Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

Viking Chess Using MCTS

Design Document

Declan Murphy – C00106936
Supervisor: Joseph Kehoe
2016

Contents

1. Introduction

1.1. About this Document

Design of the application began in late 2015. This document has been modified and updated to reflect the latest state of the application since this time. This current version reflects the final application being submitted as the 4th Year Project in April 2016.

1.2. Background

The Viking Chess application is a recreation of the variety of ancient Viking board games collectively referred to as Tafl games. This implementation of the game uses the Monte Carlo Tree Search (MCTS) algorithm in an effort to create a competent AI player for the user to play against.

Due to the lack of early-game and end-game playbooks for Tafl, the large branching factor of its game tree, and the difficulty in scoring non-terminal board states, the MCTS algorithm appears to be a sensible approach for creating an AI player; for much the same reasons as it has shown success in creating AI players for the game of Go.

The potential user base for the game is anyone interested in learning about and playing the ancient Tafl board game. In addition, those looking to research the MCTS algorithm may find the application of interest as a case study for the effectiveness of the algorithm.

1.3. Purpose

The purpose of this document is to describe the implementation of the Viking Chess application. The Viking Chess application is intended to allow a user to play a Tafl game against another user locally on the same machine or against an AI player employing the MCTS algorithm.

1.4. Scope

The Viking Chess application consists of three main components. First, the XAML pages control the implementation of the User Interface. Second, the game objects control the representation and implementation of the game and its necessary objects. Finally, the MCTS class and associated Node class controls the operation of the MCTS algorithm.

2. Architecture

2.1. Introduction

The application will be developed for Windows 10 as a Universal Windows Platform (UWP) application using C#. The application will be designed to run at a resolution of 1920x1080 on a Windows 10 Desktop and will support both mouse and touch controls.

The User Interface will be developed using the Extensible Application Markup Language (XAML) developed by Microsoft.

No outbound or inbound connections will be required by the application as all tasks will be carried out locally on the system the application is installed on.

2.2 Architecture Diagram

2.2.1. Core Architecture

The following diagram shows the core architecture of a UWP application.

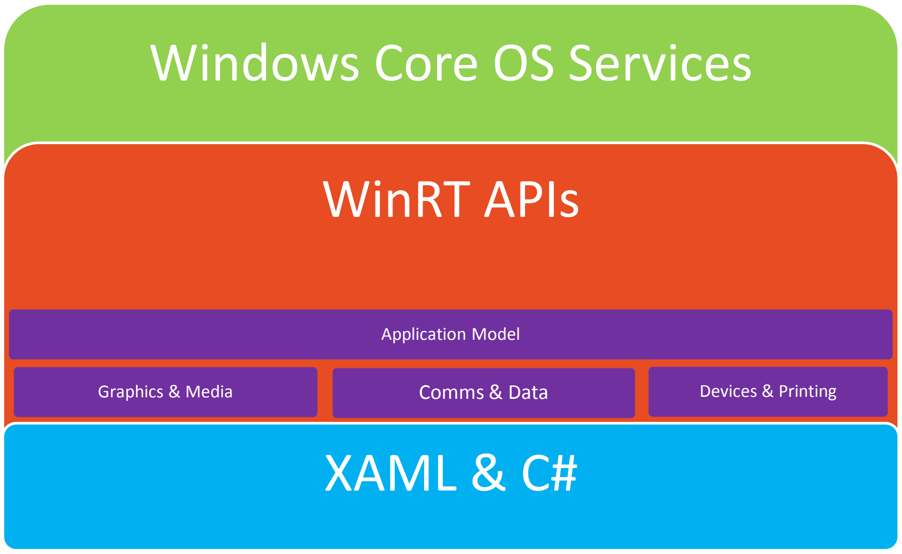

Fig 1 - Windows 10 UWP Application Architecture

2.2.2. Software Architecture

The following diagram shows the software architecture used for the application. The architecture used was the MVVM (Model-View-ViewModel) architecture.

