1. Introduction

In this assignment, the task was to implement a game of chess in C++ employing strong object-oriented design techniques. Chess is a board game played on an 8-by-8 checkerboard. There are two sets of pieces, one white and one black, which are owned by two opposing players. The player on the white side moves first. Each piece can capture the opposing side's piece by moving onto the same square occupied by the piece, subsequently permanently removing it from the game. A piece that can capture another is said to be attacking it. There are six distinct pieces: pawn, bishop, knight, rook, queen, and king, and all pieces have distinct movements and methods of capturing. The goal of the game is to protect one's piece while attacking the opponent. If either king is being attacked, it is under check, and if the king has no routes to escape, it is checkmate and the game is over.

2. Overview

The overall project is built off of multiple abstract classes, each implemented to serve as an interface for the multiple moving parts of chess. The primary class responsible for managing and running the game is **Board**, which is an abstract class that serves to encapsulate all functions. **Board** is an object that represents an extremely high-level board game. It oversees general game flow such as starting the game and running it, making moves and taking turns between all players, and ending the game once a game over is raised. Once a game ends, the board will reset all its parameters to be ready for the next game. Every board has cells that make up the literal board, pieces that are used to play the game, players who manipulate the pieces, and moves that are taken throughout the course of the game. The **TextDisplay** class has a board and prints out the current state of the board game whenever the game is updated.

The class **Chess** is a concrete implementation derived from **Board**, and therefore focuses on chess-specific game mechanics. This includes checking for checkmate, check, and stalemate, setting up the board to an 8-by-8 grid and placing pieces in the correct starting order, and creating two players for the white and black sides. It extends **Board**'s features, as it is a board game while adding the distinct features of chess.

A **Cell** refers to the specific slot of a game board, that is it only holds enough space for exactly one piece to be on it. All cells have a relative position on the board, which for simplicity is notated as a row (x) and column (y). Cells may have a piece on top of them, or they will be empty. Cells can also be "threatened" by pieces, in which a piece can "attack" or move to the cell in the resulting turn.

The **Piece** class is an abstract interface that incorporates general functionality for all board game pieces, and each specific piece will be subclasses of this object. Pieces are owned by the board game and have a player which can control and move them. Pieces also must sit on top of a cell on the board. They may have different colors or sides, different names, or different ids to differentiate them from other pieces. Pieces also have a set of valid moves to which they can move in a turn. All chess pieces (pawn, bishop, knight, rook, queen, king) can be implemented by extending this class.

