<u>CS246 A5 Chess Design Document</u>

By: Nicholas Lee, Oditha Amarasinghe, and Gianpaulo Pineda

1.1 Overview

In our project, we have various abstract classes that represent the different key components of Chess. The **Board** class holds the actual board which is a 2-D vector of **Tiles**. It also holds the **Pieces** and **Players** of the game. The **Board** class runs and resets the game. It allows players to take turns, keeps track of the score, and allows players to make moves.

The **Piece** class is an abstract class and holds general information about chess pieces such as the colour ("white" or "black") and the **Tile** that the piece is on. Each **Piece** also has a tracker which helps distinguish pieces and a value which is based on the type of piece. With every **Piece**, there is a map of validMoves which the **Player** can use on their turn.

The **Player** class is an abstract class which has two derived classes called **Human** and **AI**. The **AI** class is also an abstract class, which has one derived class called **Level1**. A chess game must be played with two players, which can be any combination of **Humans** or **AI** (with the **AI** being **Level1**). Each player type has their own characteristics and commands that they can input into the Chess game. For example, **Human** players must manually input their commands, and **Level1** (**AI**) players have their moves randomly made for them.

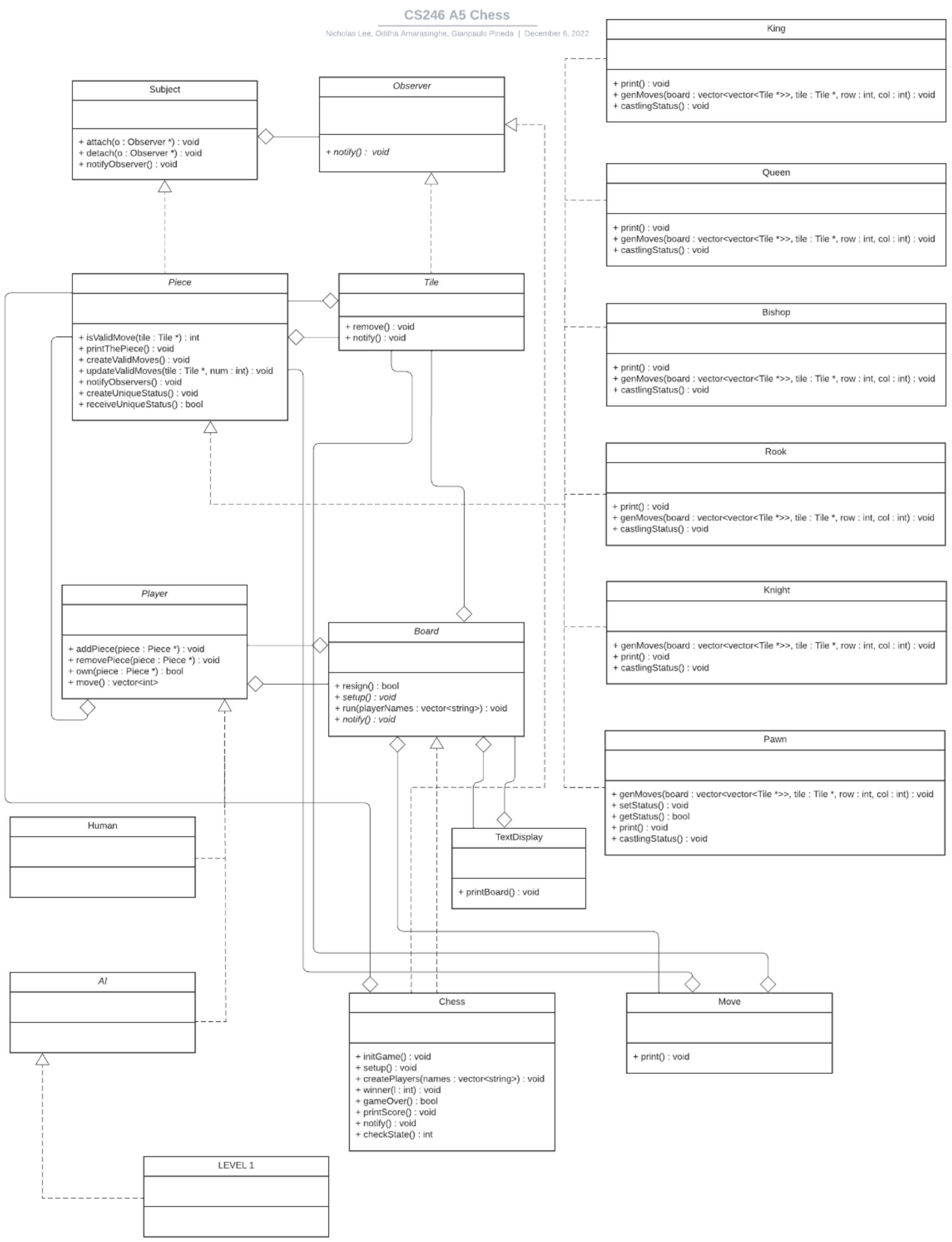
The **Move** class holds various information regarding a move that is made on the board. It knows useful information like the **Tile** where the move began, the **Tile** where it ended, the **Piece** that used to be on the destination **Tile**, and the Piece that currently stands on the destination **Tile**. Additionally, it can print all the information that was previously listed to standard output.

