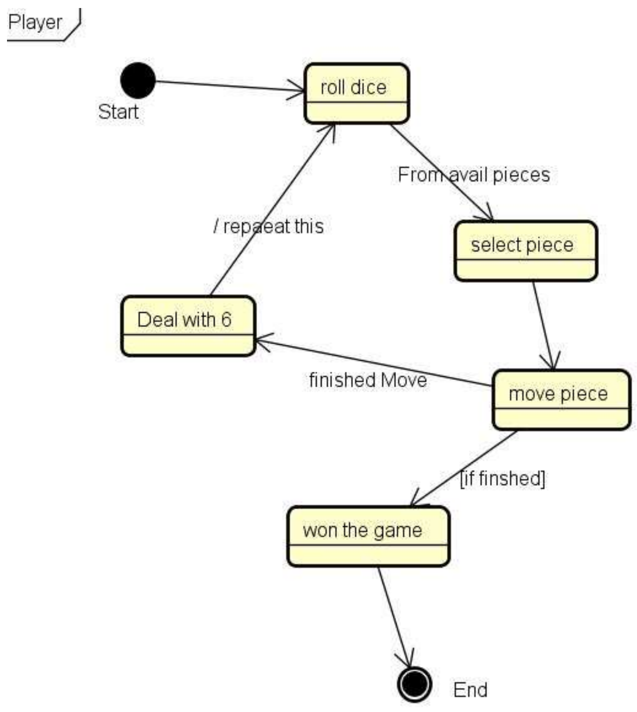


Figure 2 - Player State Diagram

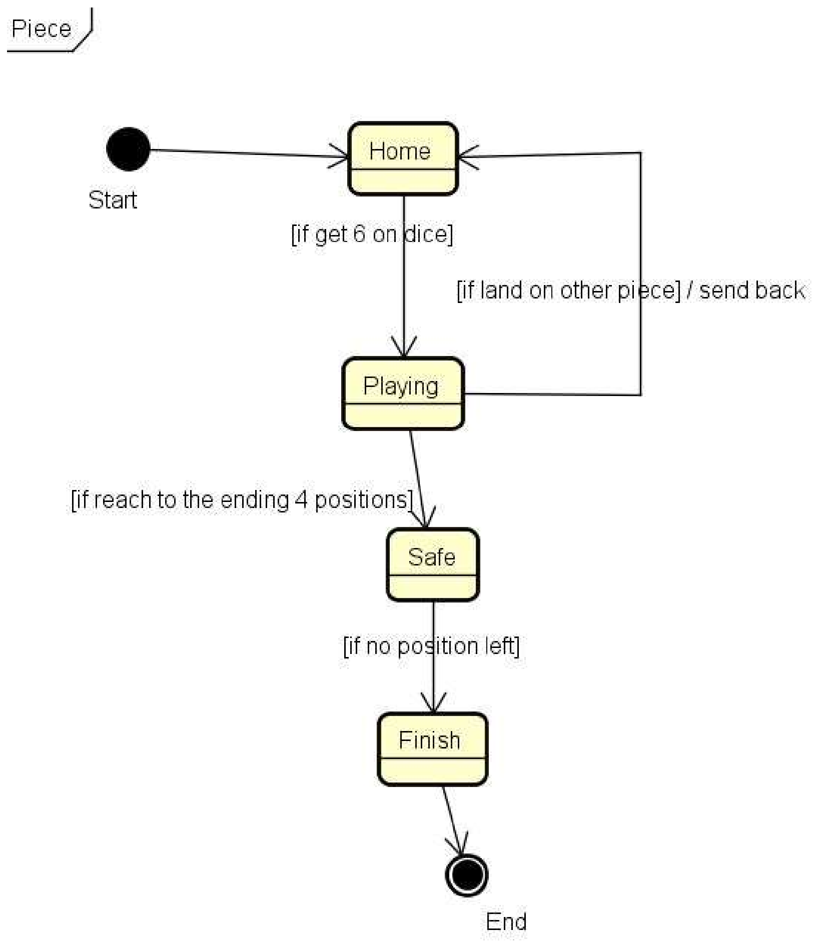


Figure 3 - Piece State Diagram

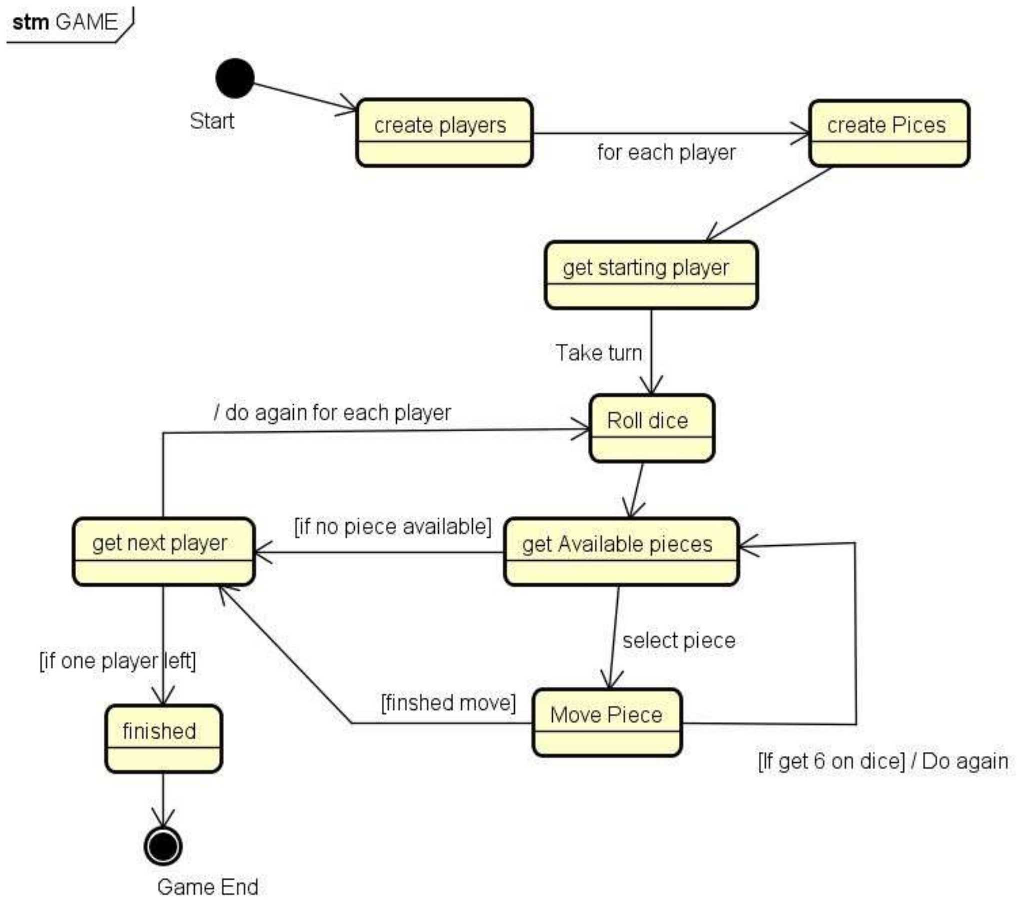


Figure 4 - Game State Diagram

Part 2

Method (Operation) Specification

Take Turn Methods:

Roll the dice

1. User selects to roll the dice.
2. Initial roll again Boolean is set to false (if a player rolls a six, they can roll again).
3. The roll method of the dice is called and the dice value is set.
4. If a six was rolled, we will also set the roll again Boolean to true.

```
public void RollDice()
        {
            RollAgain = false;
            DiceValue = dice.Roll();
            if (DiceValue == 6)
                RollAgain = true;
        }
```

Display Available Pieces

1. The get available pieces method for the current player is called and the available pieces list is populated.
2. Returns a list of available pieces.

```
public List<Piece> DisplayAvailablePieces()
        {
            try
            {
                List<Piece> availablePieces = CurrentPlayer.GetAvailablePieces(DiceValue);
                return availablePieces;
            }
            catch (NullReferenceException ex)
            {
                return null;
            }
        }
```

Move Piece

1. Method accepts a piece and changes its position on the game's board.
2. The piece will call its move method with the dice value as an argument.
3. The board will call its move method with the piece, dice value and players offset as arguments.
4. If the moved piece is piece is not returned, we call the return home method on the moved piece.

```
public void MovePiece(Piece p)
        {
            p.Move(DiceValue);
            Piece returned = board.Move(p, DiceValue, CurrentPlayer.Offset);
            if (returned != null)
            {
                returned.ReturnHome();
            }
        }
```

Set Next Player

1. Sets the current player based on its index position and increments the index.
2. If the player is positioned last in the index, the player index is reset to 0.

```
public void SetNextPlayer()
        {
            if (playerIdx == (Players.Count))
            {
                playerIdx = 0;
            }
            CurrentPlayer = Players[playerIdx];
            playerIdx++;
        }
```