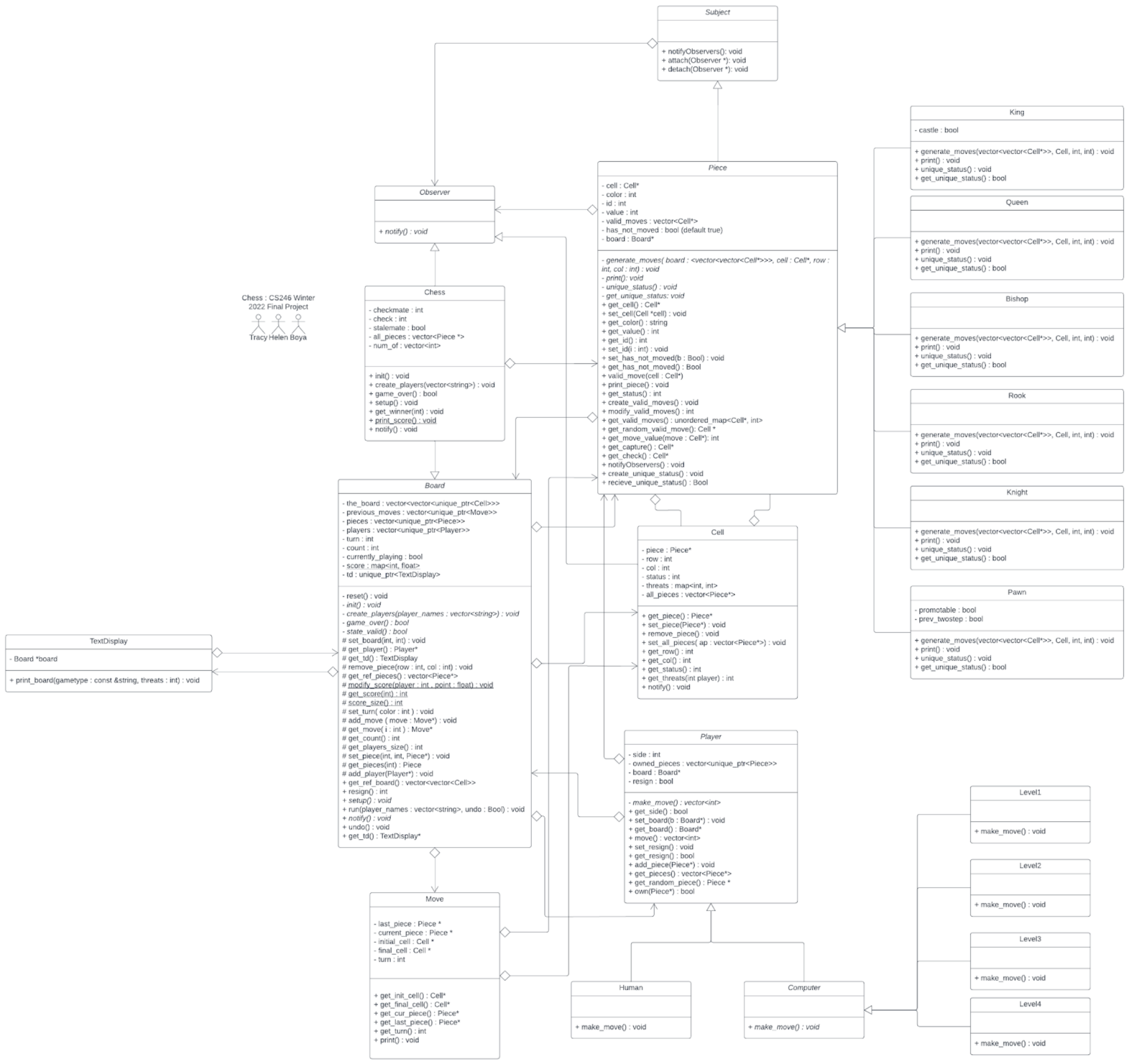
Player represents any player of the game, be it a human or a computer, and a board game can have an arbitrary number of them. Any **Player** has a set of pieces that are theirs and only they may move. Players also have a “side” that dictates which turn of the game is their opportunity to move. Players must create an order and pass it to their pieces, and thus the overall board, for each round throughout the game. Both human players and computer players inherit from this class.

The **Move** object is utilized for bookkeeping and recording previously performed moves. It thus knows the cells upon which a move was made, the piece or pieces whose positions were changed during the move, and the round which this move was made.

Additional classes include **TextDisplay**, which is used to print out the board game to standard output, as well as **Observer** and **Subject**, both of which are abstract interfaces to implement the observer pattern in the design.

3. UML

Below is the UML for this project:


Subject
+ notifyObservers(): void
+ attach(Observer *): void
+ detach(Observer *): void
Observer
+ notify() : void
Chess : CS246 Winter 2022 Final Project
Tracy Helen Boya
Chess
- checkmate : int
- check : int
- stalemate : bool
- all_pieces : vector<Piece *>
- num_of : vector<int>
+ init() : void
+ create_players(vector<string>) : void
+ game_over() : bool
+ setup() : void
+ get_winner(int) : void
+ print_score() : void
+ notify() : void
Piece
- cell : Cell*
- color : int
- id : int
- value : int
- valid_moves : vector<Cell*>
- has_not_moved : bool (default true)
- board : Board*
- generate_moves(board : <vector<vector<Cell*>>>, cell : Cell*, row : int, col : int) : void
- print(): void
- unique_status() : void
- get_unique_status: void
+ get_cell() : Cell*
+ set_cell(Cell *cell) : void
+ get_color() : string
+ get_value() : int
+ get_id() : int
+ set_id(i : int) : void
+ set_has_not_moved(b : Bool) : void
+ get_has_not_moved() : Bool
+ valid_move(cell : Cell*)
+ print_piece() : void
+ get_status() : int
+ create_valid_moves() : void
+ modify_valid_moves() : int
+ get_valid_moves() : unordered_map<Cell*, int>
+ get_random_valid_move(): Cell *
+ get_move_value(move : Cell*): int
+ get_capture() : Cell*
+ get_check() : Cell*
+ notifyObservers() : void
+ create_unique_status() : void
+ recieve_unique_status() : Bool
King
- castle : bool
+ generate_moves(vector<vector<Cell*>>, Cell, int, int) : void
+ print() : void
+ unique_status() : void
+ get_unique_status() : bool
Queen
+ generate_moves(vector<vector<Cell*>>, Cell, int, int) : void
+ print() : void
+ unique_status() : void
+ get_unique_status() : bool
Bishop
+ generate_moves(vector<vector<Cell*>>, Cell, int, int) : void
+ print() : void
+ unique_status() : void
+ get_unique_status() : bool
Rook
+ generate_moves(vector<vector<Cell*>>, Cell, int, int) : void
+ print() : void
+ unique_status() : void
+ get_unique_status() : bool
Knight
+ generate_moves(vector<vector<Cell*>>, Cell, int, int) : void
+ print() : void
+ unique_status() : void
+ get_unique_status() : bool
Pawn
- promotable : bool
- prev_twostep : bool
+ generate_moves(vector<vector<Cell*>>, Cell, int, int) : void
+ print() : void
+ unique_status() : void
+ get_unique_status() : bool
Board
- the_board : vector<vector<unique_ptr<Cell>>>
- previous_moves : vector<unique_ptr<Move>>
- pieces : vector<unique_ptr<Piece>>
- players : vector<unique_ptr<Player>>
- turn : int
- count : int
- currently_playing : bool
- score : map<int, float>
- td : unique_ptr<TextDisplay>
- reset() : void
- init() : void
- create_players(player_names : vector<string>) : void
- game_over() : bool
- state_valid() : bool
# set_board(int, int) : void
# get_player() : Player*
# get_td() : TextDisplay
# remove_piece(row : int, col : int) : void
# get_ref_pieces() : vector<Piece*>
# modify_score(player : int , point : float) : void
# get_score(int) : int
# score_size() : int
# set_turn(color : int) : void
# add_move (move : Move*) : void
# get_move(i : int) : Move*
# get_count() : int
# get_players_size() : int
# set_piece(int, int, Piece*) : void
# get_pieces(int) : Piece
# add_player(Player*) : void
+ get_ref_board() : vector<vector<Cell>>
+ resign() : int
+ setup() : void
+ run(player_names : vector<string>, undo : Bool) : void
+ notify() : void
+ undo() : void
+ get_td() : TextDisplay*
TextDisplay
- Board *board
+ print_board(gametype : const &string, threats : int) : void
Cell
- piece : Piece*
- row : int
- col : int
- status : int
- threats : map<int, int>
- all_pieces : vector<Piece*>
+ get_piece() : Piece*
+ set_piece(Piece*) : void
+ remove_piece() : void
+ set_all_pieces(ap : vector<Piece*>) : void
+ get_row() : int
+ get_col() : int
+ get_status() : int
+ get_threats(int player) : int
+ notify() : void
Player
- side : int
- owned_pieces : vector<unique_ptr<Piece>>
- board : Board*
- resign : bool
- make_move() : vector<int>
+ get_side() : bool
+ set_board(b : Board*) : void
+ get_board() : Board*
+ move() : vector<int>
+ set_resign() : void
+ get_resign() : bool
+ add_piece(Piece*) : void
+ get_pieces() : vector<Piece*>
+ get_random_piece() : Piece *
+ own(Piece*) : bool
Move
- last_piece : Piece *
- current_piece : Piece *
- initial_cell : Cell *
- final_cell : Cell *
- turn : int
+ get_init_cell() : Cell*
+ get_final_cell() : Cell*
+ get_cur_piece() : Piece*
+ get_last_piece() : Piece*
+ get_turn() : int
+ print() : void
Human
+ make_move() : void
Computer
+ make_move() : void
Level1
+ make_move() : void
Level2
+ make_move() : void
Level3
+ make_move() : void
Level4
+ make_move() : void

Overall, the final UML varies greatly from the original draft. This is primarily due to the addition of further parameters to keep track of the state of the board, as well as getters and setters to these private parameters. Many excess functions needed to be added in order to preserve encapsulation and follow the NVI idiom. The core concepts of our design remained the same throughout, primarily the game logic and the functionalities of each individual class.