The **TextDisplay** class is what repeatedly prints out the text-based chess board to standard output.

The **Chess** class is a concrete class that is derived from the **Board** class and focuses on the logic of the game of Chess. The functionality that is implemented includes checkmate, check and stalemate. Regarding subsidiary mechanisms, the class supports pawn promotion and castling. The class includes an *initGame()* function which sets up an 8-by-8 board and places pieces in the correct order. It also creates two players, one for each side.

Following the Subject/Observer pattern that will be discussed later, an **Observer** and **Subject** class was implemented

1.2 Updated UML


CS246 A5 Chess
Nicholas Lee, Oditha Amarasinghe, Gianpaulo Pineda | December 6, 2022
Subject
+ attach(o : Observer *) : void
+ detach(o : Observer *) : void
+ notifyObserver() : void
Observer
+ notify() : void
King
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Queen
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Piece
+ isValidMove(tile : Tile *) : int
+ printThePiece() : void
+ createValidMoves() : void
+ updateValidMoves(tile : Tile *, num : int) : void
+ notifyObservers() : void
+ createUniqueStatus() : void
+ receiveUniqueStatus() : bool
Tile
+ remove() : void
+ notify() : void
Bishop
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Rook
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Player
+ addPiece(piece : Piece *) : void
+ removePiece(piece : Piece *) : void
+ own(piece : Piece *) : bool
+ move() : vector<int>
Board
+ resign() : bool
+ setup() : void
+ run(playerNames : vector<string>) : void
+ notify() : void
Knight
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ print() : void
+ castlingStatus() : void
Pawn
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ setStatus() : void
+ getStatus() : bool
+ print() : void
+ castlingStatus() : void
Human
TextDisplay
+ printBoard() : void
AI
Chess
+ initGame() : void
+ setup() : void
+ createPlayers(names : vector<string>) : void
+ winner(l : int) : void
+ gameOver() : bool
+ printScore() : void
+ notify() : void
+ checkState() : int
Move
+ print() : void
LEVEL 1

1.3 Design

Our project utilizes the Model View Controller (MVC) design pattern. In our original UML, our **Board** class was responsible for both the game logic and communicating the results. In order to apply MVC, we separated these responsibilities so now we have a **Chess** class that is responsible for the game logic such as check/checkmate, pawn promotion, castling, and the state of the game. The **Chess** class acts as the Model, the **Board** class acts as the Controller, and the **TextDisplay** class acts as the View. The **Board** class delegates the task of outputting the actual chess board to the **TextDisplay** class

1.4 Resilience to Change (WIP)

In order to accommodate change within our program, we will design our program in a way that heavily focuses on modularization, which is the process of splitting a program into many modules. In this case, we will create separate .h and .cc files for each class, and connect them through the use of the #include command. This is important because when changes need to be made to the program, all we need to do is target and recompile the relevant modules, preventing accidental edits from being made to irrelevant components.

