How it different from due1?

We add a class `MoveHistory` and `Tracker` to remember the move history of a piece if it makes an invalid move. If the player makes an invalid move, we will undo that move based on the piece's history. The player can then re-put the piece.

We delete class `Setup` and `CommandInterpreter` and implement public methods `setup( )`and `start( )` inside the class `Game`. The class `Game` "owns-a" class `Player`, `ChessBoard`, `ScoreBoard`, `Tracker` and `Piece` to strengthens cohesion.

The class `GraphicsDisplayCell` is no longer a subclass of `Observer`since we do not want invalid moves to be notified by GraphicsDisplayCell.

Overview of Aspects:

For different types of pieces, we implement 'PieceType' as a subclass of 'Piece', and it will override Piece::getNormalMoves() and Piece::getCaptureMoves() based on their specific types.

We implement 'Player' with two subclasses(i.e 'Human' and 'Computer). After typing in command "game white-player black-player", we enter the Game::setup( ) to let players putting pieces on the chessboard and assigning pieces to the players until player decides to leave setup mode by typing "done". Well check if there is one king for each player, if there are invalid pawns on first and last row, and if any king is in checked. Players stay in setup mode until these conditions are satisfied.

Game::start( ) will then start the game and let players type command(move or resign) alternatively. In case of "move", if the player is a computer, Computer::playNextMove( ) will make piece move based on different level of difficulty. If the player is human. Player needs to type in source and target position. We will check validity of each move based on its specific PieceType. If there is an exception(i.e. invalid move), the class 'MoveHistory' and 'Tracker' will track the invalid moves in order to undo the move action and let the player re-put the piece. We will check if pawn promotion is allowed. If yes, player needs to input specific PieceType to promote, and we set that Pawn to be a new PieceType. After each valid move, we will check if it is in enpassant case or castling case. If it is castling case, then the piece on target position is removed. We will then re-display the chessboard and let the opponent to be current player who will enter the loop of typing commands. Also, before the next player make moves, we will check if there is checkmated or stalemated. If so, the game is end, we call ScoreBoard::incrScore( ) to add 1 point to the winner or 0.5 point to both players in case of stalemate.

In case of “resign”, the opponent wins and game ends. If player types in invalid command, let him/her re-input. When the game is over, we will destruct ‘Game’ by deleting ‘Player’, ‘Pieces’, ‘Tracker’ and ‘ChessBoard’. The ScoreBoard will record players’ scores.

We use observer design pattern. We let the Observer change of states on TextDisplayCell which will record states of each Cell on ChessBoard.

Cohesion: modules pass objects back and forth, and modules affect each other’s control flow.

High Coupling: class ‘Game’ “owns-a” class‘Player’, ‘ChessBoard’, ‘ScoreBoard’, ‘Tracker’ and ‘Piece’. These elements cooperate to make ‘Game’ run.

We use Object-Oriented programming and make each module handle a specific object to strengthen modularity and maintainability. Therefore, if there is any change on rules or features, we can simply revise the class which handle that functionality. For example, if there is change on how game rules, we can change methods in ‘Piece’ and its subclasses.

Question1: 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?

It is important to use git for version control.

Question2: What would you have done differently if you had the chance to start over?

We use normal pointer in our program, and we need to remember its owner in order to free it. We have many objects depending on each other, so it cause memory leak sometimes. In the next time, we will use shared pointers to avoid memory leak. Shared pointer is a stack-allocated object that wraps a pointer. Therefore, we do not need to know its owner. When the shared pointer of an object is destructed, the wrapped pointer will also be deleted.

We do not use “const” in our program, so we will use “const” as more as possible to avoid unexpected changes on programs.

Question Part

Question1: 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.

Question2: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

Create a class called `MoveHistory`. `MoveHistory` "owns-a" `Pieces`, and `Player` "owns-a" `MoveHistory`. Inside class `MoveHistory`, create a stack of map with string keys `"movedPiece"`, `"capturedPiece"`, `"row"` and `"col"`. The captured `Piece` still stores its position when it is removed. The captured piece can be a `Piece *` or `nullptr`. When one wants to undo his last move, we pop out the last item from the stack and reverse the process. If the capture piece is a `nullptr`, there is nothing to put back.

Question3: 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.

Changes are below:

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