An additional **TextDisplay** class was added upon learning and attempting to implement the single responsibility principle, allowing a class to manage the printing and display functionalities of **Board** as opposed to adding extra functions in the already hefty class, and possibly creating more coupling by assigning it an overloaded friend operator<<. This also allowed flexibility in design as an additional feature of displaying the threats can be added. Another addition is the **Observer** and **Subject** abstract classes. Though the idea of using the observer pattern was always planned, additional abstract classes were required to fully implement the concept into the code.

4. Design

The overall running and playing of the game is handled within the **Board** class. Since chess is turn based, and similar to other board games, follows the general gameplay of players taking turns making moves until the win condition is reached, the abstract **Board** class aids to incorporate these general features which may be reused for other board games outside of chess. **Board** has unique pointers to **Cell**, which make up the game board, **Move**, which signifies previous moves made in the game, **Piece**, which are all pieces in the game, and **Player**, which are the players currently playing the game. This is due to **Board** having ownership and overseeing these objects over the course of its existence.

In order to keep track of the rounds and which player's turn it is, **Board** has an int *turn* and *count*, the former of which tracks whose turn it is and the latter tracks the round number in the overall game. For ease of design, players are numbered starting with 0, and increasing for each additional player. Then *turn* can match the player's number to signal that it is their turn to move. Since **Board** must be aware of when to stop running, there is a boolean value called *currently_playing* which is true if the game is in session, and false otherwise. Upon reaching a game over, *currently_playing* will be set to false, and will be set to true once a new game is set up. Since a requirement was to keep track of the number of times a side, white or black, has won, **Board** has a parameter which is a map of an int to a float. The in represents the player, in accordance to their player number, and the float is the score, as a 0.5 score can be allotted.

The function *run()* is the main function which runs the game. It very generally follows the steps of starting and initializing the game, taking in moves from each player, and checking if a winning condition is ever met. The run function calls the virtual functions *init()*, *create_players()*, and *game_over()*. This is in accordance with the non-virtual interface idiom, which states all virtual members should be kept private.

These functions are the backbone of *run*:

- *init()*
  - Function which initializes the very start of the game
  - For chess
    - Create an 8-by-8 checkerboard

        - Generate exactly 8 pawns, 2 knights, 2 bishops, 2 rooks, 1 queen, and 1 king for each of the white and black sides.
- *create_players()*
    - Creates players depending on whether the program was passed "human" or "computer"
    - Fulfills the purpose as a factory method to create the desired players for the instance of the game.
- *game_over()*
    - Checks whether or not the game is over and if a player has won the game
    - In chess, this would check for checkmate, stalemate, or a player's resignation

To keep track of resignation, a helper function *resign()* is used to iterate over all players in the game and check if any of them resigned; a player has resigned, their corresponding number is returned and the game knows they lost. Observe that the above three functions will vary depending on the game, hence they are virtual and are concretely implemented in the **Chess** class.

In order to track scores, a static map is used as it will exist throughout all constructions of a **Board** object. A map is chosen because the individual players correlate to the keys, and their corresponding score is the value, which makes a key value pair that a map can easily store. The parameter is static so that it will track all board games that were created and return the scores of all white players and all black players. It is simple to clear the scoreboard if needed by calling *std::map::clear()* and **Board**'s *get_score()* function can easily return the desired score given a player's number by searching for the "key" (player) and returning the "value" (score). A static function is also provided to manipulate this scoreboard.

