Changes

Neither the UML nor the design differed in any major way between due date 1 and 2, as we retained all of the classes, and merely added or subtracted fields and functions from some of them. We mostly added supplemental fields to the classes and appropriately added getter and setter functions to access them. We also added isValidMove(string) and isNAMEMove(string) functions to all the sub-pieces, with most having isBlockedPath(string). These functions are used in the process of verifying if a given move is valid for the individual piece, based on its current location and type. To the King and Rook, we specifically added hasMoved() fields that have getter and setter functions to access and mutate them. They are used for when someone attempts a castle and we want to verify that it is a valid move. We neglected the fact that a piece has a colour field, and added its getter and setter functions. We also cleaned up a bunch of the sub-piece's UML to reduce how many repeated functions were included in their diagram. We updated the move() function in the Computer class to take in nothing and return a Boolean, depending on if there was a valid move or not. We also removed resign(), castle() and movePiece() from the player classes and put them in the game and main classes. In the textView class we added a number of functions that are used for printing the board, updating the board, and determining if a given tile should have a dash or space printed, returning what char a given position has printed, and printing the final score. In the game class we removed updateBoard(), added castle() removed the print functions and put them in the view, added calcPosition(int, int) that takes in a x and y coordinate as ints and returns a string that is the board's position, added a second setup() function for loading games. There are a couple other utility functions we added to Game such as getPieceAt(), and removePiece(). Their purpose is to ease managing the board adding pieces during setup, and removing captured pieces. The purposes of these changes are mostly to make sure the functions are within the most relevant lasses and hat they are all related

Those are the most material changes to our UML and the functions contained in them.

Design patterns, all the aspects of the project and choices you made.

The design choices we made included having all the pieces most of their behaviours and traits inherit from an abstract class Piece, having a Game that contains a 2D-array of tiles, a view that displays text and having an abstract player class that the human and computer inherit many of their behaviours and traits from. We made these choices to optimize code reusability, and to keep the design simple and intuitive. By having a 2D array of tiles, it was very simple to write the code to access tiles, and even easier to read and understand each other's code that we had written.

Talk about each class and how it interacts with others

Piece: A pure virtual class that all the different individual pieces inherit from.

Methods:

+ Piece() : constructor
+ move(string): void :moves a piece to string location that was taken in
+ getX(): int : gets value of field x
+ getY(): int : gets value of field y
+ getName(): string : gets value of field name

+ getPos(): string : gets value of field position
+ getColour(): char : gets value of field colour
+ setPos(string): void : sets value of field position
+ setColour(): void : sets value of field position
+ setName(): void : sets value of field name
+ setY(): void : sets value of field y
+ setX(): void : sets value of field x
+ getGame(): Game * : gets the game the piece is attached to
+ setGame(): void : sets the game the piece is attached to
+ intPosToStr(): string : takes in two ints (x and y coordinates) returns as a string
+ isValidGame(): bool : returns whether the game is valid

Queen: Queen piece on board can move in any direction, as many spaces as it wants. Inherits from Piece

Methods:

+ Queen () : constructor
+ move(string pos): void : moves piece to location pos
+ isBlockedPath(string pos): bool: takes in a string, and returns whether the piece's path to it is blocked by other pieces
+ isQueenMove(string pos): bool : takes in a string and returns whether it's a valid pos
+ isValidMove(string pos): bool : returns whether the move is a valid one for the piece

King: King piece on board can move in any direction, one space. Capture it to win. Inherits from Piece

+ King () : constructor
+ move(string pos): void : moves piece to location pos
+ setHasMoved(): void : after the king's first move, we set its move param to true using this
+ getHasMoved(): bool : return whether the king has moved before
+ isKingMove(string pos): bool : takes in a string and returns whether it's a valid pos
+ isValidMove(string pos): bool : returns whether the move is a valid one for the piece

Pawn: Pawn piece on board can move in any direction, as many spaces as it wants. Inherits from Piece

+ Pawn () : constructor
+ move(string pos): void : moves piece to location pos
+ isPawnMove(string pos): bool : takes in two ints (x and y coordinates) returns as a string
+ isValidMove(string pos): bool : returns whether the game is valid

Bishop: Bishop piece on board can move diagonally, as many spaces as it wants. Inherits from Piece

+ Bishop () : constructor
+ move(string pos): void : moves piece to location pos
+ isBlockedPath(string pos): bool: takes in a string, and returns whether the piece's path to it is blocked by other pieces

