

SOFTWARE REQUIREMENT AND DOCUMENTATION FOR CHESS

PREPARED BY:

Mohammed Atif Siddiqui

Tristan Gant

Yang Fengming

Chenyu Hao

Alec Mcgylnn

Purpose

This document specifies all the requirements for the chess game. These requirements relates to the functionality, constraints, performance and attributes of the program.

Definitions

Bishop: one of two pieces of the same color that may be moved any number squares diagonally, as long as no other piece blocks its way. One piece always remains on White squares and the other always on Black.

Castling: to move the king two squares horizontally and bring the appropriate rook to the square the king has passed over.

Check: To make a move that puts the opponents King under direct attack.

Checkmate: a situation in which an opponent's king is in check and it cannot avoid being captured. This then brings the game to a victorious result.

Chess Board: A board you need to play Chess. Have 64 black and white square.

Chess: A game played by 2 people on a chessboard with 16 pieces each.

King: The main piece of the game, checkmating this piece is the object of the game. It can move 1 space in any direction.

Knight: This piece can move 1 space vertically and 2 spaces horizontally or 2 spaces vertically and 1 space horizontally. This piece looks like a horse. This piece can also jump over other pieces.

Pawn: One of eight men of one color and of the lowest value usually moved one square at a time vertically and capturing diagonally.

Player or user: A user or a player will be the person that is playing the chess game.

Queen: This piece can move in any number of spaces in any direction as long as no other piece is in its way

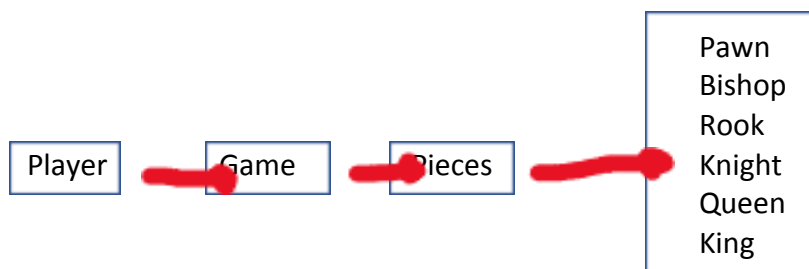
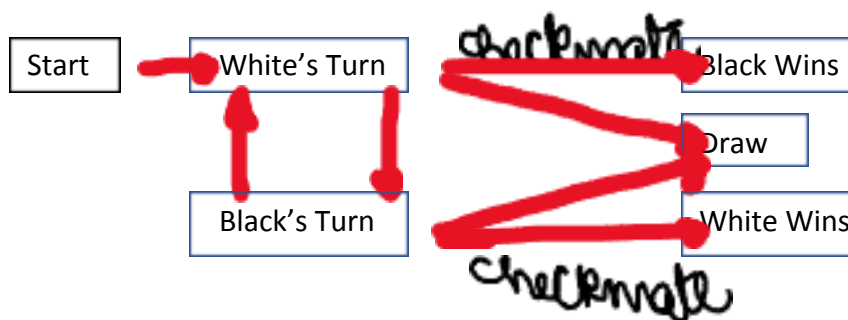
Rook: one of two pieces of the same color that may be moved any number squares horizontally or vertically, as long as no other piece blocks its way.

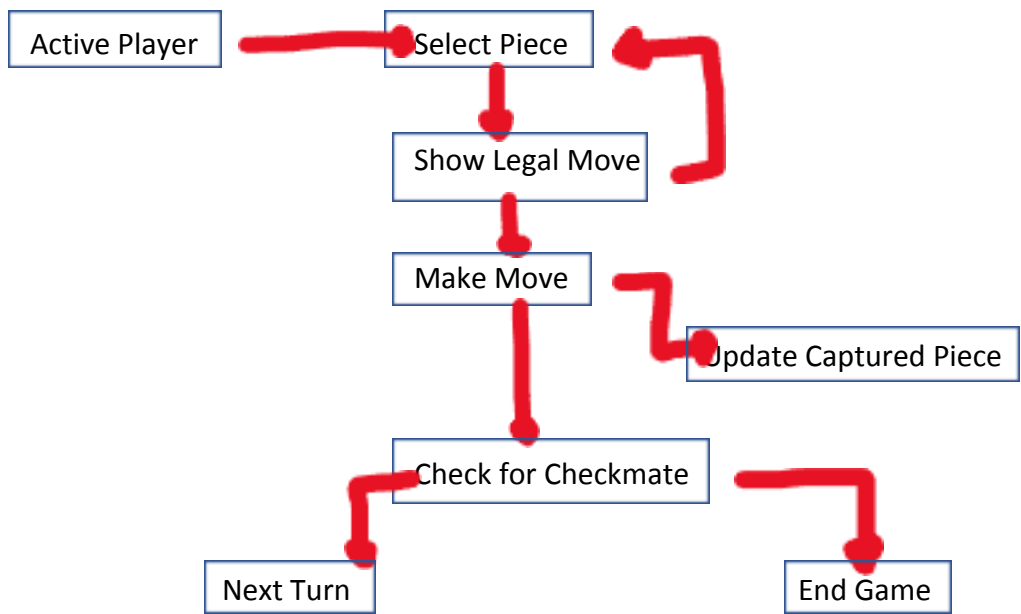
Stalemate: A situation in which a player's king is not in check, but that player can make no move. This then results is a stalemate, which is a draw.