Once main.cc reads a EOF signal from input, it calls chess_game's *print_score()* function. Note that the requirements outline printing "white" and "black" for the two players, a trait unique to chess, thus this function is handled in **Chess**. Similarly, concrete implementations of *init*, *create_players*, and *game_over* are written in **Chess**, as outlined above. This follows the rules of chess by checking for checks, checkmates, and stalemates every move.

In order to track the state of the board, **Chess** observes all of its pieces, namely the king. If **Chess** discovers that the cell a king is situated on is under threat, by simply getting the cell of the king then calling *get_threats(opposing color)*. This causes **Chess** to change the value of *check* to the number of the player in check. If the player has no moves to get out of check, **Chess** declares the side to be in checkmate and updates *checkmate* accordingly. In *game_over()*, if *checkmate* is ever not -1, the function returns true and the game is over, as handled in **Board**.

Chess must also check for chess-unique board situations, such as castling or pawn promotion. Once again it must observe its pieces and if the conditions are correct, it must allow these moves to happen. For castling, the king and rook must have *has_not_moved* set to true, the cells in between must be empty, and they cannot be threatened by the opposing player. If these rules are followed, the king moves two squares and the rook moves to the other side of the king. **Chess** observes the king moving two cells, then updates the rook as needed and creates a corresponding **Move** object for the rook, adding it to previous moves with the turn the move was taken being the same turn as the king. This allows bookkeeping of castling and the undoing of the move. For pawn promotion, if either pawn makes it to the back row, **Chess** will prompt standard input to provide a piece that the pawn should promote to. It then will remove the pawn from the game and add a new updated piece. A move is also created in this instance, with *last_piece* being the pawn and *current_piece* being the updated piece. Since these situations may arise at any moment throughout the game, **Chess** is thus frequently notified every turn when a piece moves.

In both these situations, king and pawn must be in a very specific configuration to allow these moves to occur. For that reason, all pieces have a *create_unique_status()* where a completely distinct and unique status of the piece can be updated. This calls the virtual method *unique_status*, which is overloaded by each piece. For the king, it is the *castle* boolean value. For pawn, it is the *promoted* boolean values. For other pieces that do not have this situation, by default their *unique_status* is false. Note this status is required to prevent repeated situations, such as **Chess** trying to promote a pawn multiple times.

The cells in chess can either have a chess piece upon it or it will be empty. To replicate this behavior, a **Cell** has a raw pointer to a **Piece**, or nullptr if it is empty. Note that cells may only have at most one piece on it at any given time. This imitates capturing in chess. Observe that cells do not necessarily own the pieces and thus a reference is sufficient. Since **Board** has unique pointers to all pieces, the program can use as many raw pointers as needed to serve as references and pass parameters. **Cell** will have a status which signals whether or not it holds a piece. 0 means it is empty (ie. pointing to nullptr), 1 implies it contains a white piece, and 2 implies it contains a black piece. This is useful for implementing capturing and getting valid moves of pieces, as pieces are completely blocked off by those of their own colors, but can capture those of opposing colors and hence can move to that **Cell**.

Cells must also know which pieces are attacking or “threatening” it. To do so, **Cell** has a mapping of ints, representing the color, to another int, which represents how many of that color is threatening the cell. This must be frequently updated every turn, as the attacks will constantly change throughout the course of the game. In order to implement this, **Cell** observes **Piece** using the observer pattern, more specifically every single cell in the game is a concrete observer and must observe every single piece, which is a concrete subject. Then once a piece moves and changes position, it notifies its observers, and every **Cell** then runs its own *notify()* and updates its map<int, int> threats. It does so by checking every piece and seeing if itself is a valid move to which the piece can advance in the next turn. At higher levels, the computer AI must avoid cells that are under attack, and this provides an efficient way for the algorithm to prioritize cells with low threat rates.

It also allows pieces to easily check if they are under attack by checking the threats of the cell beneath them. Once again, this can be used in a computer play style where they move pieces that are being attacked to safety. More importantly, this allows **Chess** to check if either king is in check, that is being threatened. Simply if the cell the king is on is being threatened by the opposing colour, then the king and its corresponding side are in check. If additionally, the king's vector of valid moves is empty, then by definition the king is in checkmate and the game is over.