Decision Charts

Draft Expanded Take Turn Decision Chart

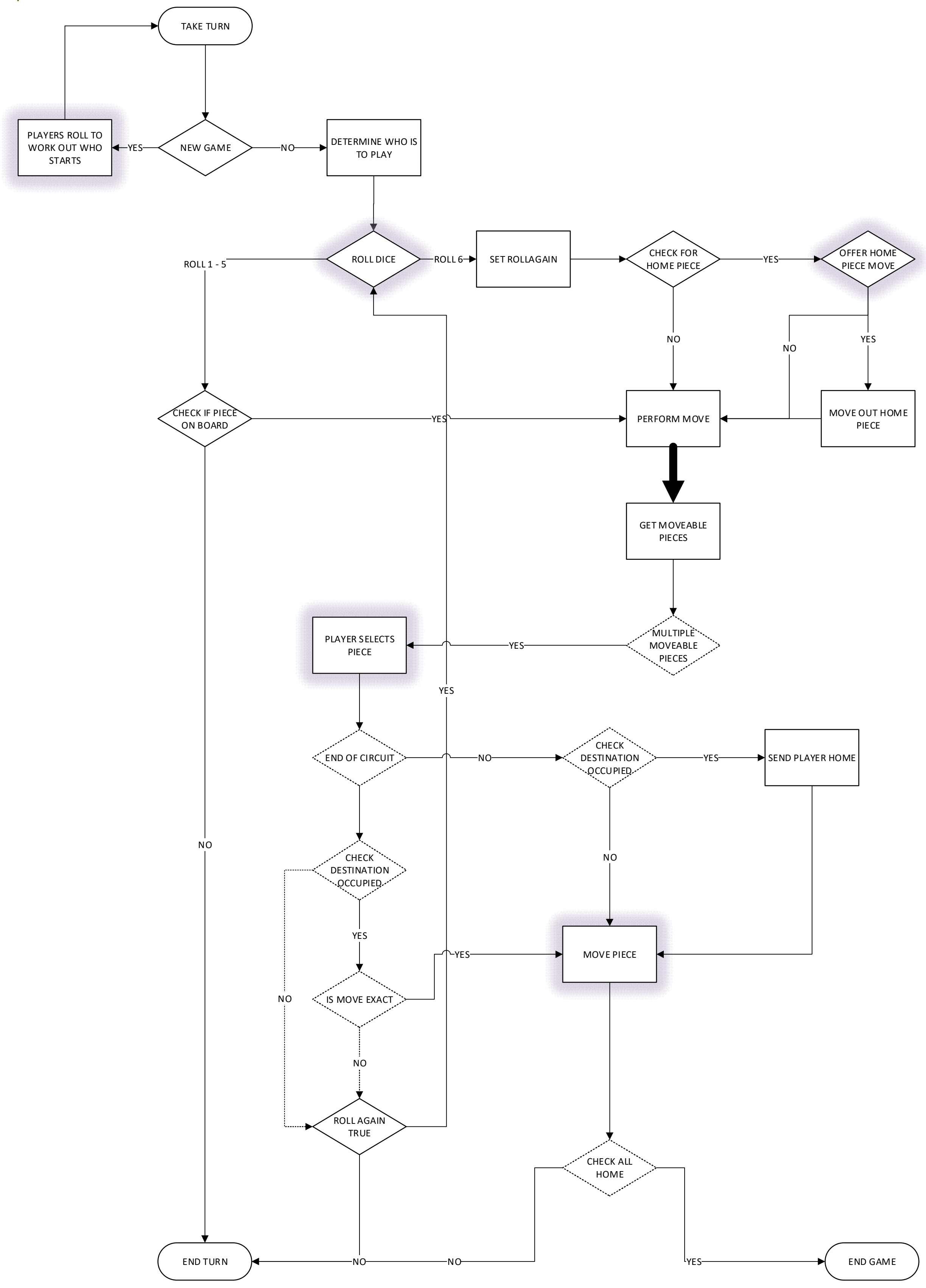


Figure 5 - Game Decision Chart

Final Take Turn (Simple) Decision Chart

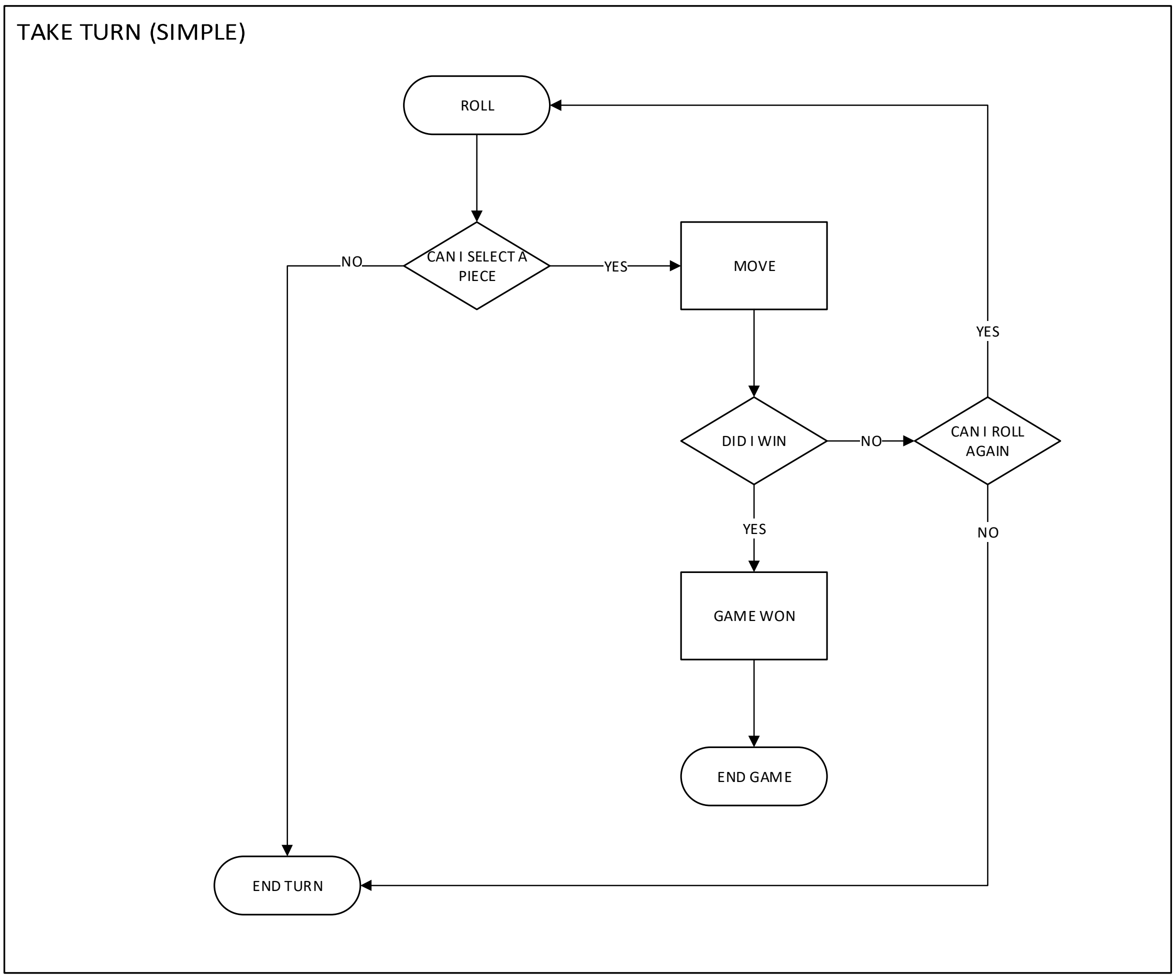


Figure 6 - Simplified Take Turn

Part 3

Detailed Class Diagram

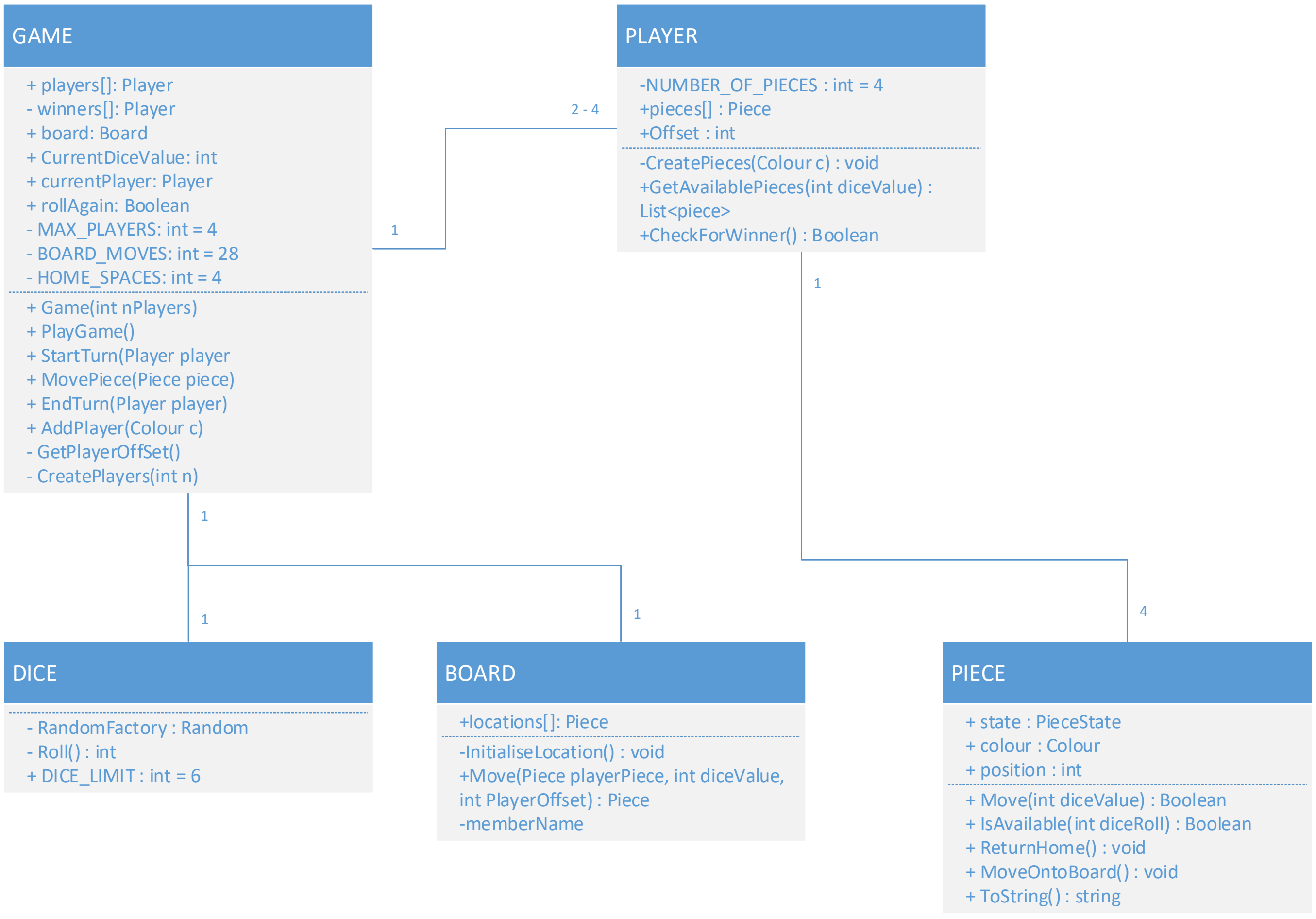


Figure 7 - Detailed Class Diagram

Part 4

Implementation and Testing of Classes in C#

Class: Game

```
class Game
    {
        public List<Player> Players { get; private set; }
        public List<Player> Winners { get; private set; }
        public Boolean RollAgain { get; private set; }

        public Player CurrentPlayer { get; private set; }
        private int playerIdx;
        public int DiceValue { get; private set; }
        public Board board { get; private set; }

        const int MAX_PLAYERS = 4;
        internal const int BOARD_MOVES = 28;
        internal const int HOME_SPACES = 4;
        Dice dice;

        public Game(int nPlayers)
        {
            Players = new List<Player>();
            Winners = new List<Player>();
            dice = new Dice();
            board = new Board();
            CreatePlayers(nPlayers);
            RollAgain = false;
            playerIdx = 0;
        }

        public void SetNextPlayer()
        {
            if (playerIdx == (Players.Count))
            {
                playerIdx = 0;
            }
            CurrentPlayer = Players[playerIdx];
