CS 246: A5 Chess Design Document

1. Introduction:

Once the final project was released, we looked into the specifications of each of the different games and decided to go with chess. We then created our initial UML trying our best to incorporate OOP principles and the design patterns learned in class as well as a project timeline to make things go smoothly and avoid last minute changes.

During the days following the first submission, we made best use of the latest concepts learned in class including inheritance, polymorphism, encapsulation, and data abstraction. We also tried to achieve high cohesion and low coupling in all of our classes to improve code readability and followed best design practices.

2. Overview:

On a bigger level, our program follows the MVC (Model-View-Controller) model where the game is the controller, the board the model and the textdisplay and graphicdisplay the view. Board manages most of the data for the game and these classes along with others all interact with one another to create a working chess game. Game takes in user's requests and calls board's methods which update the board. Board further calls textdisplay and graphicdisplay to update the user view.

We used various design patterns to support MVC as well as the 2 players discussed in detail later.

3. Design:

- **Classes and Inheritance**

  We employed various OOP principles in our program to structure and design our code effectively. We focused on utilizing objects that are instances of classes. Our goal with this design plan was to organise code in a more reusable manner and enhance code readability. Classes are one of the main concepts of OOP and we used these as a blueprint for many of our objects. In our chess game, we created a superclass "Piece" and subclasses such as "Pawn", "Rook" which inherited from this superclass. The piece superclass has all the common attributes which are shared by the subclasses. This also means that any new type of piece can be easily added and should also have these attributes. This allows for code reusability and extensibility. It also becomes easier to modify a subclass without affecting the core structure of the code. Moreover we did the same thing with the Player superclass, where we have Computer and Human as different types of players. In these classes, **polymorphism** is evident as pieces if various types can be manipulated with a common superclass. We also used **Abstract** classes through the program which allowed us to capture common attributes and methods without having to be specific for each subclass and implementing accordingly.

- **Encapsulation**

In our implementation, each chess piece encapsulates its specific behavior, such as how it moves or captures opponents. This encapsulation prevents direct access to the internal details of a piece from outside the class, promoting information hiding. We also used access modifiers such as private and protected to ensure only the required class or subclass had access to specific fields and methods. For private fields, we used getter and setter methods to enforce encapsulation and providing controlled access

- **MVC**

  We used Model-View-Controller architecture in the design of our chess game for separation of responsibility. The Game class as the Controller orchestrates user interactions, the Board class manages the game state, and the TextDisplay and GraphicsDisplay classes serve as Views, providing different ways to visualize the game.

  Model:
  The primary model component in our game is the Board class which manages the state of the chessboard, and the Piece classes which encapsulate the behaviour of individual chess pieces. The Model is responsible for enforcing the rules of the game, handling moves, and updating the state accordingly. The implementation of the Observer pattern, as previously discussed, allows the Model to notify registered observers (Views) about changes in its state.

  View:
  The View is responsible for presenting the data to the user and outputting user input. The TextDisplay class serves as the textual representation of the chessboard on the command line, while the GraphicsDisplay class presents a graphical view in a window. These Views observe the Model and update their representations whenever the Model's state changes. The separation between the Model and the View ensures that the display logic is independent of the underlying game logic.

  Controller:
  The Controller manages user input and translates it into actions that affect the Model. In our chess game, the Game class serves as the Controller. It receives user requests, such as moves or commands, and delegates them to the Board (Model) for processing. The Game class acts as an intermediary between the user and the game logic, facilitating communication between the user and the game. The Player classes also (HumanPlayer and ComputerPlayer) act as Controllers, handling user or AI input and making appropriate calls to the Model. The Player classes encapsulate the logic for choosing and executing moves based on user or AI decisions. By encapsulating this logic within separate Controller classes, we ensure that the user interface and game mechanics remain loosely coupled.

- **Design Patterns**

  We implemented the observer pattern to display our game on the command line as text and also graphically in a window. This pattern allowed us to decouple the game logic from the display logic, allowing for independent modifications in each component. We used the notify method which allowed communication between the subject (Board class) and its observers (GraphicsDisplay and TextDisplay classes). Everytime a piece in the game is updated we call the notify method for the subject from our Abstract class subject, which notifies all the observers attached to our subject stored in an observer's vector. The observer pattern also supports the relationship between the Model and the Views. Views register as observers of the Model and update their displays whenever the Model undergoes changes.

- **Cohesion and Coupling**

  In our program, we ensured high cohesion by structuring the code in a way that promotes collaboration. The core of our design lies in a well-defined Game class, which encapsulates the overall game logic. This class collaborates with a Board class, responsible for managing the state of the chessboard and the interactions between pieces. The Board class, in turn, collaborates with individual Piece objects, each representing a specific chess piece. This hierarchical organization ensures that elements within each class cooperate seamlessly to handle distinct aspects of the game, enhancing clarity and maintainability.

  Additionally, we aimed for low coupling, minimizing the connections between different components. The Game class, for instance, communicates with the Board class to manage the overall game state, but it does not have direct access to the internal implementation details of individual Piece objects. This loose coupling allows changes in one module to have minimal impact on others. Furthermore, the Player class, which is a superclass for human and computer players, exhibits low coupling with the game logic. Each player interacts with the Game class through well-defined interfaces, ensuring extensibility and adaptability. Each class has its own concerns which results in a more resilient and easier to understand code.

