**Software Design  
Document**

for

Game of Chess

Version 1.0 approved

Prepared by Kevin Zarzana

Condor Army

2/10/18

# Contents

[1.1 Contents i](#_Toc507709284)

[1.2 Introduction 1](#_Toc507709285)

[1.3 Purpose 1](#_Toc507709286)

[1.4 System Overview 1](#_Toc507709287)

[1.5 Definitions, Acronyms and Abbreviations 1](#_Toc507709288)

[1.6 Supporting Materials 2](#_Toc507709289)

[1.7 Document Overview 2](#_Toc507709290)

[2 Architecture 2](#_Toc507709291)

[2.1 Overview 2](#_Toc507709292)

[2.2 Component 1..n 2](#_Toc507709293)

[3 High-Level Design 4](#_Toc507709294)

[3.1 View / Model Component 1..n 5](#_Toc507709295)

[3.1.1 Overview . 5](#_Toc507709296)

[*3.1.2 Requirements for Design* 5](#_Toc507709297)

[4 Low-Level Design 7](#_Toc507709298)

[1.1 View / Model Component 1..n 7](#_Toc507709299)

| Version | Primary Author(s) | Description of Version | Date Completed |
| --- | --- | --- | --- |
| Init 1 | Kevin Zarzana | Took IEEE template and built basic document specific to Game of Chess | 02/11/18 |

~~<This template serves as a basis for a Software Design Specification. As in the SRS document, all italics refer to the “comment” style. Comments in blue are general and apply to any SDS, these that are in black are applicable specifically for this course. This template is based on the work by Karl. E Wiegers, Steve McConnel of CXOne group and the IEEE standards.>~~

# Introduction

## Purpose

The Game of Chess program is a program used to play a game. The program allows for a User to play chess against a computer player. The player can select the board pieces presented on a 3-D game board and move the pieces with mouse clicks on the game board.

## System Overview

The Game of Chess begins with a opening title screen giving the option to Play a New Game, Load a Past Game, Change Settings, or Quit. After the New Game button is pressed the program will load the Game Board and pieces in the starting positions. If the Load Game button is pressed a list of previously saved sessions will be presented and the User will be able to make a selection of which game to load.

~~<Brief high-level description of system structure, functionality, interactions with external systems, system issues, etc.~~

## Definitions, Acronyms and Abbreviations

**Chess** – A game in which 2 players move game pieces in an attempt to eliminate the other opponents King piece.

**Piece** – Standard game piece, used as a base Class for rest of the Chess game pieces. Moves along the 8 x 8 chess grid board.

**Queen –** Most powerful Chess game piece. Moves an unlimited range in every direction as long as no enemy blocks its path.

**King** – Most important Chess game piece. Moves in all directions exactly 1 spot. The objective of this game is to get the opponents King piece in “checkmate” (Defined further down).

**Knight** – Chess game pieces that move in an L-pattern in any direction. There are two of these pieces per player.

**Bishop** – Chess game pieces that move diagonally an unlimited range unless an opponent blocks the path. There are two of these pieces per player.

**Rook –** Chess game pieces that move vertically and horizontally an unlimited range unless blocked by an opponent. There are two of these pieces per player.

**Pawn –** Chess game pieces that stand in the front line at the games start. These pieces can only move forward 1 space (or 2 if its the piece’s first move) and attack diagonally (forward). This piece cannot go backwards. There are eight of these pieces per player.

**Chess Board –** An 8x8 Black & White grid that the Game of Chess is played on.

**Player** – Computer or human who controls game pieces actions.

**Check** – A Game Status which indicates your King piece is under direct line of fire. Player who is in “check” must move King out of harm’s way this turn or risk losing the game.

**Checkmate** – Ultimate version of “Check”, this means the King piece is under direct line of fire AND there are no possible ways to protect the King. This signals a loss for the player and results in an end of game.

**Attack –** Common chess piece move. All chess pieces can “attack” in which they use their defined movement pattern to end on an opponents piece thus knocking them off the game board and out of play.

**Move Timer** – A countdown timer showing how much time the user must move.

## Supporting Materials

<Note any references or related materials here.