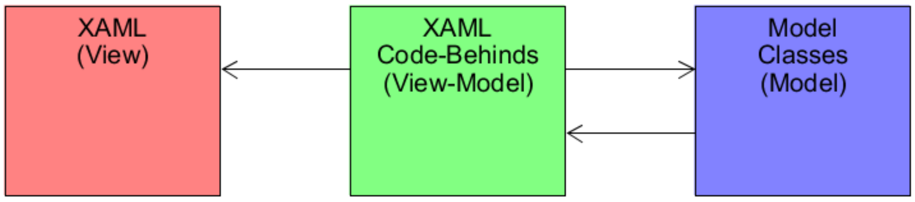

Fig 2 - MVVM Architecture

2.3. Code

2.3.1. Introduction

The following section details the code design for the application. An object-oriented approach was taken for the design of the necessary classes.

2.3.2. Classes

This section provides details on the various classes required by the Viking Chess application in addition to providing the Domain and Class diagrams.

2.3.2.1. Domain Model Diagram

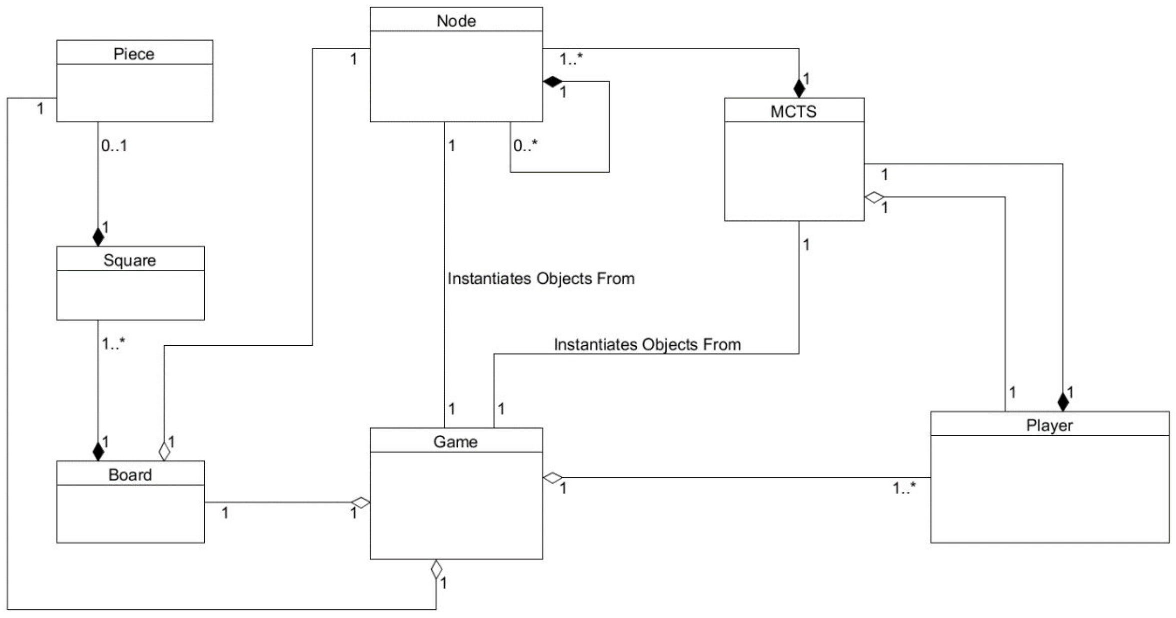

Fig 3 - Domain Model Diagram

2.3.2.2. Class Diagram

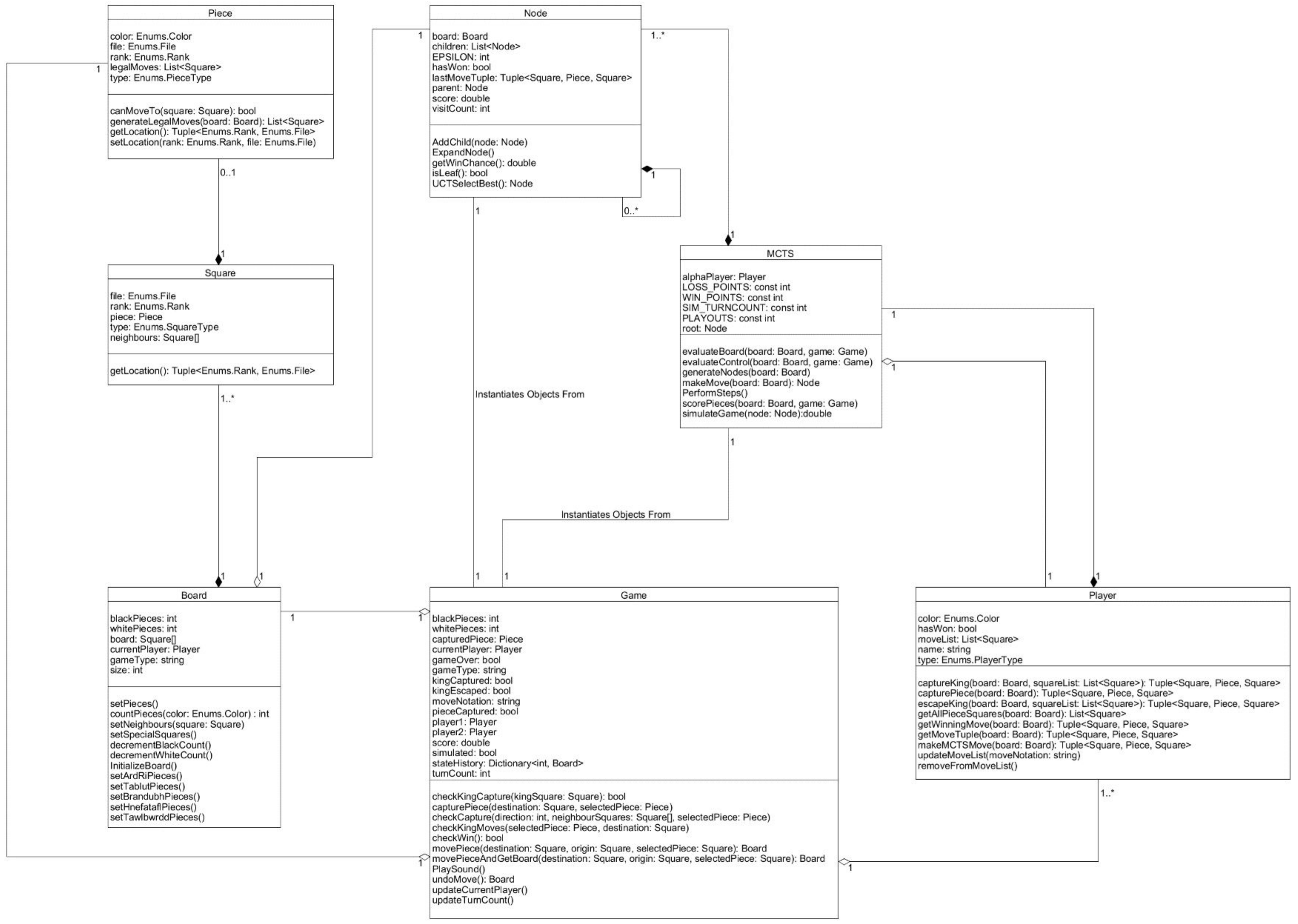
Piece
color: Enums.Color
file: Enums.File
rank: Enums.Rank
legalMoves: List<Square>
type: Enums.PieceType
canMoveTo(square: Square): bool
generateLegalMoves(board: Board): List<Square>
getLocation(): Tuple<Enums.Rank, Enums.File>
setLocation(rank: Enums.Rank, file: Enums.File)
Node
board: Board
children: List<Node>
EPSILON: int
hasWon: bool
lastMoveTuple: Tuple<Square, Piece, Square>
parent: Node
score: double
visitCount: int
AddChild(node: Node)
ExpandNode()
getWinChance(): double
isLeaf(): bool
UCTSelectBest(): Node
Square
file: Enums.File
rank: Enums.Rank
piece: Piece
type: Enums.SquareType
neighbours: Square[]
getLocation(): Tuple<Enums.Rank, Enums.File>
MCTS
alphaPlayer: Player
LOSS_POINTS: const int
WIN_POINTS: const int
SIM_TURNCOUNT: const int
PLAYOUTS: const int
root: Node
evaluateBoard(board: Board, game: Game)
evaluateControl(board: Board, game: Game)
generateNodes(board: Board)
makeMove(board: Board): Node
PerformSteps()
scorePieces(board: Board, game: Game)
simulateGame(node: Node):double
Instantiates Objects From
Instantiates Objects From
Board
blackPieces: int
whitePieces: int
board: Square[]
currentPlayer: Player