- **Exceptions**

  To increase the reliability of our program we incorporated exception handling to address things like invalid inputs and out of bounds moves. For instance, when a player attempts to make an illegal move or enters invalid input, the corresponding functions responsible for handling these actions throw exceptions. By utilizing the throw mechanism, we signal that an error has occurred, allowing the program to transition to an appropriate error-handling mechanism. This approach not only prevents invalid moves but also provides clear and informative error messages, which aid in debugging and user interaction. We strategically placed try-catch blocks in the code where exceptional handling might arise. Thai prevents the program from crashing and ensures that the game can recover from errors.

- **Changes from initial plan:**

There were not many significant changes in our UML from DD1 however we did change a few things to make our code more efficient. Specifically, we realized that we did not need Move to be a class of its own and we could simply define it as a struct in board to handle levels of automation since it has no class invariants of its own. One of the other things we overlooked at first was giving board a concrete array of Piece objects which as soon as we started programming realized was not a good approach. Since we changed that to unique pointers, some other method parameters also needed to be changed. Thirdly, we did not have enough time to implement level4 for computers so we had to take that out of the UML. We also thought it would be a good idea to give each of our virtual methods of Piece and Player, a reference board as one of the parameters to make data abstraction seamless. Overall, most of the changes are a result of changing what has been described above and simply adding mutator/ accessor methods for private fields to prevent the user from accessing the core underlying interface and enhance encapsulation in addition to making the game our controller which simply calls board to implement the game rules.

4. Resilience to Change:

Our chess program has been meticulously created and executed, emphasising flexibility and resistance to change. This program's resilience is demonstrated in a number of important ways, demonstrating our dedication to building a stable and adaptable chess-playing environment.

- **Modular Architecture and Polymorphism/ Abstraction:**

  Instead of having one big program, we separated different functionality into different modules. Each module handles a different aspect of the game and one such example is the piece superclass which has 7 subclasses classes for the 6 different types of chess pieces and an empty piece (empty square on the chessboard). Hypothetically, if we were to add a different kind of piece, it could have its own subclass with the updated functionality/ rules for movement and none of the main interface would need a change except for adding its value in the enum and a few lines of code. Similarly, if we were to add a different player, it could be a subclass of the abstract player class and override the virtual method makeMove. Therefore, this modular design and use of polymorphism facilitates easy modification of chess rules or replacement of individual components without affecting the entire system.

- **Scalability:**

  The program is built keeping scalability in mind. For example, if the size of the board was to change in a future upgrade, we can simply change the size of the array that contains all the pieces on the board and update the size of the board for array traversal through boardSize, an int variable that tracks the size instead of manually changing all for loops. Using such constants instead of raw values allows for changes to default values to happen much faster and almost always will require only a few extra lines.

- **Additional Features to enhance resiliency:**

Most chess rules have been subdivided and have their individual methods in the board class so a change in rules only requires a change in specific methods instead of changing the entire interface.

The program we built closely follows SRP (single-responsibility-principle) as classes and objects were carefully designed to have very specific functionality and minimise the overlap in responsibility between various classes. This can be seen through how the various pieces are solely responsible for their checking if their own movement to another square is valid. It is also evident with how board only has methods relating to the board and no fields relating to particular piece information like colour or type.

5. Answers to Questions:

Q1: 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. See for example https://www.chess. com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

The best way to go about supporting a standard book of opening move sequences would be creating a child class of the existing Board class. This class would act as a container for storing sequences of opening moves. These can be represented in the form of a map where each sequence is linked to the move or sequence that comes after it, as strings or be made into different subclasses whichever is deemed more efficient. The Board class can have an accessor function that is given the previous sequence of moves and that retrieves the next move from the given sequence using the previous sequence as the key. This highly resembles the "book" nature of opening chess moves. Depending on the overall implementation, the accessor function can return either a singular move or a sequence of moves. If no matching key is found, no value is returned. This design also allows for future customizations and enhancements should we wish to change or update the opening moves.

Q2: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

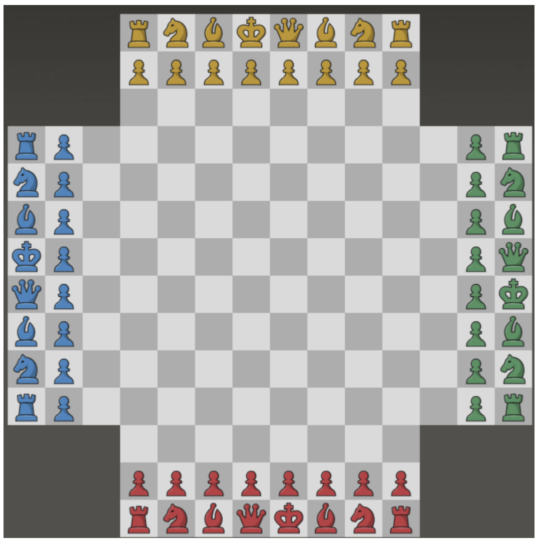
One approach to add the functionality of allowing a player to undo their last move would be to create a stack data structure of the board states throughout the game after each turn. In this design each turn would create a new board that would be appended to the top of the game stack. Through accessing the top board of the stack would be from where the current game state is determined. When it comes time for a player to want to undo their move all they would have to do is input undo and the top board would be popped from the stack and the new board state would be displayed (either through text or graphics depending on the selected mode).