## Document Overview

# Architecture

<The architecture provides the top level design view of a system and provides a basis for more detailed design work. This is the section where you should include your High-Level design Component Diagram.

# Overview

The GameManager class will create the pieces, and assign them the required starting positions. It will also track the pieces movement throughout the game in a two dimensional array. It will also include all the necessary functions for gameplay, including the check(), checkmate(), stalemate and playGame() functions. Once the board is setup and ready for play, the playGame() function will be called, and the remainder of the game will be driven by a while loop inside of this function.

The pieces will all extend the ChessPiece class. The ChessPiece class will include a getPossibleMove() function that is overridden in the individual piece classes. This allows the function to be called with the superclasses type, as is necessary when manipulating the gameplay via an array.

# Component 1..n

The primary compenents of Chess will consist of classes for each of the unique chess pieces (i.e. king, rook, knight, queen, pawn and Bishop). There will also be a unique class to check for specific game conditions, such as check and checkmate. Should the movement of a piece mean the activation of a check condition against the active/current player, the move or attack will be prohibited. For example, if an opposing rook is directly in front of the king, impeded only by the players queen, then the queen shall not be allowed to move in a direction that will leave the king vulnerable to the rook’s attack.

The individual piece classes will provide movement and attack abilities for each of the pieces. With the exception of pawn class, the attack for each piece is handled much in the same way as the move function, so an additional class for attack may not be necessary.

2.2.1 – Piece Classes

**Pawn -** The pawn class will need to include a Boolean variable, *hasMoved*, that will change from false to true after the individual instance of the pawn class has moved the first time. This *hasMoved* variable will be read each time the pawn is selected for movement. The first time any instance of the pawn is moved, it will be allowed to move two square forward on the playing surface. After the first move, *hasMoved* will be changed to true. Everry subsequent time that the pawn is moved, the true value in the *hasMoved* variable will ensure that the pawn is only allowed to move one position forward. There are certain circumstance in which the pawn will not be allowed to move forward, such as if it is blocked by another piece on the board. The pawn’s first move will also not be allowed if it would mean a pawn moving past another piece on the board.

The attack condition for the pawn will be handled by the *attack* member function. Upon selection, the pawn class will invoke the *attack* function which will invoke the *gameManager* class in order to ensure that the conditions are met for a pawn to be able to take or capture an opponents piece. These conditions will be satisfied only if an opponents piece if forward of the pawn by one square, *and* left or right one square. Additionally their must not be an active check on the board that is not effectively defended by the pawn’s attack.

The pawn will also have the unique *promotion* member function. Once pawn’s reach the opposing end of the board, they are allotted one immediate promotion to any of the other pieces or ranks. That is, they may become rooks, knights, queen’s or bishops. However, a pawn may never be promoted to the position of King. The *promotion* function will check the pawn’s position after it is moved, and, should the pawn have reached the opposing end of the board, the user will be prompted to choose a promotion for the pawn.

**Rook –** The rook class will include a *move* function that ensure the rook can move directly forward, backward, left or right on the playing surface, but no combination of the above (i.e. the rook shall not move diagonally in any direction). The rook’s *move* function will only allow the piece to move so far as its course is unimpeded by other piece on the board. Additionally, in order for the rook to move, there must be no active check conditions on the board, or the rook’s move must effectively guard against the existing check condition.

The rook’s attack function will allow the rook to move into an occupied square on the board so long as the square is occupied by the opponents piece. Because the rook’s attack is otherwise no different from the normal move, the attack will be handled in the *move* function.

**Bishop** – The bishop class’ *move* function will allow the bishop to move in a diagonal motion. In order to do this, the function will ensure that the forward or backward movement value is equal to the left and right movement value. As with the other pieces, the bishop shall not be allowed to move past or over other pieces on the board. Should there be an active check condition, the bishop will only be allowed to move in such a way that effectively guards against the check condition.

Should the one of the bishop’s possible paths include an opposing piece, the bishop will be able to capture the piece, so long as there is no active check condition, or the taking of the opponents piece effectively guards against the check condition.

