Intro:

Chess is a strategic, turn-based board game with the objective of "checkmating" the opponent's king. The project involves implementing each of the chess pieces, and their behaviours through abstract classes and concrete implementations. We take advantage of Decorator and Factory Method to ensure a high level of modularity, and low coupling. This in turn has allowed for new features and changes- which is shown in the addition of a sophisticated lvl. 4 bot with intelligent move selection, a special game mode named *Atomic Chess* and useful features such as *Undo*.

Overview:

- The project heavily requires on the working of each piece, especially the design patterns (which will be discussed in the later sections), without any single piece, the project would be exponentially harder to implement in the function
- When creating the project, we designed it to allow as much variable change (which will also discussed in the later sections) which allowed me to implement my functions very easy, as each function has a abstract parent class which allows for low coupling and higher cohesion, as the subclasses implement the pure virtual methods.
- Here is a high level quick overview of the structure of our project:

Game

game.cc is our main/harness, which the client interacts with. It contains an aggregate relationship with observers, and players. That is, it can contain some observers/players, but is not necessary. game.cc contains a composite relationship with board class, since every chess game must have a board.

Player

The client passes inputs through the game, which is translated in players (abstract), which is implemented by humans and another abstract class computer (which is implemented by lvl1, 2, 3, 4). As required, the level 1 computer behaviour is completely randomized, while level 2 computer behaviour prioritises randomized attacks over checks over randomized moves (attacks > checks > random moves that don't fall into the above). However, in level 3, we prioritize saving our pieces on top of the level 2 priority behaviour (saving pieces > attacks > check > random moves that don't fall into the above). Level 4 behaviour will be explained in the extra features section.

Board

In the board, we have access to almost every object class we implemented. This is where all the calculations are made, and the changes made to the piece's position, en-passant, castling, legal moves, etc. The board has access to a 2D array of pieces.

Pieces

Abstract parent class for knight, king, etc…. Also has a concrete component, *blank.* Each implemented decorator class has its own unique defined movesets, and generates a set of valid moves (all possible moves that do not include legality, i.e. pinned, checked, etc.)

Observers

The observers are the graphical interfaces you work with. There is a text observer and graphics. The subject is Board, which is passed through in the observer methods.

Edge Cases

We did come across some difficult to handle edge cases such as en-passant, castling, and some bots. These edge cases will be further discussed down the line.

Design:

Decorator Pattern

The board stores its pieces in an 8x8 shared_ptr vector of **Pieces (Abstract Class)**. The **Piece** class is the abstract base class for all types of pieces (kings, knights, etc.), and it has a concrete component class ***Blank*** (a blank spot in the board grid). In every piece, we stored its current position in the board, has-moved (en-passant, castle), and a possible move and attack function. To do this, we have made a new class which holds their current pieces, which is the ***Position*** class. The concrete decorator classes are types of pieces (king,queen, knight ,etc). These classes implement the abstract methods from pieces (possibleAttacks, possibleMoves), and back a vector of all possible positions where the piece can move/attack, which will be checked in our *Board* class to see if it is a *legal* move (not legality vs possible moves).

We believe that by far, this was one of the most important patterns we needed in our game. By giving each piece a single abstract task, to generate possible moves/attacks, and with each type of chess piece adding onto that task, it helped us create more flexible and reusable code. By doing so, we could add more functionality to each piece without worrying about blowing up our main client.

Factory Method Pattern

The *Player* class is an abstract base class for the players that the user will define when starting up the game. These *Player* classes are responsible for 1 thing only, handling moves, and nothing else. We have 2 main classes, the *Human* class, which is the player in which the human defines the inputs. Their player move function is responsible for moving only valid inputs that the user inputs, and keeps asking until we get one. We also have the computer class, which is abstract, as it sets the base definitions for the 4 difficulties of the computer. Each of these subclasses implements the main function, playerMove(). The sophistication of how each of these subclasses increase as we increase in levels. By adding the factory pattern for enemy moves, helps to increase the flexibility and freedom that we have in handling our moves. As the board and the players aren't dependent on each other, it helped us keep our client code clean, and helped us with code maintenance and debugging whenever there were issues.