We will also make sure to implement effective module design through designing modules which help us achieve high cohesion and low coupling. With our modules mainly being header and implementation files for our classes, we will have made sure that all relevant code for each class is separated and kept close together. For coupling on the other hand, we unfortunately cannot eliminate it because some classes are derived from others, however we will minimize it by only using the #include command to include header files which are relevant to each module.

One more thing that will make it easy for us to modify our program is an effective use of documentation. Since our program will be coded by multiple individuals, having supporting documentation for each complex piece of code will help increase our overall understanding of each other's contributions. Supporting documentation for this project will be short and informative, communicating functionality in the simplest way possible to the reader.

1.5 Answers to Questions

1. **Question:** Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.
    - Moves would ideally be inputted using the Alphanumeric system (i.e A14).
    - Fundamentally, this implementation would require many conditionals to check if certain **Pieces** have moved to certain **Tiles** and then dictate whether there exists a counter move within a list.
    - In C++, we plan to use the <map> library which would allow us to hold key value pairs of two types. This would allow us to hold a move and a counter move within the same index where calling a move (a key) would return its counter move (the value).
    - All this functionality will be within the (x CLASS) and we would simply call the move function on the counter move.

2. **Question:** How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
    - To give players the ability to undo their last move, we plan to let **Pieces** track their past moves.
    - To begin, we could add a **Move** class which holds the starting **Tile** where the move began, the location of the **Tile** where the move ends, a boolean that tracks if the **Move** captures any piece, and a pointer to the **Piece** that may have been "captured" by the last move.
    - From there, we could make **Pieces** store each **Move** they make in a vector called pastMoves by pushing **Move** objects to the back of the vector whenever a **Move** is made by a piece
    - Undoing a past **Move** would result in the current **Piece** using the information provided from the last element in pastMoves to return the current **Piece** and the deleted **Piece** (if applicable) to its original states.
    - Afterwards, the last **Move** in pastMoves is popped.

3. **Question:** Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.
    - In order for our program to handle a four-handed chess game, we would first need to allow the user to set up this game variant. Then we would have to create an additional two **Players** along with the additional **Tiles** and **Pieces**. The user should also have the option to play in teams of two or individually. In determining when a game is won, new rules need to be defined such as when a team vs an individual wins. The **Board** class would need to be updated to include the new game logic associated with this variant.

1.6 Final Questions

1. **Question:** What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?
    a. There are many lessons that this project taught us about developing software in teams. We learned how to use git and version control practices such as committing with meaningful messages. We learned of the importance of creating an UML and plan document that allowed us to map out the project and create reasonable deadlines for ourselves. We also learned the importance of communication which was vital since many of the components that we worked on intertwined with each other. Working in a team allowed us to rely on each other which was very useful in debugging our code. Overall, developing software in a team was a unique experience that allowed us to gain real world experience.
2. **Question:** What would you have done differently if you had the chance to start over?
    a. If we had the chance to start over, we would have been more careful forming the design document because we made some careless mistakes which could've been avoided. In the planning phase, we would have also tried and brainstormed together how to best implement certain aspects of the game, because most of the implementations were split up and done independently. Having everyone informed as to how things were implemented would've generally made debugging easier on everyone. In the coding phase, we would probably try to be more careful when coding because we witnessed a surprising amount of careless mistakes resulting in errors that ruined our program's functionality. Additionally, we would've made an effort to secure study rooms earlier because we didn't

realize that study rooms are usually booked by students a week ahead of the actual time.