gameType: string
size: int
setPieces()
countPieces(color: Enums.Color) : int
setNeighbours(square: Square)
setSpecialSquares()
decrementBlackCount()
decrementWhiteCount()
InitializeBoard()
setArdRiPieces()
setTablutPieces()
setBrandubhPieces()
setHnefataflPieces()
setTawlbwrddPieces()
Game
blackPieces: int
whitePieces: int
capturedPiece: Piece
currentPlayer: Player
gameOver: bool
gameType: string
kingCaptured: bool
kingEscaped: bool
moveNotation: string
pieceCaptured: bool
player1: Player
player2: Player
score: double
simulated: bool
stateHistory: Dictionary<int, Board>
turnCount: int
checkKingCapture(kingSquare: Square): bool
capturePiece(destination: Square, selectedPiece: Piece)
checkCapture(direction: int, neighbourSquares: Square[], selectedPiece: Piece)
checkKingMoves(selectedPiece: Piece, destination: Square)
checkWin(): bool
movePiece(destination: Square, origin: Square, selectedPiece: Square): Board
movePieceAndGetBoard(destination: Square, origin: Square, selectedPiece: Square): Board
PlaySound()
undoMove(): Board
updateCurrentPlayer()
updateTurnCount()
Player
color: Enums.Color
hasWon: bool
moveList: List<Square>
name: string
type: Enums.PlayerType
captureKing(board: Board, squareList: List<Square>): Tuple<Square, Piece, Square>
capturePiece(board: Board): Tuple<Square, Piece, Square>
escapeKing(board: Board, squareList: List<Square>): Tuple<Square, Piece, Square>
getAllPieceSquares(board: Board): List<Square>
getWinningMove(board: Board): Tuple<Square, Piece, Square>
getMoveTuple(board: Board): Tuple<Square, Piece, Square>
makeMCTSMove(board: Board): Tuple<Square, Piece, Square>
updateMoveList(moveNotation: string)
removeFromMoveList()
1
1
1..*
1
0..*
0..1
1
1
1..*
1
1
1
1
1
1
1
1
1
1
1
1
1
1..*
1

