CS247 Chess Final Design Document

Introduction

We have successfully created the well-known board game chess in C++, which includes multiple features such as standard gameplay, playing against computer bots, graphical and text displays, and a board setup mode. Throughout working on the project, we made effective use of robust design patterns such as the Observer and Strategy patterns, as well as crucial OOP principles such as encapsulation, inheritance, and polymorphism. Our end result is a well-built, robust object-oriented design program that provides an enjoyable chess experience.

Overview

Our implementation of chess can essentially be divided into five main components that perform a central function of the overall program, with each main component itself breaking down into more granular classes and entities to implement all aspects of chess. Additionally, we have also defined a few program-wide enumeration types and custom data structures to help out with passing cohesive data in a convenient format around different components.

The central `Game` class serves as the core logic controller for the entire process, essentially binding all of the other components together to run the program. To do this, `Game` takes ownership of multiple objects; an `Interpreter` to intake player commands in order to perform the desired operations, two `Display`s to display the state of the game in both text and graphic format, two `Player`s that represent the users interacting with the game, and a `Board` object, which represents the actual state of the game. By combining these components, `Game` is also able to handle player moves, maintain overall game data like the score, and check for special game states, such as checkmate, check, stalemate, and more.

The `Interpreter` class is responsible for handling user input and parsing user commands. That is, the `Interpreter` reads a command from `std::cin`, parses its arguments, and returns a concrete subclass of abstract superclass `Command`. These concrete subclasses, one for each command, will contain argument data which will then be used by the `Game` class to influence game logic.

The display implementation utilises the Observer pattern, in which the `Board` serves as the "publisher," generating data and updates in the game, while the displays `TextDisplay` and `GraphicDisplay` are the "subscribers," who receive updates from the board and display it. Essentially, we have the abstract classes `Subject` and `Observer`, from which `Board` and the `Display`s inherit from, allowing us to leverage polymorphism to define what each display does with the information from Board to actually display the game.

To represent the different types of users (namely, humans and computers) and how they make moves, we've introduced an abstract class `Player`, from which the classes `Human` and `Computer` inherit. The only responsibility of these objects are to make moves in the game - we pass in the `Interpreter` into the `Player` so that they can enter in text-based commands that are then parsed and communicated with the `Board`. In terms of the `Computer` class, it intakes

a command, and then calculates its moves by executing algorithms that look at the `Board` to determine its next move through the Strategy pattern. `Computer` itself is still an abstract class with four concrete subclasses, each implementing the virtual function `computeMove` differently according to its algorithm complexity. The players ultimately return their moves in the form of a `Move` object, which is a data struct we've defined that encodes what a move is (position movements, promotion type if necessary, etc.).

The `Board`, `Tile`, and `Piece` classes represent the principal entities of the game of chess, and is what the program constantly manipulates and reads to carry out a game of chess. The `Board` owns (or is made up of) an array of `Tile`s and many `Piece`s, and is responsible for `Piece` manipulation among those tiles, with methods like adding, moving, and removing pieces. Each `Tile` has (or does not have) a `Piece`. Each `Piece` is responsible for knowing what its valid moves are on the given board. `Piece` is an abstract class with six concrete subclasses for the six unique pieces. Each subclass is responsible for knowing what its valid moves are, and the pieces `King` and `Pawn` also take into special moves relevant to themselves, like en passant and castling.

Putting it all together, `Game` runs a main program loop. It constantly intakes `Command`s using an `Interpreter`, and uses these commands to execute logic appropriately. In the case of starting a game, `Game` will run a game loop that alternates allowing player 1 to move, and player 2 to move. The loop calls `makeMove` on the player whose turn it is, which allows the player to input their move. The result, a `Move` object, is passed off to the `Board`, which then determines if it is a valid move or not by asking its pieces if it is valid through `Piece`'s public interface. If it is, the piece is then moved on the board using `Board`'s public interface. The `Game` then does additional checks after this move, including if it has resulted in a check, promotion, checkmate, or stalemate. This game loop continues to run until it finishes. In the case of setting up a game, `Game` runs a setup loop that intakes commands, and generally manipulates the `Board` and its pieces using `Board`'s public interface. In order to display the game, we use the Observer pattern, where `TextDisplay` and `GraphicDisplay` are observers that view `Board` as the subject. Board provides public interfaces allowing these displays to retrieve information, and on rendering the board, the displays are "notified" and display the board as expected.

Updated UML

Please refer to uml.pdf for our updated UML diagram.

Design

Interpreter & Commands

We have a class Interpreter to handle and parse inputs from the user. One instance of the interpreter is owned by the Game class. Every command sent by the user is received by the Interpreter. When user input is desired, the Game class would call the method Interpreter::readCommand(). Inside the method, the Interpreter would read from std::cin to parse

the commands and return the appropriate concrete subclass of the abstract class Command. For example, if the user entered `game human human`, it would return an instance of the StartGame concrete class (more precisely an shared pointer instance of it). Inside of that instance, it stores payload information about the command's arguments. In this example, it would have fields denoting player1 and player2 to be Humans. All Commands have a type denoting which command it is. Therefore the Game, upon receiving a Command, can deduce its type, cast accordingly, and retrieve the necessary payload information.