A big part of the ***Players** class* is the ***Computer** Class (Abstract base class)*, and its implementations (***Computerlvl1, Computerlvl2, Computerlvl3, Computerlvl4 Classes***) each concrete implementations have different number of vector of ***Move** classes* (Move contains 2 ***Position** class* fields: Prev Posn, Curr/Move to Posn). We take all possible moves from every piece that is on the team of the computer, and pass them through the board to get legal moves. The bot then choose a random legal move from the vector of Move objects

Observer Pattern

In our chess game, we have 2 different observers, we have a text observer, which outputs on std::cout, and a graphic observer, which outputs onto X11 windows. These are both subclasses of the abstract concrete observer class which passes down the notify function to their subclasses. In our case, our concrete subject is *Game.cc*, where it continuously gets the state of our subject, which is the *Board* class, as it holds the chessboard in which we are printing out. By ensuring that our observers have only 1 task, which is to output the board, not only did it make testing easier in our beginning stages, but it also made it far easier to edit our functions without worrying how it would seem on our client side, as they wouldn't affect the observers.

Board & Game:
We have a 2D vector of shared_ptrs to **Piece objects** which allows us to pass the valid moves of each piece, and validate the legality of each move. For example, if we generate a vector of valid moves from the Queen, we pass that vector of valid moves to the ***Board*** *class* to check the legality of the moves. The legality of moves are concerned with:

1. Does the move cause a Check or a Mate? (i.e. pinned)
2. Castling? En-Passant?
3. Stalemates
4. Other legality

These functions then return a boolean value on whether or not the move is legal. If the boolean value is false, then generally, the player returns another move to check for legality. This repeats until the chosen move is legal and can be played. **Game** is our test harness. It contains a static Board and all the observers and players required for Chess to run!

Resilience to Change:

We tried to design our code to be modular and designed to minimize coupling and maximize cohesion. To illustrate, let us start with the use of dynamic objects over static objects in the **game (main).**

In the **game (main)**, we primarily opted to have an aggregate relationship with most of our implemented objects (i.e. Players, Observers). We set Players and Observers as vectors to unique_ptrs (for memory management). This allows us to dynamically add more Players & Observers as we wish, without fundamentally changing our code.
How does this address Coupling and Cohesion?
Having different aggregate relationships allows us to be flexible to add or remove other objects, which removes the reliance of the **game (main)** to other aspects/modules, lowering coupling.
For example: What if we wanted 3 or even 4 players at a time?
Simple! emplace_back a new player onto the vector.
What if we wanted another graphical display that draws the pieces with more detail?
Add a new graphics class, then emplace_back to the vector!

Next, let's talk about the ***Board class*** and how that promotes modularity and resilience to change. **Board** is basically our centre of operations. It contains everything that is required to change the board. i.e. move the pieces, checkmates, checks, stalemates, undo, ect.
Each of these calculations are stored in functions that take in a large variety of parameters (i.e. player team, the 2-D grid of pieces, current position, next position, etc.) This greatly modularizes, and allows us to make changes quickly if necessary. For example, if we needed to change checkmate, or check for whatever reason, we don't need to mess with the main function, we simply make the changes in the functions as required.

How does this address Coupling and Cohesion?
Many of the parameters we actually pass in these functions are actually objects from other elements (i.e. Pieces, Position, ect.), as such the Board does not have much dependence on these objects, should changes were to occur.
Splitting these objects from the board and having their own independent classes, allows for elements to be independent which decreases coupling and allows for more concentrated elements that serve a specific defined purpose (increases cohesion).
Board is simply the centre that takes all these independent elements (Players, Pieces, ect.) and connects them together.
That way, the **Player class** doesn't need to know how **Piece**s is defined or implemented, because the Board handles that.
The **Player class** simply provides a move action command, and the Board handles the rest.
Conversely, **Piece**s doesn't need to know how **Player** works. It just does its job and gets translated through the **Board.**
Overall, this promotes a loosely coupled design, and encourages its other modules to have its own specific/unified design function, i.e. Highly cohesive modules.

Additionally, our design pattern implementations allow us to keep the project lowly coupled, while highly cohesive.

For example, in our factory method, we keep the **Player class abstract.** This ensures that we can have independent implementations of this class, each having a specific purpose.