This means that though **Cell** points at **Piece**, **Piece** must also know and point to the **Cell** it is on. This is also necessary for tracking valid moves a piece can make, as it is relative to its current position. Pieces also have a *colour*, an int represented which side it belongs to, white or black. Players cannot move pieces whose colour does not match their side. Also, a **Piece** cannot move past a piece of the same colour. However, pieces can capture pieces of opposing colors. Pieces also have a *value* depending on what type of piece it is.

In chess:

- King: 10
- Queen: 9
- Rook: 5
- Bishop: 4
- Knight: 3
- Pawn: 1

The value is initialized during the construction of a concrete **Piece**. It allows for an easy way of checking the type of piece as well as associating pieces with their importance. In high-level chess, a computer may prioritize capturing and protecting pieces with higher value, such as a queen, over lower valued pieces, such as a knight. However, since multiple pieces can have the same type, a unique int id is required to differentiate each individual piece. This is needed when tracking moves in a game, as a specific piece, say a pawn, is moving or being captured and should not be confused with other pawns. It is also important to know whether or not a piece has moved, as in some board games this allows for different behaviours. For example in chess, pawns may move up two cells if they have yet to move. Also, castling is only allowed if both the king and rook have not yet moved. Thus *has_not_moved* is required to track this behaviour.

As mentioned above, pieces have a vector *valid_moves* of cells it can move to. As these values can change during any move, either because the piece itself moved, or a piece that was blocking its path was moved. Hence, at every turn during **Board**'s *run()*, all pieces must call their *create_valid_moves()* to generate possible cells they can move to. Note this includes capturing moves. Since each piece has a distinct way of movement, each subclass of **Piece** overrides its private virtual *generate_moves()* function, which is called in the public *create_valid_moves()*. Once again this follows the NVI idiom. Notice that in order to get and traverse possible valid cells, **Piece** must hold a reference to the board.

Each piece will move and capture differently

- King: Move one square in any direction
- Queen: Moves in any of the 8 possible directions
- Bishop: Moves in any of the 4 diagonal directions
- Rook: Moves in any of the 4 horizontal or vertical directions
- Knight: Moves in an L shape (row ± 2, col ± 1) or (row ± 1, col ± 2)
- Pawn: Moves 1 square forward
    - 2 if it has yet to move
    - Captures diagonally forward one square

Other moves, such as castling and en passant, must also be considered in the respective classes.

When being drawn to the text display, pieces must also uniquely print out a symbol to represent itself. For this design, letters are used. Uppercase represents a white piece and lowercase represents a black piece. To implement this, **Piece** calls its *print_piece()* function, which then calls a private virtual void method *print()*. This method will be overloaded for each unique piece type

- King : K or k
- Queen: Q or q
- Rook: R or r
- Bishop: B or b
- Knight: N or n
- Pawn: P or p

The class that will manipulate pieces is **Player**, which abstractly represents a singular player of the board game. Their "side" is simply represented by an int *side*. Players hold references to every piece they own. That is, the white side player will have pointers to white pieces in the vector *owned_pieces*, and the black side player has pointers to black pieces. Players must also be aware of the board and the cells throughout the duration of the game, thus they hold a reference to **Board** *board. Since players must be able to resign, which is independent from the current state of the board, pieces, or cells, players have a bool *resign* which is set to true if they resign. **Board** will check if any player has raised the resign flag and declare the game over if it is true.

Player has a public vector<int> *move()* method in which the player will make a move once it is their turn. *move()* calls the virtual function *make_move()*, which is overloaded by the **Human** and **Computer** subclasses, as they make moves differently. **Human** reads move from stdin in the form of "move a1 b2", whereas **Computer** reads "move" from stdin, then chooses based off of a preset algorithm, which move to take. For this reason, **Computer**, and by extension **Player**, must observe the board to select the best move. Each call of the while loop in **Board**::*run()* will check *turn*, then call the player who corresponds to *turn*, and call their *move()* function, thus implementing the different turns and moves in a chess game.