Fig 4 – Model Layer Class Diagram

2.3.2.3. Model Layer

The Model layer contains all classes necessary for implementing a game of Viking Chess and the MCTS algorithm.
The following is a list of all classes in the model layer:

Class	Description
Player	Contains the necessary attributes and functions for a player in Viking Chess. Can be an Attacker or Defender, and a Human or a CPU.
Piece	Contains the necessary attributes and functions for a piece in Viking Chess. Can be an Attacker's or Defender's Pawn or a Defender's King. Also contains information on its location on the board.
Square	Contains the necessary attributes and functions for a square on the Viking Chess board. Each Square object can contain a Piece object or none at all and will be one of three types: Regular, Corner or Throne.
Board	Contains the necessary attributes and functions for a Viking Chess board. The board contains an array of Square objects which is configured to reflect the layout of one of five Viking Chess variants.
Game	Contains the attributes and functions necessary for processing a game of Viking Chess. This class serves to process moves and captures, modify the game board, check if the win conditions have been met and return information to the View layer to update the User Interface as the game progresses.
MCTS	Contains the attributes and functions necessary for implementing the MCTS algorithm. The algorithm builds a tree of Nodes and performs the four steps of the MCTS algorithm on the tree for a specified number of playouts.
Node	Contains the attributes and functions necessary for implementing a Node for the MCTS tree. Each node represents a board state in a Viking Chess game.
Enums	Contains various enumerators used to define object properties such as PieceType, PlayerType, Ranks, and Files.

2.3.2.4. *View/View-Model Layer*

The View and View-Model Layers are closely linked in the form of XAML pages and their associated code-behinds. The XAML pages control the User Interface presentation while their associated code-behinds handle the initialization of UI elements, UI updates and receiving information from the model layer.