            playerIdx++;
        }


        public void MovePiece(Piece p)
        {
            p.Move(DiceValue);
            Piece returned = board.Move(p, DiceValue, CurrentPlayer.Offset);
            if (returned != null)
            {
                returned.ReturnHome();
            }
        }

        public void RollDice()
        {
            RollAgain = false;
            DiceValue = dice.Roll();
            if (DiceValue == 6)
                RollAgain = true;
        }


        public List<Piece> DisplayAvailablePieces()
        {
            try
            {
                List<Piece> availablePieces = CurrentPlayer.GetAvailablePieces(DiceValue);
                return availablePieces;
            }
            catch (NullReferenceException ex)
            {
                return null;
            }
        }

        private int GetPlayerOffset()
        {
            int offset = 0;
            switch (Players.Count)
            {
                case 1:
                    offset = 7;
                    break;
                case 2:
                    offset = 14;
                    break;
                case 3:
                    offset = 21;
                    break;
                default:
                    offset = 0;
                    break;
            }
            return offset;
        }

        private void CreatePlayers(int n)
        {
            Colour[] colours = { Colour.Blue, Colour.Green, Colour.Red, Colour.Yellow };
            for (int i = 0; i < n; i++)
            {
                Players.Add(new Player(colours[i], GetPlayerOffset(), "Player " + (i + 1)));
            }
        }

    }
```

Class: Player

```
class Player
    {
        const int NUMBER_OF_PIECES = 4;

        public List<Piece> pieces { get; private set; }
        public int Offset { get; private set; }
        public String Name { get; private set; }

        public Player(Colour c, int offset, string name)
        {
            //record colour in player
            pieces = new List<Piece>();
            Name = name;
            Offset = offset;
            CreatePieces(c);
        }


        private void CreatePieces(Colour c)
        {
            for (int i = 0; i < NUMBER_OF_PIECES; i++)
            {
                pieces.Add(new Piece(c));
            }
        }

        public List<Piece> GetAvailablePieces(int diceValue)
        {
            List<Piece> availablePieces = new List<Piece>();
            foreach (var item in pieces)
            {
                if (item.IsAvailable(diceValue))
                {
                    availablePieces.Add(item);
                }
            }
            return availablePieces;
        }

        public Boolean CheckForWinner()
        {
            Boolean hasWon = false;
            hasWon = pieces.Any(p => p.State.Equals(PieceState.Finish));
            return hasWon;
        }
    }
```