For example, a concrete implementation of the **Player class** is the **Human class**. The **Human class** only implements the moves for a human, and specifically so. This gives 1 central purpose to the **human class**.
For the **Computer Class,** it is the base factory class for **Computerlvl1,2,3,4 classes**.
In each of these classes, we can implement exactly what we need to.
The best illustration of this would be the observers. We have an **abstract Observer class** that is implemented with **Graphics** and **TextObserver**.
Both of these classes are made with the intent to display graphics to the user, but they both do it in very different ways.
Graphics utilizes XWindows graphics to display the chess board, whereas TextObserver uses iostream to display the chessboard. We separated the classes to promote the higher cohesion while limiting the amount of coupling present.
How does this address Coupling and Cohesion?
Separating the classes into a design pattern (Observer design pattern, Factory Design Pattern, ect.) allows us to have a parent abstract class which doesn't need to know the concrete classes' definition info. It simply needs to be filled. This way, if we wanted to change the XWindows graphics, we don't have to change the whole entire **Observer** code, we simply change that class's code. A similar point can be made in the **Player modules**.
That way the abstract classes are not dependent on other classes and can be standalone, which lowers the coupling present.
This also promotes that each class has a very specific function in mind. **Player module** only implements the moves a player makes, and doesn't need to decipher what the **Board** is doing, and the **Observer Module** doesn't really care about what the **Board** is doing. It simply takes the **state** of the Board, and outputs it on the screen, therefore increases the cohesiveness of each module.

Similarly, our decorator pattern allows for the same types of additions. As we only require the pieces to generate the possible moves/takes at their current position, we can create as complex of an algorithm as we want, and as long as it returns moves/takes, we don't have to worry about it affecting our main code.
The low coupling throughout our entire classes/functions enables us to easily add new features to our code. Not only do the aforementioned design patterns have very low coupling, but so do our classes' functions in general. We have a separate function to see if we are in check, in checkmate, in stalemate, and a whole function to see the state of the board, this allows us to change essentially every single winning condition in the game individually, without worrying about affecting any other function, which makes it very flexible and open to change.
To illustrate our point, we had a different function to add pieces, remove pieces, to swap pieces, to copy pieces, and to even promote pieces. We also had different case handling for each move, as the separate cases were moving pieces, taking pieces, and even enpassant and castling. So very little of our code was hardcoded to just fit our base version of chess, and meaning it allows for a lot of different possibilities and changes.

A testament to our code's resistance to change was when we implemented some bonuses for this assignment.
For example, when we implemented atomic chess, we knew that most of our functions were independent and each only provided one purpose, we only had to target the necessary functions to change, which was fairly easy. Without the low coupling, adding this new game mode would take a lot of extra testing and bug fixing, however, by ensuring we have low coupling and high cohesion, adding atomic chess required very few and very simple changes.
Another very easy addition for us was to add infinite undos. In this case, we can store the temporary boards in game.cc, and revert back to this state if necessary. This was made very easy, since we implemented a deep copy of board and pieces, where all other classes were either:
1. Not required for implementation
2. Statically typed and does not require a deep copy.
In the end, we just called this deep copy, and stored it in an easy-to-keep deque, to allow for unlimited undoing.
The modularity and over cohesion and coupling in our code base makes it simple to make new additions or changes to the code.

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.

We believe that the answer to this question would stay mostly the same. This is b/c maps have extremely efficient look-up times, which is quite imperative in order to implement a vast number of openings and its variations.
Now that we have implemented the actual functions, it is clear what we need to do to achieve a book of standard openings.

```
class OpeningMoves {
      std::map<std::string, std::vector<Move> openings;
      ...
      ...
}
```

Storing a object of “OpeningMoves” in an instance of computer lvl1,2,3, or 4, we would simply random select this openings (where the key is stored in an array of strings), we would pass in the positions back to into board, where it will validate the moves. In the event that the move is illegal (i.e. it is a pinned move, or something is blocking it), we will just take the next alternative stored in the vector of Moves (for reference, we implemented a move that stores currPosn : Position, and nextPosn : Position).

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?

This question was implemented as a part of our bonuses. We actually stayed very close to what we set out in our previous submission. In the previous submission, we claimed to use a linked-list-like structure to store previous Board states, and cycle through.
In the actual implementation, we used a deque (default deque class by C++), in order to have fast pop and peaks at the back and top of the list. Whenever there was a turn, we created a temporary board that copied the previous board state (via copy ctor).