The following is a list of all classes/XAML pages in the View and View-Model Layer:

Class	Description
MainPage	Contains the UI presentation layout and functions for the home page. This is the page displayed to the user upon launching the application
SettingsPage	Contains the UI presentation layout and functions for the settings page. The settings page is displayed once the user selects to play a new game from the main menu. It allows for the configuration of certain game parameters before the game is loaded and also controls which game variant page to load.
RulesPage	Contains the UI presentation layout and functions for the rules page. This page details the rules of the game to the user.
AboutPage	Contains the UI presentation layout for the about page. This page displays information about the application and the developer to the user.
[GameVariant]Page	There are multiple Game Variant pages. Each page is named in the format [GameVariant]Page e.g. HnefataflPage. These pages display the game to the user while their code-behinds control UI updates and minor processing of information received from the Model Layer.

3. System Sequence Diagrams

3.1 Start New Game

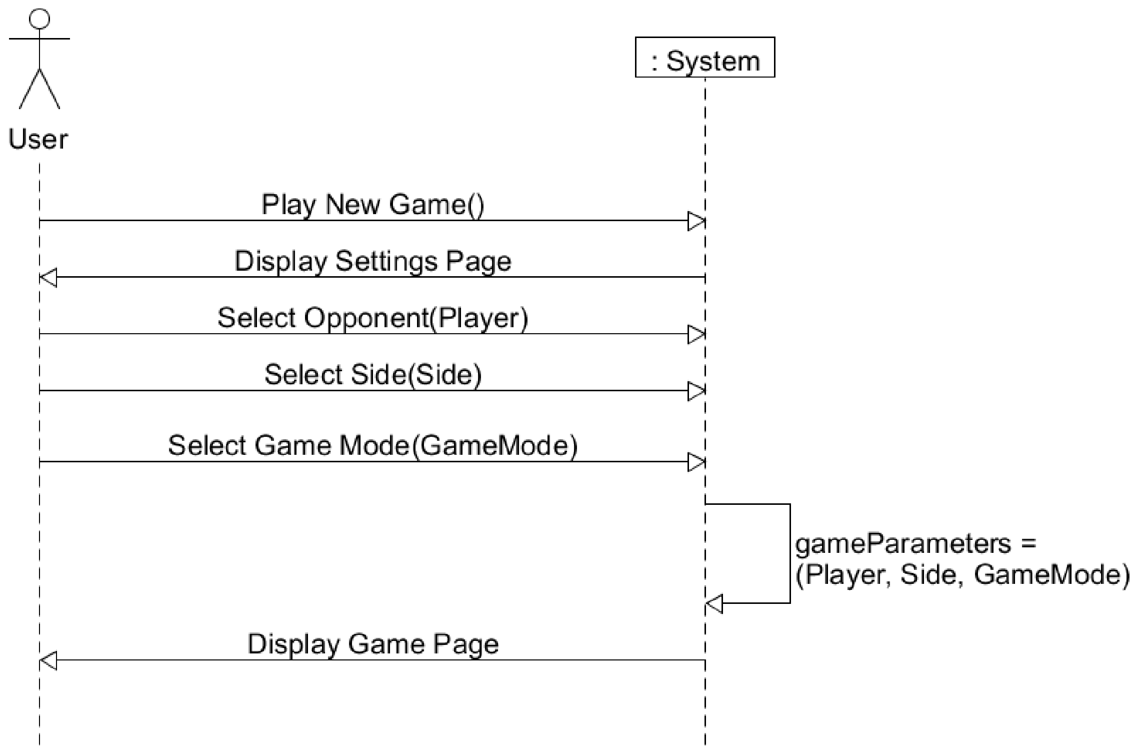

Fig 5 - System Sequence Diagrams for Starting New Game

3.2 Make Move

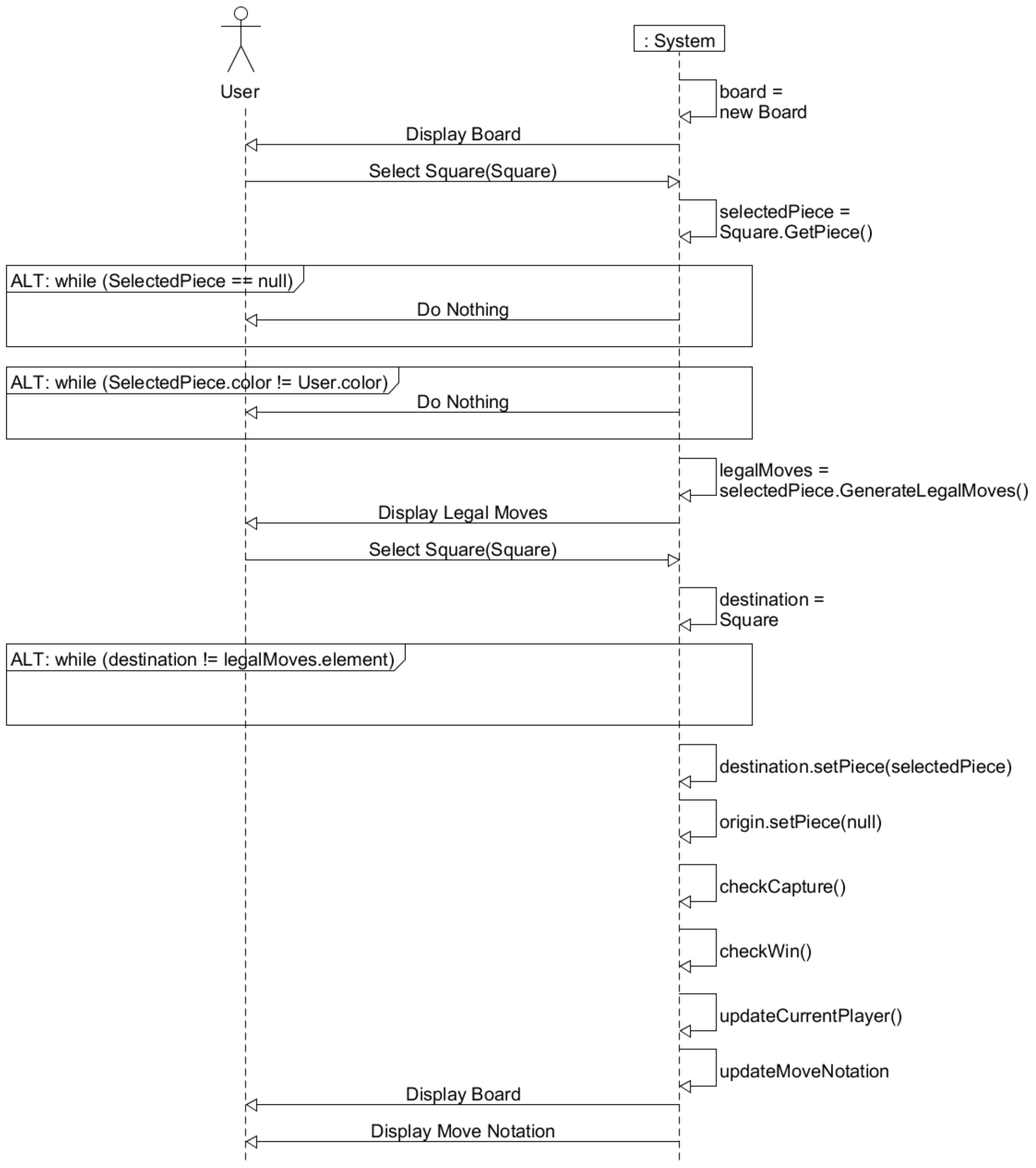

Fig 6 - System Sequence Diagram for a human player making a move.