**Knight** – The knight’s *move* function will allow the knight to move two spaces in the left, right, up or down directions, and one space orthogonally to the original direction. This meaning that should the knight move two spaces forward, it must also move one space left or right. Unlike other pieces, the knight’s movement is only stymied in the event that there is another piece in the position that the knight is attempting to move to.

In the event that the knight is attempting to move to a position that is occupied by an opponents piece, it will capture the opposing piece in conjunction with its move. Like other pieces, the knight shall not be allowed to move or capture if there is an active check condition unless the move effectively guards against the check condition.

**Queen** – the queen’s *move* function will allow the queen to move in any direction as far as the user requires, so long as her path is not impeded by other pieces. This is to say that she may move forward, backward, left, right, or in any diagonal direction.

The queen may attack an opposing piece by moving to the square in which the opposing piece sits. She will not be allowed to move or attack should there be an active check condition, unless moving or attacking effectively guards against the check condition.

**King** – the king’s *move* function will allow the king to move one square in any direction, be it forward, backward, left, right or diagonally. The king may not move into an occupied square unless it is occupied by an opposing piece.

The king may capture opposing pieces that are directly adjacent to the king, and therefore within it’s movement range. The king will not be allowed to move or attack if there is an active check condition, unless moving or attacking will effectively guard against the check condition.

2.2.2 – Game Class

The *game* class will handle the special conditions of *The Game of Chess*, such as check, checkmate and stalemate. This class will also track all of the piece positions on the board so that the piece classes may invoke the *getOccupiedPositions* member function as they move about the playing surface.

After every move, the *check* class will be invoked to see if there are any active check conditions on the board. The check condition will be made active if a king is one move from being captured but the king is also defended by no more than one move. Should there be an active *check* condition, the game will alert the user by displaying an on screen message that says “Check!”. The piece classes of the next player up will then be restricted to moves that effectively guard against the check condition.

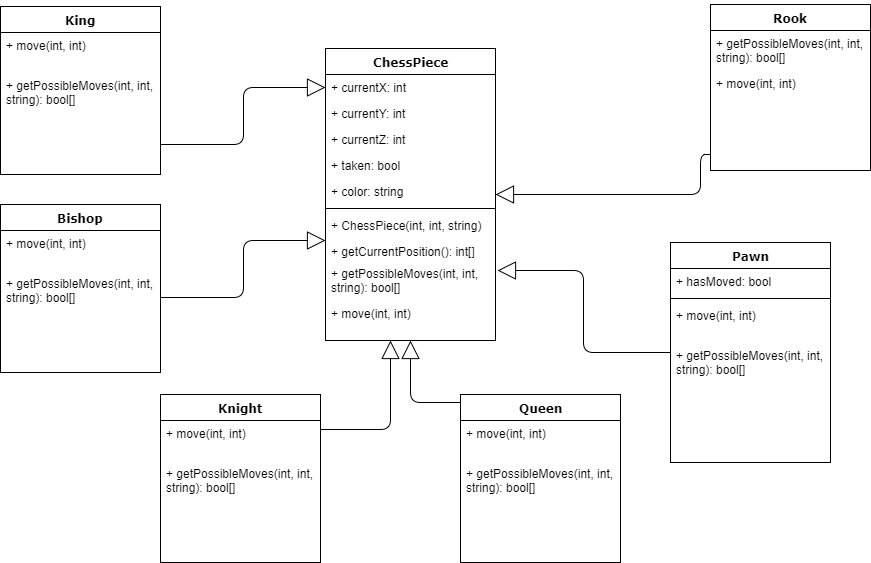
If the check condition is met, it will invoke the *checkmate* function to see if conditions for victory are met. The conditions for victory are met if the next player up cannot guard against the active check condition with a single move.

After every move, should the condition for check or checkmate not be met, the *stalemate* function will be called to check if the stalemate condition is met. The stalemate condition is met if the next player up has no moves that will not result in a check condition, but is not currently in check. This will result in the end of the game under the tie circumstance. No player shall be awarded the victory.