Class: Board

```
class Board
    {
        public Piece[] locations { get; private set; }

        public Board()
        {
            InitialiseLocations();
        }

        private void InitialiseLocations()
        {
            locations = new Piece[Game.BOARD_MOVES];
        }

        public Piece Move(Piece playerPiece, int diceValue, int playerOffset)
        {
            int absoluteMove = playerPiece.Position + playerOffset;
            //moving around the board from 28 round to 0 again
            if (absoluteMove > locations.Length)
            {
                absoluteMove -= locations.Length;
            }
            //handle out of bounds exception if offset is 0
            if (absoluteMove - diceValue > 0)
                locations[absoluteMove - diceValue] = null;
            Piece boardPiece = locations[absoluteMove];

            locations[absoluteMove] = playerPiece;
            //if a piece has moved to home, remove from the board
            return boardPiece;

        }

        public List<String> DisplayBoard()
        {
            List<String> l = new List<String>();
            for (int i = 0; i < locations.Length; i++)
            {
                Piece p = locations[i];
                String colour = "Empty";
                String pieceLocation = "Empty";
                if (p != null)
                {
                    colour = p.Colour.ToString() ?? "Empty";
                    pieceLocation = p.Position.ToString() ?? "Empty";
                    l.Add(String.Format("{0} Relative Location: {1} Board Loaction: {2}", colou
r, pieceLocation, i));

                }
                else
                    l.Add(String.Format("{0} Empty", i));
            }
            return l;
        }




    }
```

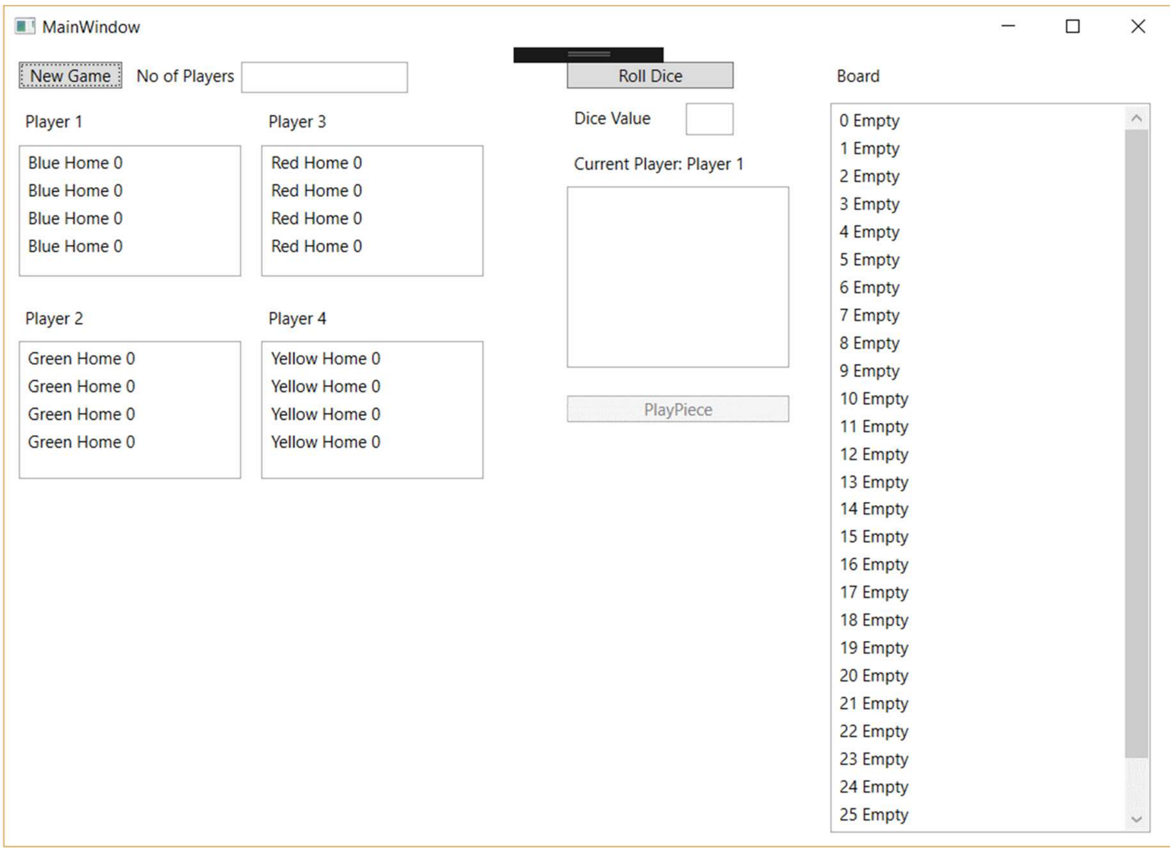
Class: Piece

```
enum PieceState { Playing, Home, Safe, Finish }
    enum Colour { Red, Yellow, Green, Blue }
    class Piece
    {

        public PieceState State { get; private set; }
        public int Position { get; private set; }
        public Colour Colour { get; private set; }

        public Piece(Colour pc)
        {
            ReturnHome();
            Colour = pc;
        }

        public Boolean Move(int diceValue)
        {
            Boolean complete = false;
            //if 6 and home
            if (diceValue == 6 && this.State.Equals(PieceState.Home))
            {
                complete = MoveOntoBoard();
            }
            else if (Position + diceValue <= Game.BOARD_MOVES)
            {
                Position += diceValue;
                complete = true;
            }
            else if (Position + diceValue <= Game.BOARD_MOVES + Game.HOME_SPACES)
            {
                State = PieceState.Safe;
                Position += diceValue;
                complete = true;
            }

            return complete;
        }

        public Boolean IsAvailable(int diceRoll)
        {
            bool available = false;

            if (State.Equals(PieceState.Home) && diceRoll == 6)
            {
                available = true;
            }

            else if ((State.Equals(PieceState.Playing) || State.Equals(PieceState.Safe)) && Pos
ition + diceRoll <= Game.BOARD_MOVES + Game.HOME_SPACES)
            {
                available = true;
            }
            return available;
        }

        public void ReturnHome()
        {
            State = PieceState.Home;
            Position = 0;
        }
        public bool MoveOntoBoard()
        {
            State = PieceState.Playing;
            Position = 1;
            return true;
        }

        public override string ToString()
        {
            return String.Format("{0} {1} {2}", Colour, State, Position);
        }

    }
```