In the **Computer** levels, there are currently two implemented levels, level 1 and level 2. These levels are created as subclasses of the abstract class **Computer**. The virtual method *make_move()* is also kept as abstract in **Computer** and is overwritten in the subclasses **LevelOne**, **LevelTwo**, and so on. This means that **Computer** is an abstract subclass of the abstract class **Player**, allowing for easy implementation of the different levels.

In level 1, the algorithm makes valid moves on any random piece. To obtain this condition, a pointer to a piece is chosen from the available pieces on the board (while iterating through all the cells of the current board) that is on the side of the player, then the piece is checked to have at least one valid move. If there are 0 valid moves, a new piece is chosen. After that, a random valid move for the piece is chosen with the function *get_random_valid_move()*. Lastly, since *get_random_valid_move()* returns a cell pointer, the coordinates of the cell are then obtained using *get_row()* and *get_col()* to then be used to update the cells on the board.

In level 2, the algorithm prioritizes capturing and checks. Since valid_moves is an unordered_map<Cell*, int>, the int value is utilized in a way to identify the type of move it is.

The different types of moves are:

- A move into an empty cell (1)
- A move to capture a piece (2)
- A move that checks (-1)
- A move onto a piece of the same color (3)
    - Known as “protecting” the piece
- A move that differentiates from general logic (4)
    - Examples include castling and en passant

These integers serve as conditions when determining the priority of the move in level 2. In this algorithm, it iterates through all the pieces that are on the board that are owned by the player. Then, it checks if the number of valid moves are greater than 0, then it checks if any of the valid moves either capture a piece or checks. If not, it selects a random piece that has a valid move just like level 1.

Level 3 can be implemented by utilizing the number of threats towards a piece. Currently, the threat levels are already implemented and stored within the cells. Also in level 3’s algorithm, there should be logic to determine the threat levels of the valid_moves surrounding the piece and pick the most optimized one for the piece to move towards.

Observe that a computer will be selected from a pre-calculated set of moves that are already known to be valid, and hence will also make a valid move. However, for **Human**, as it is reading from user input, it must check the correctness of the move made. First, it checks the first position (row_initial, col_initial). It will check the board and the specified cell to see if there is a piece residing on that cell. If not, the move is invalid. If there is, **Human** then checks the piece's *valid_moves()* to see if (row_final, col_final) is a valid cell to move to.

After making a move, **Board** must record it. Therefore, *move()* returns a vector of ints which provides information about a move, in the form of

[row_initial, col_initial, row_final, col_final, current_piece_id, last_piece_id]

where current_piece refers to the piece that is currently on the cell and last_piece is the piece that was previously on the cell. Though last_piece_id is usually -1, as the cell is most likely empty prior to having a piece moved to it, in the case of capturing and pawn promotion, **Move** needs to track these pieces that were removed from the board.

Move is a container that points to the **Cell** *initial_cell* and *final_cell*, the **Piece** *last_piece* and *current_piece* and an int turn which tracks which round the move occurred. This object allows for the undoing of moves, which is implemented in this design by a player reading in the command "undo". As memory stores the state of the board, cells, and pieces prior to and after the move, it is simple to revert the board to its prior state one turn before. Observe that this is required as in chess, when a side's king is under check it absolutely must make a move that frees them from check. As such, the program will attempt to move. If it discovers the king is still under check, it will send an informative message to standard output and undo the board, prompting the player to retry.

In order to display the chessboard each round, a **TextDisplay** object is created, which holds a reference to a board and prints out the board depending on its type. In this case, it would be "chess". **TextDisplay** makes use of **Piece**'s *print_piece()* function and is called each round in order to update the output. This object allowed for the further implementation of displaying threats on the chessboard if the player chooses to. By inputting "threats-white" or "threats-black", the cells on the board will display the number of pieces of that colour that are threatening themselves.

Setup mode is straightforwardly implemented by initializing a board if needed. Then it can call **Board**'s modify functions to edit cells and add or remove pieces from them, adding to **Board**::pieces if needed. Once "done" is read, it checks the pieces for exactly one king of each color, and also checks if pawns are not at the back rows. Then, using the same method as above, it will check whether or not either king is in check.

