Overview

For different types of subclasses like `Piece` and `Player`, we have created `subType()` methods to identify their types. We have put the rules of each pieces inside the subclasses of `Piece` so that each `Piece` object on the chessboard can return their available normal or capturing moves.

We have implemented `Player` with two subclasses `Human` and `Computer`. The `Human` class reads from human player's command and the `Computer` class automatically plays the next available move according to the level. Each level prioritizes some strategies as specified in the assignment specification. In the level 4 computer, the computer prioritizes on capturing non-pawn pieces.

The `game white-player black-player` instantiates a `Game` class with two players, computer or human players depending on user's choice. The `setup` command allows players to put pieces and checks validity such as existance of king in both sides and the placement of pawns before the game can start. If both players are computers, typing command `done` will make two computers play each other until a draw or stalemate occurs or one is defeated. Otherwise, it is human versus human or human versus computer. In these modes, human player can choose to resign to end the game. When a game ends, score is saved in the `ScoreBoard` and user may change opponent and setup a new game. If the player sends an `EOF` signal, the game prints the final score of both sides.

The AI of computer is based on the undo feature. When computer play a move, it tries every possible move and undo the move, then decides what move will be the next. The computer will never make illegal move or any move that makes it checked, since the rules are implemented inside each subclasses of `Piece` and the computer can always implicitely undo. Human players may make errors such as moving an enemy's piece. In this case, we handle different exceptions and prints error messages accordingly (see the UML for exception classes).

The main design pattern used in the game is the observer pattern. It changes the actual state of the text representation when each cell is changed (the piece placed upon is changed or is removed).

Answers to Questions

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?

The lesson we have learned in team development is to cooperate as closely as possible. Also, it is essential to split the tasks beforehand and design the UML and thus we can avoid unecessary coding. We learned the importance of version controlling using git and BitBucket. An online repository allows us to backup codes, submit pull requests and perform code review inside the team. Version controlling is useful because we can always revert back to the previous version, branching to add new features and prevent accident losses.

What would you have done differently if you had the chance to start over?

If we had the change to start over, we should use shared pointer in our program from the beginning as it reduces chances of getting memory leaks and segmentation faults.

Also, we should be using `const` keyword as often as possible to avoid unexpected changes in the program.

How is it different from Due Date 1?

We have added classes `MoveHistory` and `Tracker` to track the moves made by either human or computer player. This allows human players to undo their moves and computer players to "try" to make a move to forecast the outcome of one move.

We have removed the classes `Setup` and `CommandInterpreter` and instead implemented methods `Game::setup()` and `Game::start()` as classes such as `Human` needs to read commands within itself and it is difficult and unecessary to make a command interpreter that accomodates all classes. In the new UML, the class `Game` "owns-a" class `ChessBoard` and `Tracker`. Along with `Player`, `ScoreBoard`, these classes strengths cohesion as they cooperate to perform one task.

We have implemented the `GraphicsDisplay`. Unlike `TextDisplayCell`, the `GraphicsDisplayCell` class is not a subclass of `Observer` since the computer needs to constantly make and undo moves. We do not want this to be displayed on the graphical user interface and instead we added a `GraphicsDisplay::refresh()` to update the graphics only if it is required.

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.

The solution to this problem is to create a class as follows;

```
class Move {
    std::map<int, std::map<int, std::map<int, std::map<int, Move>>> moves;
public:
    int startRow;
    int startColumn;
    int endRow;
    int endColumn;
    Move & getNextMove(int pStartRow, int pStartCol, int pEndRow, int pEndCol);
};
```

The class `Move`, which acts like a node, is used to store standard moves. In order to get the next Move, use player's move to search for the next move. Once we find that standard move does not exists against player's move (i.e. key does not exist in the map), the computer will use other strategies.

Question: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

We have created a class called `Tracker` that uses a vector to store `MoveHistory` objects, which is used to store necessary information to revert the chessboard to the last state. Each time a player makes a move, a new `MoveHistory` objects is pushed into `Tracker`. We have overloaded the constructor of `MoveHistory` class to accomodate different situations such as castling, promote, capture and promote, normal move, capture move and enpassant. Human and Computer player can call the `Player::undo()` function to undo a move and pop the last `MoveHistory` object out of the stack. If a human player calls `undo` command, `Player::undo()` twice so that the opponent's move is undone. If the stack is empty or has only one `MoveHistory` object, the current player cannot undo but the opponent can undo.

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.

Player can choose to play in two vs two mode or in single one vs three mode. Create a method `Setup::checkMode()` to check the mode option, and `Game::makeAllies(Player *p1, Player *p2)` to combine any of two players as one team randomly. In two vs two mode, we create a method `Piece::isAlliesPiece()` that will check if ally's pieces are in the range of possible moves. If yes, we

consider the pieces as the player's own pieces.

The method `Player::isCheckmated()` will check if king is in the range of opponent's possible moves. It returns true if both opposing kings are checkmated in a two vs two mode. If only one can be checkmated, it is a draw. In a single game, `Player::isCheckmated()` works as two-handed chess game: if a player is checkmated, the checkmated player can either remove their pieces from the board, or the player that checkmated can use the remaining pieces during that player's turn, and the game continues until one player left. The remaining player wins.

Other changes are below:

`ChessBoard` : the size should increase to 8 x 8 + 4 x 3 x 8 = 160 grids by by adding three rows to each side.

`PieceColor` : we should have four `PieceColor` (i.e. Black, White, Green, Blue) instead of two.

`Game` : four players `Game::Game(p1: Player, p2: Player, p3: Player, p4: Player);`

`ScoreBoard` : displays scores of four players.

`Player` : four players.

'TextDisplay': should have initial configuration as below:

```
14  XXXXXXr1n1b1q1k1b1n1r1XXXXXX
13  XXXXXXp1p1p1p1p1p1p1p1XXXXXX
12  XXXXXX  __  __  __  __XXXXXX
11  r2p2  __  __  __  __  __p3r3
10  n2p2__  __  __  __  __  p3n3
9   b2p2  __  __  __  __  __p3b3
8   k2p2__  __  __  __  __  p3q3
7   q2p2  __  __  __  __  __p3k3
6   b2p2__  __  __  __  __  p3b3
5   n2p2  __  __  __  __  __p3n3
4   r2p2__  __  __  __  __  p3r3
3   XXXXXX__  __  __  __  XXXXXX
2   XXXXXXp4p4p4p4p4p4p4p4XXXXXX
1   XXXXXXr4n4b4k4q4b4n4r4XXXXXX
    a b c d e f g h i j k l m n
```