Another approach I considered was creating a vector of moveDescription on board. These moveDescription objects will have multiple fields describing all details of a move such as what piece moved, where to and from, if and what piece it took in the process, the taken pieces location, if a special move such as en passant or castling took place, and other details. Then an undo method would be added to the Board that performs the inverse based on the last move that took place thus moving current pieces on the board to their start location (found from the moveDescription object) and adding back any taken pieces.

Considering these two approaches I think the first is preferred as although it requires saving many board states which are larger objects in terms of memory as compared to moveDescription causing it to be more memory intensive. However this approach is more straightforward and simple as you are just implementing a stack of boards, whereas the undo function is more complex to implement and prone to potential bugs and obscure cases. Also if I were to add features later such as viewing move history in which you can step through the game move by move, forwards or backwards it would be more efficient through the first method (although you would have to change it to a vector or doubly linked list, to allow for O(1) lookup of the next or previous item). However the second strategy would require using either the built-in move function used when playing which is less efficient as it checks if a move is valid or creating a redo function which adds another level of complexity.

Q3: 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.

A four-handed chess would mean 4 players. Firstly, we would need to change the board size. In addition to the 8 x 8 grid, we would also need to add 3 rows and 8 columns (3 x 8) to each side of the board. In our code we would have to make the 4 corner coordinates on each side out of bounds and check that a move these coordinates is not valid. The image can be seen below. Then there are two possible variations. The players can either play in teams of two on the opposite side or four individual players. This could be provided as an option to the user. Assuming we are playing as teams of two on the opposite sides we would need to change many of our functions and checks to account for the extra pieces. We would need to check things like valid moves since you can't move to square if it's your teammate but if it's the opposite team then you can capture. Having double players for each team would double the conditions each function uses to check for certain things. For the game to end we would have to see that both the players on the team have been checkmated. If only one is checked, he can not play anymore and we would need to change the number of turns and consider that the leftover pieces would still be on the board. We also need to change the checks for promotion(change it to the 11th position) and



the capture(each team can capture any piece of the other team). Special Moves would work the same way.

If we are playing individually, we would have to check for what pawn can promote when it reaches the first row of any player. When a player is checkmated their piece would still be on the board and the last player standing would win. In our implementation we would have to change the number of players and turns based upon this. We would also have to change the Graphics and TextDisplay to add the new pieces and board. The observers must be attached to all 4 players and their pieces.

6. 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?

Effective Communication and Collaboration:

Working as a team on this project reinforced the importance of clear, consistent communication. Regular team meetings and status updates helped ensure everyone was aligned and aware of the project's progress and any emerging challenges. This was made evident by how difficult it was to regroup after significant changes were made after not having met in some time.

We discovered the power of diverse perspectives. Each team member brought unique skills and insights, leading to more creative solutions and a robust final product. These perspectives helped enable us to tackle problems from various angles. This was made clear whe

The practice of conducting regular code reviews and seeking peer feedback was invaluable. These collaborative sessions not only maintained the quality of our code but also fostered a deeper collective understanding of our entire codebase.

Project Management and Organization:

The project highlighted the importance of effective project management in a team setting. Assigning roles based on each member's strengths and scheduling regular checkpoints helped us stay on track and meet most deadlines efficiently.

Managing a complex project required us to be agile and adaptable. We learned to be flexible in our approach, ready to pivot our strategies in response to unforeseen challenges or feedback.

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

Structured Approach to Version Control:

Reflecting on our experience, a more disciplined approach to version control could have significantly streamlined our workflow. Establishing separate branches for individual contributions while maintaining the main branch as a stable base would have facilitated smoother integration and minimised conflicts.

Regular code merges and comprehensive reviews should be a standard practice in future projects. This would ensure early detection and resolution of conflicts or bugs, maintaining the integrity of our codebase.

Earlier Integration of Advanced Debugging Tools:

Initially, we underestimated the complexity of debugging a large-scale project. Relying on print statements for debugging was less effective than anticipated, especially for identifying and resolving deeper, systemic issues.

Integrating advanced debugging tools like gdb from the project's inception would have been more efficient. These tools offer critical insights into the program's execution and state, which are essential for tackling complex bugs.

In future projects, incorporating sophisticated debugging tools early on will be crucial. This approach will enable a more streamlined and effective problem-solving process, saving time and enhancing the overall quality of the software.

7. Conclusion:

In conclusion, the development of our chess program has been a collaborative journey that brought together the combined knowledge and enthusiasm of different individuals. Our growth as developers has been aided by the challenges we faced, the hours we spent debugging and the lessons we learned during the development process, which included strengthening our understanding of OOP and observing how a software functions when it is created from scratch.