The implementation also includes an undo mode, where upon adding the flag *-undo*, the game will allow the undoing of moves by typing in "undo" an arbitrary amount of times. If the player wishes to make a move, they must type "play" instead and the game will proceed as normal. The crux of this design is the previously stored **Move** objects in *previous_moves*. The program looks at the last move stored in the vector and reverting the board state based off of the initial and final cells, and the initial and final pieces. Note that multiple **Move** objects may be considered as in some instances, such as castling, multiple pieces have moved and therefore need to be reverted. This is tracked by checking if the turn of the **Move** objects are equivalent, that is they were made on the same round.

5. Resilience to Change

Our implementation is highly adaptable and reusable due to its use of abstraction and multiple interfaces. It follows the single responsibility principle, in which each class carries a unique and specific task. As an example, the **TextDisplay** class is very flexible and allows a variety of different implementations. For example, it could be possible to create different styles depending on user preference, using a variety of fonts, symbols, or colors. A more concrete example is the undo mode, which was straightforwardly added due to the use of **Move** classes to manage the moves taken throughout the game.

In order to implement a book of standard openings, our program can store such openings as **Move** objects stored in a vector. While this vector is not empty, an overloaded version of **Player**::move(Move *move) is called in which a player, human or computer, is forced to make the move passed to it. Once it is empty, players can call their own *make_move()* as before. This is easy to achieve as **Player** is abstract with a virtual make move function, allowing for overload, but also **Human** and **Computer** to proceed making moves throughout the game.

Players can undo their last move by utilizing **Move** objects, which store all previous moves. They can undo moves by going back to the **Move** in the stored *previous_moves()* vector with the turn that they wish to return to. The program can either replay the entire game from a fresh board to that specific **Move**, or moves can be reversed by going back through the vector until the desired turn. This can be shown by the undo mode in the implementation. where players can type "undo" an unbounded number of times.

Multiple players can be implemented as **Board** freely allows as many players to be added to the **Board**::players vector. Then some chess classes will need to be changed depending on the color of the player. For example, previously it was "white" and "black", but with 4 players it may be "red", "blue", "green", "yellow". This is needed for the scoreboard. Also, pieces' *print()* method must be changed to allow for these different colors, as simply uppercase and lowercase letters are not sufficient when there are more than two players. Outside of these additions, the players and the **Board**::*run()* method can function as normally, since *run()* updates *turn* based off the number of players, not just by swapping between 1 and 0.

6. Final Questions

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

Throughout this project, we all learned a lot about time management, version control and the importance of planning. We also understood the importance of having an accurate UML / documentation on what the components of code were supposed to do. Usually, we work alone on assignments and projects, and this was the first time we had to work together and build off of someone else's code. Having a fresh pair of eyes looking at your work also helped catch mistakes that would have gone unseen otherwise. We also learned a lot about git, more specifically how to merge and solve code conflicts. It was very necessary to keep a high level of communication throughout the project, which helped us divide work and keep track of our progress.

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

Ideally, we wish we had a more cohesive plan and a better-designed UML, as well as clear up any confusion before starting the project right away. This caused many struggles throughout the coding project as there was confusion as to exactly how the implementation, and by extension, the code, should work. We worked on many pieces of the code simultaneously - which led to some confusion since the base class was being built at the same time the dependent classes were. Since a game of chess has many moving parts that all depend on one another, it is difficult to work in parallel as many classes will depend on the implementation of one another. If we were to start over, we would have agreed on the base class structure first before using that as a stepping stone to creating the components that depended on the base class code. It would also be beneficial to thoroughly discuss implementation ideas prior to starting code, as opposed to repeatedly adding and removing functions or parameters we may or may not have needed.

7. Conclusion

In conclusion, the project has aimed and succeeded in implementing most of the desired features for a game of chess, as well as expanding into extra features which the user may find enjoyable. A considerable amount of effort was placed into following the object-oriented programming principles described throughout the CS 246 course, such as the observer pattern and good design idioms such as NVI. It was an incredible learning experience and the project may be further expanded on in the future.