3.3. View Rules

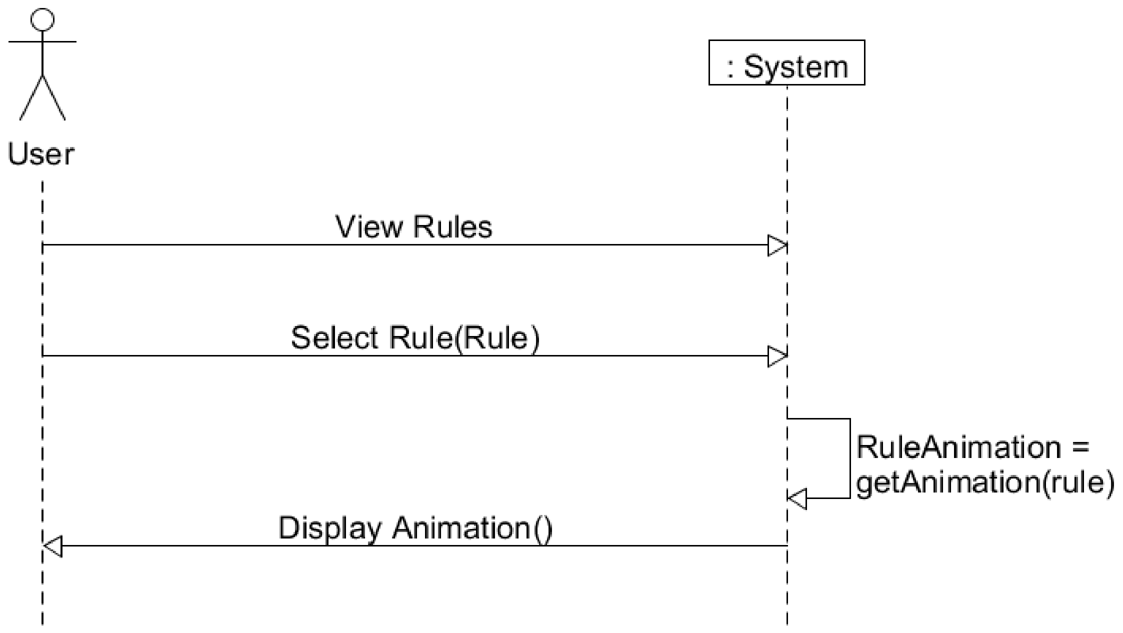

Fig 7 - View Rules System Sequence Diagram

4. Sequence Diagrams

4.1. Make MCTS Move

The following is the Sequence Diagram for when the CPU makes a move using the MCTS algorithm.

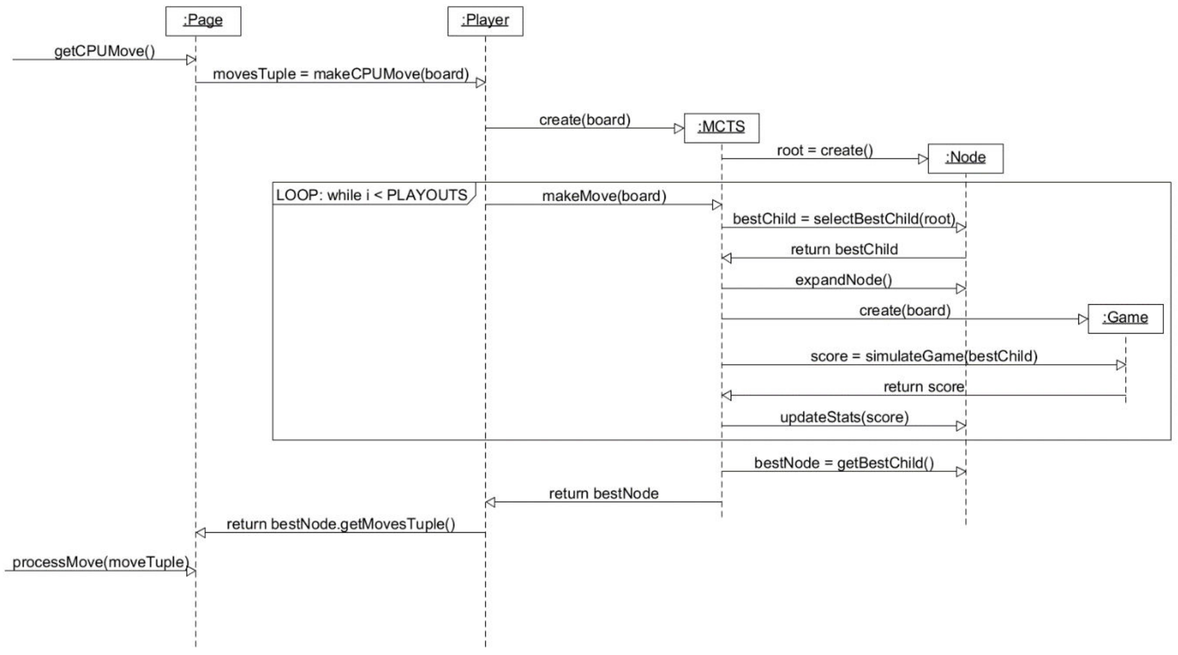

Fig 8 - Make MCTS Move Sequence Diagram