Collection of features

- 8x8 board
- Players choose between Black (LC) or White (UC) chess pieces
- Place the chess pieces on the board
- Movement
- Check if the move is legal
- Check for capture and checkmate on move.
- Quit game

Use Case Diagrams





ESTIMATE FOR COMPLETION

We estimated that the project will take around 20-25 hours. Additional 8-10 hours would go into meeting and documentation.
On average each team member has a estimate time of 5-7 hours

Agile User story

As a user I want
to play a chess
game
ET – 6 Hrs

I should be able
to place my
chess pieces on
the board
ET – 8 hrs

As a user I want
to be able to
move my
pieces
ET- 5Hrs

As a user I
should be able
to attack other
players pieces
ET- 3 Hrs

As a user I want
to play this game
with a friend
ET- 1 Hr

As a user I should
be able to
choose between
LC and UP for my
pieces
ET – 1 Hr

Design Paradigm – Object Oriented

We decided to build the chess game in C++. Object oriented Design Paradigm was our best choice. C++ being an object-oriented language gives us the freedom to bridge the gaps between the phases and smooths out the development process. It retains the best characteristics of the procedural and data driven paradigms while overcoming or minimizing the worst characteristics.

As classes also denote operations or procedures, the transition from design to implementation is much easier. Furthermore, as each class defines a new, intermediate scope (the region of a program where a variable is visible and accessible), the object-oriented paradigm also allows some but not all of the procedures in a program to access the data.

Controlling data access reduces the functional coupling that ultimately sets an upper limit to the size and complexity of software systems that could be practically created based on the procedural paradigm. The many strengths of the object-oriented paradigm make it the current best practice for creating large, complex software systems.

By using the Object Oriented Design Paradigm and C++ it allowed for the use of inheritance. Inheritance is useful in a project like chess because there are multiple kinds of chess pieces that need to be implemented. By creating a chessPieceInterface class it allowed for a 2D array of chessPieceInterface pointers to be declared in higher classes. This allowed for functions of all the chess pieces to be called regardless of what kind of chess piece it is. This is one of the many ways that Object Oriented design helped design the software.

Software Architecture – 3 Tier Architecture

We used 3 Tier Architecture for our Chess project. It is a type of software architecture which is composed of 3 layers of logical computing. Data tier consists of the chess pieces. As these are required to play a game. Application layer consists of the board and executive. It is where most of the coding is done and algorithms are implemented. As we implemented the chess game in C++ the presentation tier is the terminal.

The benefits of using 3 – layer architecture can be seen in the speed, stability, performance and availability. A specific layer can be upgraded with minimal impact on the other layers. It can also help improve development efficiency by allowing teams to

focus on their core competencies. By separating out the different layers you can scale each independently depending on the need at any given time.

For our project it might be hard to call our software architecture a 3-tier project. This is because when most people think of the 3-tier software architecture they think of a data tier being held in something like a mySQL database and they might think of the presentation tier as being a front end making use of JavaScript and CSS. Our project makes use of the 3-tier software architecture in a different way though.

Our data tier is our 2D array of chess pieces. These chess pieces store functionality for each specific piece and it's symbol that represents the piece. Our logic tier is composed of our board class and our game class. The board class initializes the board and the game class lets you play the game. These two classes can be considered the logic tier because they are doing work on our data (calling functions from our chess piece classes depending on runtime input from the users). Our presentation tier is very simple, but it is still important, as it prints out the board after every move so each user can see what is going on as the program is played.

Design Pattern -

The Design Pattern that would fit our project the best is a Creational Design Pattern. More specifically, the Creational Design Pattern being the Factory Method. As mentioned previously in the Software Architecture summary, inheritance was going to play a big part in our project. Inheritance allows for the creation of classes that are derived from base class. The base class being the chess piece interface and the derived classes being the different chess piece classes.

This best fits the Factory Method because the factory method lets the base class call whichever derived class it wants at runtime. Now when Chess starts, each user automatically starts with 16 pre-defined pieces. So choosing this design pattern might seem like it's main functionality is not used. But when the pawn class reaches the other end of the board it has the ability to be promoted to a new chess piece class. The user can pick which chess piece it wants to be promoted to. Now this would be a problem if we didn't make use of inheritance. Instead we can just create a new chess piece interface object and it can create whichever chess piece the user wants (within the rules of the game).

The previous example was one of a few examples where inheritance is used in our code. Another feature of the Factory Method is letting subclasses deal with creating instances of various objects in our game. Our main creates a game object. Our game object then goes on to call a board object. The board object creates a 2D array of chess piece interface pointers to move the pieces on the board. The board class then creates 16

pieces of each type for each team and places them on the board. Using the Creational Design Pattern, Factory Method, was by far the best fitting Design Pattern for our project because it specifically outlines features that fit well with the Object Oriented Design Paradigm.