1. If (the user wants to undo) pop the top of the deque and set the current boards state to the popped board from the deque
2. else push the previous Board state in our deque

3. Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!).

This question will also be tackled similar to how we would initially tackle the question. We already have a vector of players, as such we can simply add more players (either humans, or bots).
We also have a team as an int, thus no changes are required for that end. Since we pass in a team as a parameter for many of the board's and player’s functions, the code is modular enough to stand on itself.
We simply need to change the harness, that is, we set turn to an int, to accommodate for more players, and can change the board size if required.
One of the only major changes we would require is to change the pawn possibleMoves and takes. We implemented the move sets of pawns to accommodate for a change in y positions, and not x positions, as such we will have to change that aspect. Every other piece stays the exact same.

<u>**Extra Credit Features:**</u>

Unlimited Undo Function:

We implemented an unlimited undo function for the game. To do this, we have a deque object (from <deque>) that stores a static Board object. During the start of each turn, we store a temporary past Board object. When the player inputs the command, it has the option to request an "undo".

1. If (undo) pop the top of the deque object
2. Else push to the top of the deque object

In the event that 1. Occurs, but there are no pastBoards in the deque, it will display that there are no undo's left (This only occurs at the start of the game since no boards are stored).

This implementation of undo is efficient enough, so that we get an unlimited number of undos. This has been tested to such that the size of the deque can equal to 1000+ without any memory leaks or seg faults.

Memory Management using Smart Pointers:

Throughout our program, we implemented the use of *smart_ptrs*, specifically *unique_ptrs* and mostly *shared_ptrs*. In our program, there was no case in which we needed to use the **"new" or "delete"** keyword. Furthermore, all pointers were implemented using *smart_ptrs*. Also, we ran Valgrind on all of our tests, and none of them had returned any memleaks. As a result, we believe that all of our memory management was handled using smart pointers.

Computer Level 4 Implementation:

Our computer level 4 implementation took a huge leap from our level 3 implementation. This is the order of priority of moves:

Note: We will be talking a lot about simulating moves below. This is possible by making a deep copy of our board for each of the listed moves, moving that possible move, and seeing if our desired result is achieved.

General Priority: Taking Over Moving:

This is a general priority all priorities have to follow. If we can take a piece rather than moving, we will always prioritize taking. Furthermore, we will prioritize taking pieces in this hierarchy (excluding King): Queen -> Rook -> Bishop = Knight -> Pawn. So for example, if we could take a queen or rook, our pieces would prioritize taking a queen first.

Priority 1: Checkmating in 1 Move

Since our priority is to win, if we can win in 1 move, we will prioritize that. The way we found checkmating moves were simple. We find all of our possible moves/takes. We simulate each one of them, and see if any of these moves can checkmate the enemy immediately. If we can, we store that move.

Priority 2: Stopping checkmates 2 steps ahead:

However, if we cannot checkmate immediately, we want to prevent ourselves from being checkmated. This was implemented by looking at all of the enemy's possible moves, simulating them, and storing the moves that checkmated us.

Once we stored the possible checkmates, we looked at all of our current possible moves, and simulated each one of our possible moves, and stored the ones that would stop each one of those checkmates.

<u>Priority 3: Avoiding capture with priority sacrificing:</u>
Unlike our level 3 counterpart, our level 4 avoiding capture is also more sophisticated. The basic piece

It will first look if our highest value piece (reminder: Queen -> Rook -> Bishop = Knight -> Pawn) is under threat. If it is, it will check all possible moves to save it. Our computer is willing to sacrifice any piece that is lower/equal to it on the hierarchy to save that piece if moving away is unavailable.This will continue down the hierarchy until we reach a pawn.

<u>Priority 4: Prioritizing Attacks</u>
Unlike our previous counterparts, as explained in the general hierarchy, we are prioritizing attacks on pieces with the highest values first, in order to maximize our winning chances. This is done by looking at all the stored attacks, and picking the higher value pieces to take.

<u>Priority 5: Anti-Death Checks:</u>
One issue that we saw in the previous moves is that when we were prioritizing checks, that our pieces were moving within king range when checking, with no hope of checkmating, causing the king to take them and our computer losing pieces for no purpose. As such, in the level 4 computer, we designed it such that we will still prioritize checks, but not ones that will result in the piece getting pointlessly killed (we see if the check will end up in a 1 square radius around king).