+ isBishopMove(string pos): bool : takes in a string and returns whether it's a valid pos
+ isValidMove(string pos): bool : returns whether the move is a valid one for the piece

Rook: Rook piece on board can move forward/backward or left/right, as many spaces as it wants. Inherits from Piece

+ Rook () : constructor
+ move(string pos): void : moves piece to location pos
+ isBlockedPath(string pos): bool: takes in a string, and returns whether the piece's path to it is blocked by other pieces
+ setHasMoved(): void : after the king's first move, we set its move param to true using this
+ getHasMoved(): bool : return whether the king has moved before
+ isRookMove(string pos): bool : takes in a string and returns whether it's a valid pos
+ isValidMove(string pos): bool : returns whether the move is a valid one for the piece

Knight: Rook piece on board can displace itself 3 units in any combo of left/right/up/down and jump over pieces. Inherits from Piece

+ Knight () : constructor
+ move(string pos): void : moves piece to location pos
+ isBlockedPath(string pos): bool: takes in a string, and returns whether the piece's path to it is blocked by other pieces
+ isKnightMove(string pos): bool : takes in a string and returns whether it's a valid pos
+ isValidMove(string pos): bool : returns whether the move is a valid one for the piece

Game: The actual game class, has a 2D array of tiles, pointers to two players, a pointer to a view, pointers to two vectors of the player's pieces, and two ints that hold the number of pieces each player has.

+ Game(): void : constructor

+ ~Game(): void : destructor

+ setup(string,string): void :Takes in what type of players (human, computer) and sets up game for that.

+ setup(char Array,bool): void : sets up a saved game

+ addPiece(char,string): void : adds a piece of the specified type on the position specfied

+ removePiece(string): void : removes the piece at the specified location

+ calcPosition(int,int): string : takes in two ints and returns a string position

+ move(string, string,char): void : moves piece from first string to second

+ castle(string, King, Rook): void : code for allowing a castle move to happen

+ getPlayer(int); Player * : returns the player with the number inputted

+ isCheck(string): bool : checks if the board is in Check and returns a boolean

+ isCheckmate(): bool : checks if a board in check is also in checkmate and returns a boolean

+ isStalemate(): bool : checks if a board is in stalemate, ie. Any Player has no legal moves

+ getPieceAt(string) : char : gets the piece at the tile in the position specified

+ upgrade(Piece,char): void : Piece that gets to the end is upgraded to the type specified

+ updateBoard(string,string,char): void : wrapper for the view's updateboard function, calling the view attached to the game

+ isValidPosition(string): bool : checks whether the inputted position was valid (on board)

+ getTileAt(string): Tile * : gets the tile at the position specified

+ setPosition(Piece, string): void : Takes the piece specified and sets its position to that that was specified

Main: This function has the logic that determines when we should use each function, and runs the program.

View: Pure virtual class that the graphicView and textView inherit from.

+ View() : constructor

+ ~View() : destructor

+ printBoard(): void : prints the board out

+ updateBoard(string,string,char): void : changes and reprints the board

textView:

+ textView() :constructor

+ ~textView() :destructor

+ printBoard(): void : prints the board using standard output

+ printFinalScore(): void : prints the final score using standard output

+ setGame(Game *g): void : Takes in a game and sets it as the textView's game field

+ setPos(string,char): void : Takes in a

+ dashOrSpace(int,int): char : takes in 2 ints determine's if a given coordinate will have a dash or blank space

+ dashOrSpace(string):char : takes a string pos and determine's if a given coordinate will have a dash or blank space

+ getCharAt(string): char : returns a blank or dash at a given coordinate

+ updateBoard() : void : Changes and re-prints board

graphicsView: Empty class that was meant for containing an X11 graphics interface for the board.

Player: A pure virtual class that Computer and Human inherit from. It represents the players of the game that can a human player or a computer play.