# High-Level Design

<This section describes in further detail elements discussed in the Architecture. Normally this section would be split into separate documents for different areas of the design. High-level designs are most effective if they attempt to model groups of system elements from a number of different views.>

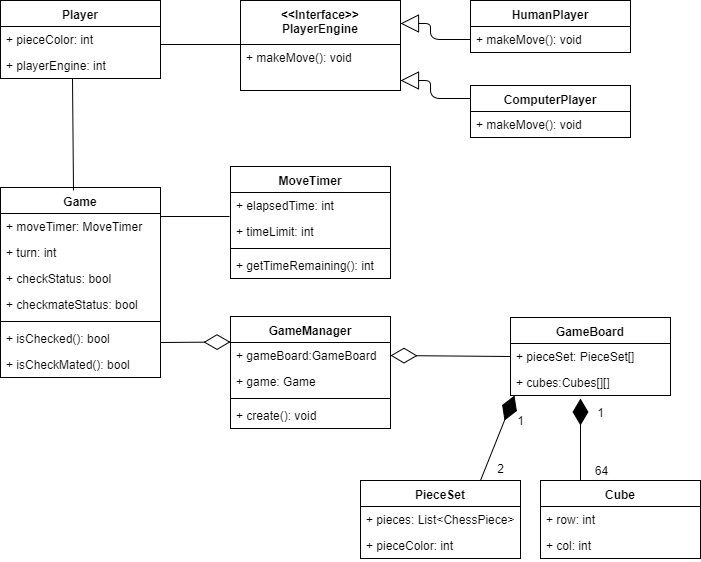
# Chess Pieces

**

# Overview - The game pieces (Rook, Pawn, Queen, Knight, Bishop, and King) will all inherit from a super class called ChessPiece which contains piece location values, a team identifier, and flag to determine if the piece is taken or not. The most important method in ChessPiece is the getPossibleMoves() and move() methods, which will have different implementations for each piece as they each have different movement rules. We will be using a strategy that is similar in nature to the Model-View-Controller pattern to handle the different operations taken by different pieces.

# *Requirements for Design*

* + 1. *Th*ere should be 1 Superclass with 6 other classes inheriting the location values, taken flag, and team identifier.
    2. *Location* will be passed using a two dimensional array, with one dimension representing the x axis, and the second representing the z axis.
    3. There will be a Rook, Pawn, Queen, Bishop, Knight, and King classes.
    4. The game pieces should each implement their own unique version of getPossibleMoves() and move() with respect to their movement rules defined above. {Sect. 2.2}
    5. The Pawn class has a special identifier hasMoved, unique to each pawn, to allow for the Pawn’s special ability of moving 2 spots forward on its first turn, after the first move of the pawn this shall be set to False.
    6. Any More?
  1. **Game & Player Classes**
     1. GameManager is the bridge between Unity UI and Chess Game Code. Explain how.



* + 1. Game Manager is the overarching class which will control the GameBoard containing the game pieces and the Game itself.
    2. The player will have a piece color that is taken as an int. The user will have an opening screen which gives the user the ability to pick a color at the start of the game, afterwards the pieces will be generated in the default player color of White.
    3. Check() and CheckMate() will be run at the start of every turn and at the end of every turn. ~~Or however we see fit.~~
    4. We will use a 2-Dimensional array of “Cubes” in order to handle the positioning of pieces between the Unity UI and the rest of the in-game functions.
    5. MoveTimer will have a constant time allotted for the user to make his/her turn. We will give the user 60 seconds to make a move before a move is made for him/her following the same algorithm used by the Computerplayer. (Another use of Strategy pattern)
    6. Any More?

<Provide a description and diagrams of a system component or set of components that describes a clearly defined view or model of the entire system or a subset of the system.

# Low-Level Design

<This section describes in further detail elements discussed in the Architecture. Normally this section would be split into separate documents for different areas of the design.

## More Information about Design Strat

* 1. More Information about Design Strategies
     1. ChessPiece getPossibleMoves()
     2. ChessPiece move()
  2. More Information about

<Provide a description and diagrams of a system component or set of components that describes a clearly defined view or model of the entire system or a subset of the system.