Furthermore, we also prioritize checks that also take pieces rather than not taking pieces (also following the piece hierarchy in what pieces to take)

Note: We check for checks just by seeing all our possible moves, simulating them, and storing the one that ends up checking the enemy piece.

<u>Priority 6 (General Moves):</u>
These moves are the leftover moves that the computer has. There is no priority in these leftovers, and we are just randomly picking from them.

<u>Atomic Chess:</u>
We also decided to implement an additional popular chess game mode, which we found pretty simple to implement due to our high cohesion and low coupling. The rules of atomic chess are simple:

1. If you take a piece, the piece that was taken will blow up your piece (including itself), and all surrounding pieces (excluding pawns), regardless of which team it belongs on.
2. We can note that kings can also blow up normally, when other pieces explode, and they also blow up when they take other pieces, so in this game, the king is extremely vulnerable.

In order to do this all we did was adjust the take piece function (include en passant), the checkmate function, the updateState (to check for checkmate) functions which were pretty simple, since they were all in individual functions, and affecting them barely affected other processes. Furthermore, since our computers also base off of the board's valid moves, the computer will also adapt to these new factors, and adjust their priority of moves based on what they can do.

We can note that if we didn't make our program as variable and free to change as it is currently, it would be extremely hard to implement atomic chess or any variance in chess.

Final Questions

1. 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?

This project was very eye-opening to us in many ways. For one, for many of us, this was one of our first experiences in collaborating for a big software project that involved many significant moving pieces. As such, we believe the most important thing we learned in software development would be ***effective communication.***

This entails:

1. Who completes what portion of the project
2. Where and who does each person work?
3. Separate git branches
4. Push and pull order
5. Talking to each-other
    a. Talking about how to tackle problems we don't immediately know
    b. How we want to design the project
    c. Completion dates

Specifically talking about remote git repos, I think figuring this out early was extremely beneficial to us. Previously I have destroyed my git commit history and codebase b/c I didn't know how to merge properly (everytime I would git push –hard). Learning how to work around git branches and properly communicating git push and pull order was so important so that our work does not overlap, and the code can function without changing anything.

Another thing we learned a lot about is enforcing good design patterns and topics covered in this course. For example, there were instances where we straight up destroyed the board, b/c we didn't enforce invariance. In the end, we took more precautions in making consts, and encapsulating methods and classes, so that we only changed the stuff we needed to. We found the design patterns learned in class (Factory Method, Observer Pattern, Decorator Pattern) were extremely useful in implementation. For example, implementing Pieces was so streamlined for us, b/c we immediately knew that it was a textbook example of a decorator pattern. We even implemented the concrete component of the decorator pattern (i.e. our Blank object).
With the idea of abstract parent methods, it made it way more organized in general. For example, when we had a bug with the bots moves, we immediately knew it had to do with

Piece's possible moves and takes, since that is the only possible place where the moves can be generated.
When the bot makes a move that puts itself into check, we know immediately the bug is in the board class, since that is where we actually processed the legality of the possible moves.
To summarize, we learned a lot about the importance of communication to keep workflow organized and clear. Furthermore, upkeeping good OOP design patterns lead us to a very streamline process in implementing functionality and made debugging wayyyyy easier (BLESS).

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

If we had a chance to start over, it would put more pressure on enforcing invariance, and perhaps think more on the implementations of the bots.
As previously mentioned, there was 1 instance where we passed in a regular board reference when in reality we should have had it protected or private, or passed in as a const, and messed up the entire board.

A minor bump in the road of implementations was definitely the bot. There were some instances where we were careless in implementing the bots, which led to a lot of bugs that we had to fix. Thankfully, a lot of the design practices we upheld made the debugging process quite smooth (as mentioned previously).

I think overall, the process went quite smoothly. In the first due-date, I personally (just one person) was quite panicked as to how we can implement all the requirements on time, but we actually ended up finishing ahead of time. This is thanks to our thorough and thoughtful UML design, which gave a lot of structure to what we needed to implement and the order we needed to implement them in.

Conclusion

Overall, I believe that our good planning resulted in a streamlined and an easy process to complete the chess project. We were ahead of schedule by a few days, and this let us work on the bonus features for much longer than we expected. We think this was a good learning experience, and we would do this project again.