+Player(int) : constructor

+~Player : destructor

+ movePiece(Piece, string): void : Takes in a piece and moves it to the pos specified in the input string

+ setPlayerNum(int): void : Sets number of players

+ getPlayerNum(): int : gets number of players

+ setGame(const &Game): void : sets the inputted game as the game associated with the players

Computer: Computer player that decides on its moves on its own. How complex and intelligent its moves are, is determined by what difficulty it is created with. It can be from 1-4. Inherits from player

+ movePiece(): bool : Makes a move, returns whether it had a valid move and worked or not

+ Computer(int, int) : Constructor, takes in level and its player number

Human: A player that submits its moves from standard input and has no assistance from the program. Inherits from player

+ Human(int) : Constructor, takes in player number

1. **What lessons did this project teach you about developing software in teams?**

   Developing software in teams is very different than developing software individually as you must communicate what you may have picked up subconsciously, always be in contact with your team, clearly comment your code and have expectations and responsibilities clearly delineated. Communication takes a far larger importance, than when independently developing software, as being able to clearly explain where or why an error is occurring can increase the speed of correcting it significantly and can allow you to produce more than the sum of your individual outputs.

2. **What would you have done differently if you had the chance to start over?**
   If we were to start over, we would include had a more rigorous timeline, using unit testing from the beginning and started with the game including functions that would allow the game to change its state. This would allow us to create more deliverable and deadline accountability, compile and test

each class earlier – improving the likelihood they all work together by the deadline, and implement en passant and build a computer player with significantly more ease. This would save countless hours, reduce the amount of communication between the partners in the group and reduce stress and sleep deprivation.

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.

Answer: You could create a class called *moveSet* that contains data for a specific move. If you only wanted the book of standard openings to display what to do, it could simply have a function that outputs something like "move pawn forward 2" or "move bishop forward left". If you wanted to actually have *moveSet* perform an action on the board, it could hold a pointer to a piece (ex pointer to queen) and the information for the move (move number, current position, opponent's move), and it could call piece->move.

This could be held in a 2d array. So the first row would be for the first move, and would hold different options (ex arr[0][0] would be move left pawn up 2 spaces, arr[0][1] could be move knight forward left, etc). Second row would hold the 2nd play, etc

Assuming the player using the book of moves knew which each did and when to use them, it could be held in a tree to lookup whatever move they typed in (ex "firstMovePawnE2").

There is no difference between how we answer the question now and before.

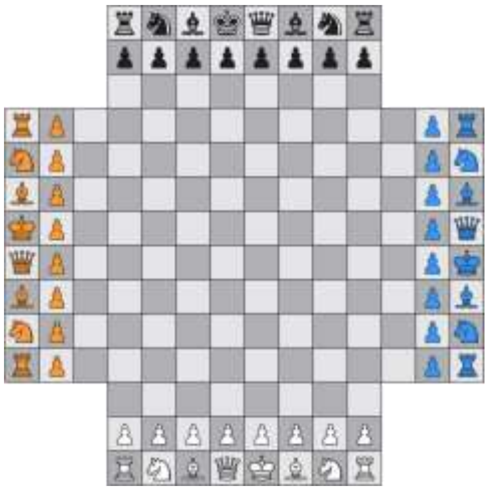
Question: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

Answer: You could use a stack for the player's move. If it's only undoing their last move, it could be called before their turn ends. Each piece could have a vector of its move history, and the board can have a vector of pieces moved. So every time a player moves a piece, that piece is added to the board's lastMoved stack and where the piece moved is stored in the piece's lastMove stack. To undo a move (or unlimited), you could simply say "undo" and the board would run a function undo() to undo the last move, essentially popping the piece off of the stack of last moved pieces.

There is no difference between how we answer the question. Only thing we would add is we should probably have implemented this to do some features quicker.

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.

Answer: First, you would need to create 2 more players. The board size would have to be increased, you would have an extra 3 rows or columns to each side respectively. Example below. There could be an option for the opposite sides to form teams to create 2 teams of 2, or an option for a free-for-all. You would also have to make sure turns taken are rotated and not back and forth. Pawns could reach the left and right ends as well as the far end to change pieces. The game would only end when one player (or team) is left.



There are again, no changes between our answers.

Question: Assuming you have already implemented all the commands specified in the Command Interpreter. Is there a way to implement the "Load a pre-saved game" feature, with maximal code reuse. Explain.

Answer: You could create a function for new games called "setup". As specified in "Command line options", loading a pre-saved came consists of 9 lines, 8 that contain the board's information. This could be loaded in a data structure/vector/array. To load the game or start a new game, a function called *setup* would be called. *setup* will have default parameters *in the function(ex. setup(x=0,y=1,z=2))* that are specifications for a new game. If you wanted to load a new game, you can just call setup with the default values. If you wanted to load a pre-saved game, you can enter in the data to override the default parameters to create a new game with your specifications.

This is how we implemented our setup() function and it worked well.