Class: Dice

```
class Dice
    {
        static Random RandomFactory;
        const int DICE_LIMIT = 6;
        public Dice()
        {
            RandomFactory = new Random();
        }
        public int Roll()
        {
            return RandomFactory.Next(1, DICE_LIMIT + 1);
        }
    }
```

Testing Interface:


MainWindow

New Game | No of Players

Roll Dice

Board

Player 1
Blue Home 0
Blue Home 0
Blue Home 0
Blue Home 0

Player 3
Red Home 0
Red Home 0
Red Home 0
Red Home 0

Dice Value

Current Player: Player 1

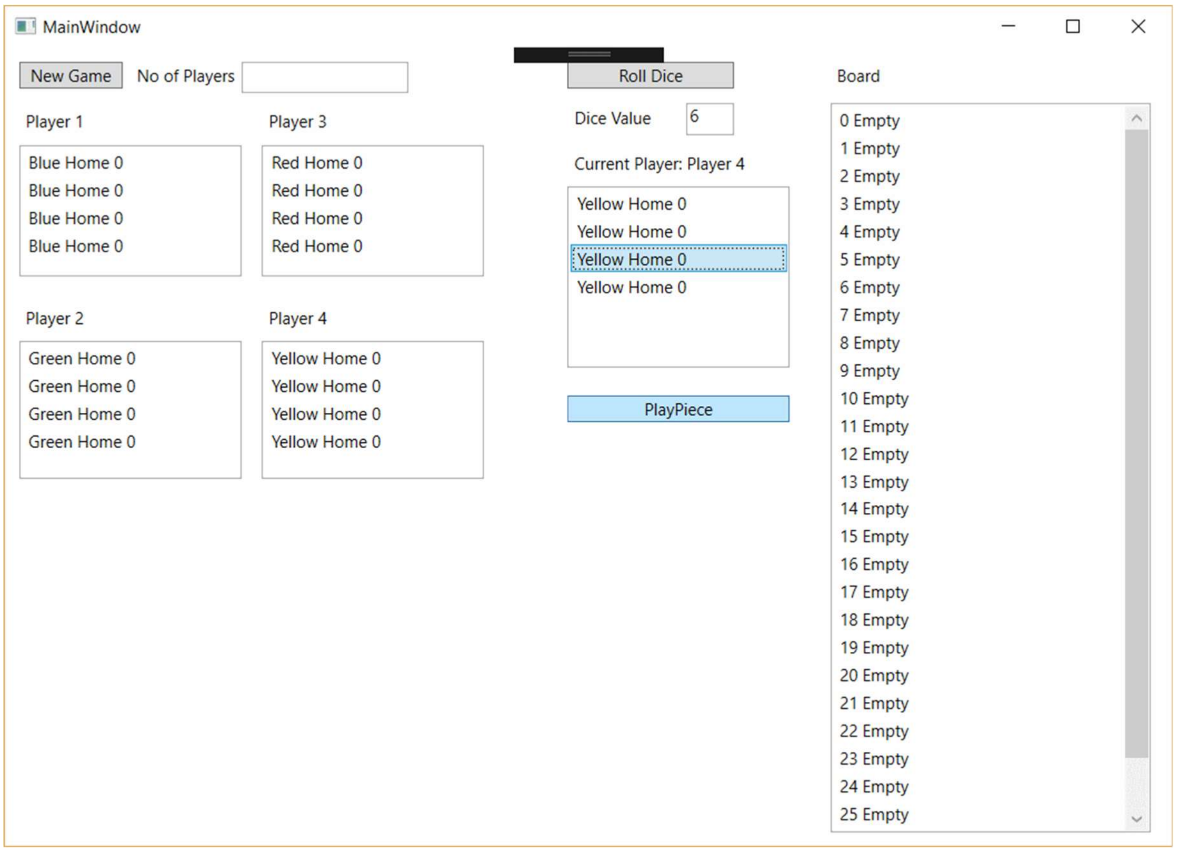
Player 2
Green Home 0
Green Home 0
Green Home 0
Green Home 0

Player 4
Yellow Home 0
Yellow Home 0
Yellow Home 0
Yellow Home 0

PlayPiece

0 Empty
1 Empty
2 Empty
3 Empty
4 Empty
5 Empty
6 Empty
7 Empty
8 Empty
9 Empty
10 Empty
11 Empty
12 Empty
13 Empty
14 Empty
15 Empty
16 Empty
17 Empty
18 Empty
19 Empty
20 Empty
21 Empty
22 Empty
23 Empty
24 Empty
25 Empty

Figure 8 - Testing Interface 1



Player 1

Blue Home 0
Blue Home 0
Blue Home 0
Blue Home 0

Player 3

Red Home 0
Red Home 0
Red Home 0
Red Home 0

Player 2

Green Home 0
Green Home 0
Green Home 0
Green Home 0

Player 4

Yellow Home 0
Yellow Home 0
Yellow Home 0
Yellow Home 0

Dice Value 6

Current Player: Player 4

Yellow Home 0
Yellow Home 0
Yellow Home 0
Yellow Home 0

Board

0 Empty
1 Empty
2 Empty
3 Empty
4 Empty
5 Empty
6 Empty
7 Empty
8 Empty
9 Empty
10 Empty
11 Empty
12 Empty
13 Empty
14 Empty
15 Empty
16 Empty
17 Empty
18 Empty
19 Empty
20 Empty
21 Empty
22 Empty
23 Empty
24 Empty
25 Empty

Figure 9 - Testing Interface 2

Part 5

Code and Walkthrough Notes

Decoupling the interface from the application logic.

The current interface is designed purely for testing. The interface has been decoupled from the logic / application layer.

DRY principles applied.

The principle of "Do Not Repeat Yourself" has been integrated from the beginning. As the code developed we moved any duplicated (or partially duplicated) elements into common components / classes.

Naming conventions.

The code has been written in camelCase and names have been as descriptive as possible.

SOLID principle applied.

Despite the relatively small size of this application, we have endeavoured to apply the SOLID (Single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) principles to allow this application to be easily maintainable and updateable.

Code structure.

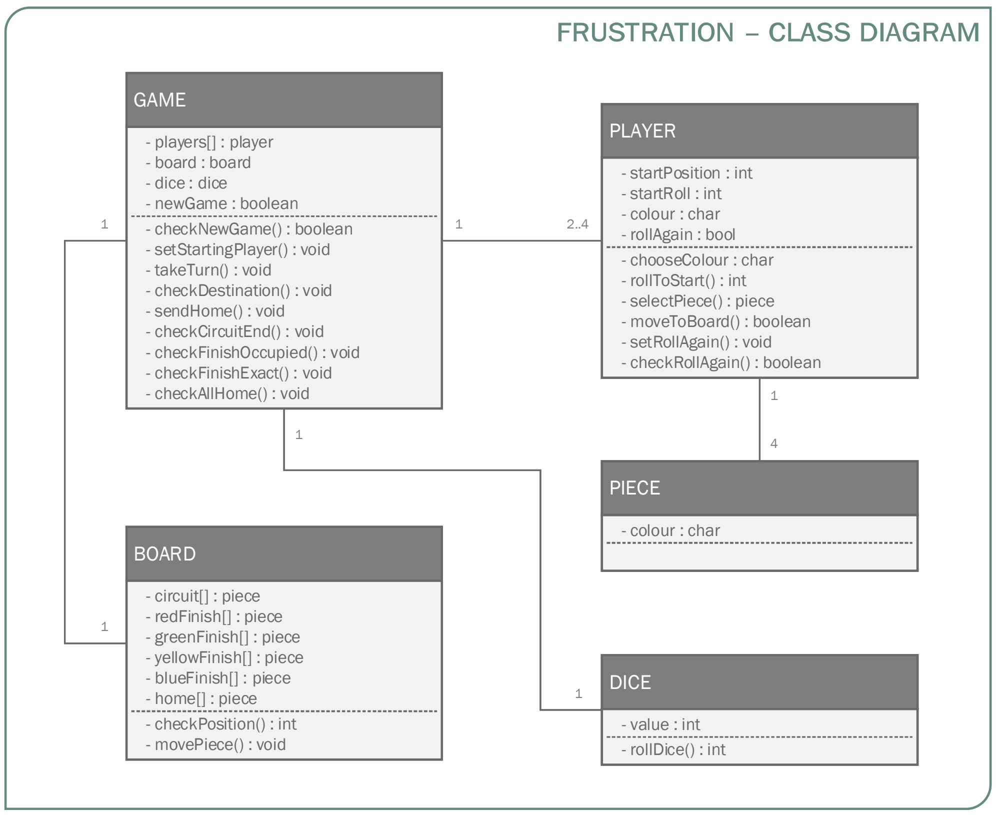
The code has been developed as part of a team using Visual Studio and GitHub for source control.

Code readability and maintainability.

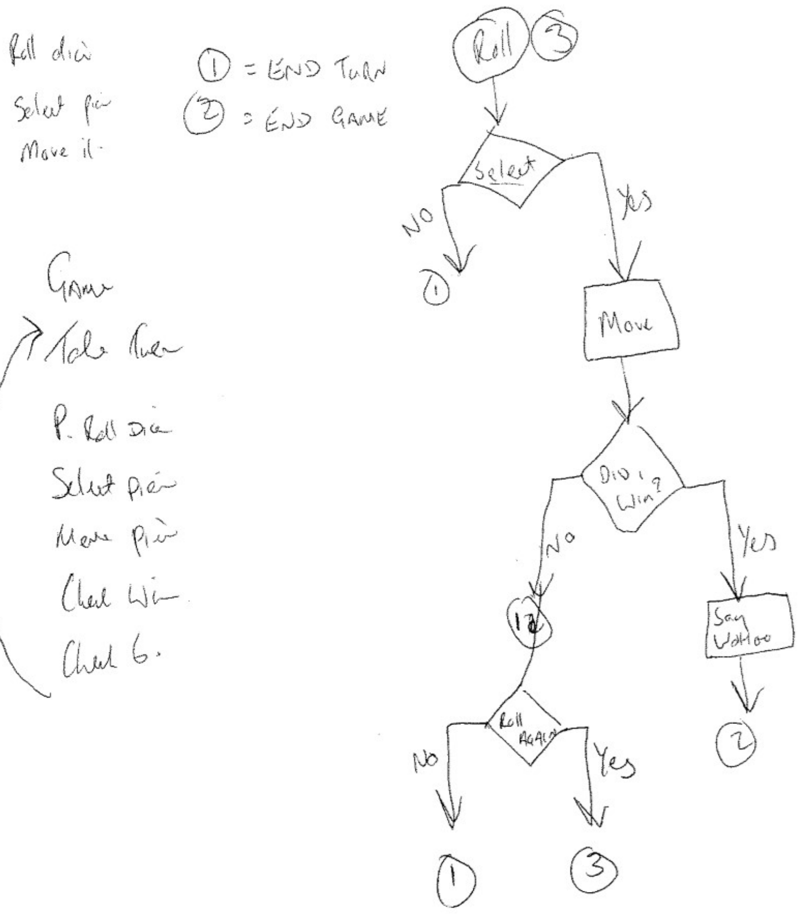
The code readability and maintainability has been a design decision from the offset. The code is commented except for obviously self-documenting code.

Appendix

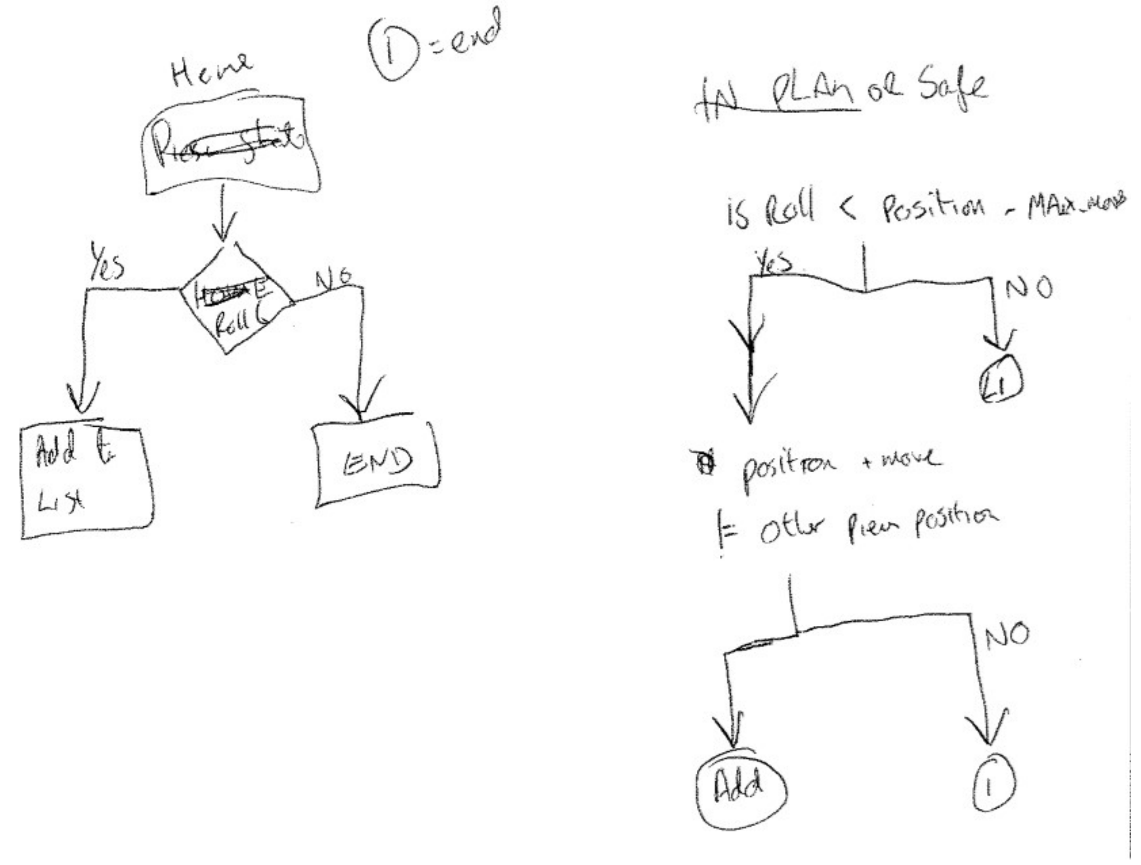
Additional Notes



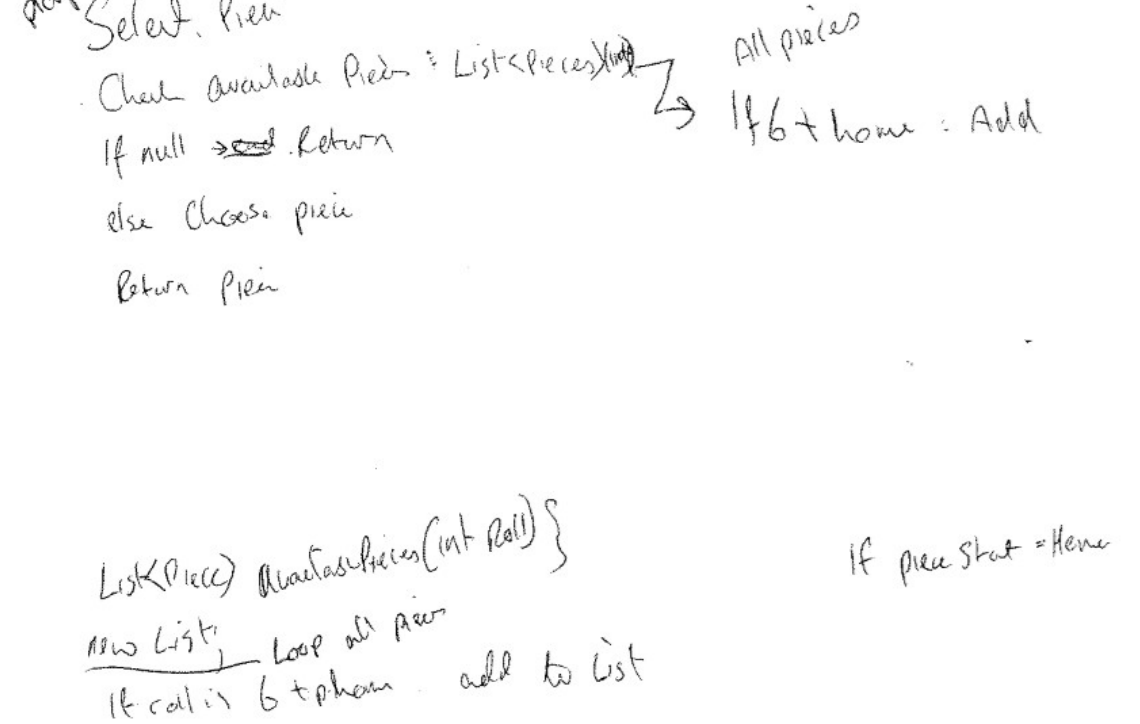
Draft Class Diagram



Design Notes



Design Notes



player:
Select. Piece
Check available Pieces : List<Pieces>(int) → All pieces
If 6 + home : Add
If null ~~end~~ Return
else Choose piece
Return Piece

List<Piece> AvailablePieces(int Roll) }
new List; Loop all pieces
If roll is 6 + at home add to List

If piece Stat = Home

Design Notes