The motivation for creating the Interpreter class, separate from the Game class, is that we wanted to separate the code for input-handling away from the Game class, which should just be focused on game flow and logic. Therefore we increase the amount of **cohesion** by creating a separate Interpreter class to handle user inputs. This makes our program more resilient to change. If for example, the syntax of a command changes (ie. `+` now becomes `add`), only the Interpreter class will need to change, not the Game class. Therefore we can say the Interpreter class has high cohesion. Furthermore, the Command superclass and its concrete subclasses also have high cohesion, due to their singular purpose to carry a payload of command information from the Interpreter to the Game. Note that there is still a moderate amount of **coupling** between the Game, Interpreter, and Command classes. That is, the Interpreter returns Command classes to carry data and can influence the control flow of Game. However, we argue that due to the necessary data required to be passed and the inherent dependency of the Game control flow on the user's commands, this amount of coupling is required.

While the Interpreter and the Command classes do not follow any particular design pattern, we will note that it is somewhat similar to the Command Pattern. That is, it encapsulates actions (in this case user inputs) as objects. The Interpreter can be seen as a Controller class which creates Command objects that are passed the Game. However, the Game does not really 'invoke' the commands but rather treats them as a payload.

<u>Players</u>

The player system of our application consists of multiple components that make use of polymorphism and the Strategy pattern to achieve low coupling and high cohesion. The `Player` class is an abstract base class that defines a common interface for all player types, with a pure virtual method `makeMove` that must be implemented by its derived classes. This method returns a `Move` object, which is a struct that encodes the information of a move. The main purpose of this struct is to provide convenience in transferring this data to multiple components around the program. Note that this does introduce some coupling with this struct, but the benefits of its convenience highly outweigh any impact. This design allows for easy extension of player behaviours without modifying existing code, which adheres to the Open/Closed principle. The `Human` and `Computer` classes inherit from `Player`, each implementing their specific logic for making moves. The `Human` class processes user input via an `Interpreter` (which is passed into the `Player` object) to generate moves, while the `Computer` also takes in an `Interpreter` (in order for it to be prompted to move) and encapsulates various strategies for

algorithmically generating moves in its subclasses, `Noob`, `Intermediate`, `Pro`, and `Grandmaster`. Each computer player class implements a different level of complexity in its move generation, from random selection to a minimax algorithm with alpha-beta pruning, as per the Strategy pattern, where each concrete strategy is encapsulated within its respective class, allowing the `Computer` superclass to delegate the move computation to these strategies without knowing their specific details. It computes these moves by taking in the `Board` object (which is passed into the `Player` object) and finding valid moves for its pieces through `getAllValidMoves`. This introduces some coupling, as the algorithms of the computers rely directly on the public interface of `Board`. However, the `Computer` class and its derivatives also demonstrate high cohesion by focusing purely on move calculation strategies. Additionally, there is also slight coupling with the `Interpreter` class, but is necessary to facilitate converting user commands into game moves. However, this coupling is minimised and encapsulated within the `Player` classes, ensuring that the core logic remains cohesive.

Game Engine

The central game engine serves as the core logic controller, handling all major game interactions and ensuring smooth gameplay. One of the most critical aspects of our design is its flexibility to accommodate changes. Since the class is designed with modularity and separation of concerns in mind, it enables us to easily incorporate new features or modify existing ones without affecting other components. For example, the class calls upon `Board` to check for conditions such as check, checkmate, and stalemate, and to facilitate piece movement. For instance, when a player attempts to make a move, `Game` validates the move by temporarily applying it on a clone of the current board. This prevents illegal moves from being committed, ensuring the integrity of the game state. Additionally, by decoupling the game logic from the board, we make it easier to introduce changes such as new game rules or different chess variants. The class also handles complex scenarios like pawn promotion and castling by coordinating with `Board` to update the pieces correctly. This modular approach enhances cohesion within the game engine and reduces coupling between different components, allowing for easy maintenance and scalability of the codebase.

Board, Tiles & Pieces

The `Board` class is the foundational element of our chess game, representing the chessboard as a 2D vector of `Tile` objects. Each Tile holds information about its position, colour, and the piece it contains, if any. This class encapsulates all board-related functionalities, including initialising the board, managing piece movements, and maintaining the game state. The design of the Board class exhibits high cohesion, as it consolidates all responsibilities related to the state and behaviour of the chessboard within a single class. Additionally, its design is built to accommodate changes efficiently, as modifications to the board size or the introduction of new

rules can be implemented without modifying the entire system. `Board` also maintains pointers to the tiles containing the kings, which is crucial for checking conditions like check and checkmate.

Each `Tile` in our chessboard is a self-contained class that holds essential information, such as its row, column, colour, and the piece it contains. The design of the class is crucial for maintaining the integrity and flexibility of the board. By encapsulating tile-specific data and behaviour within its class, we ensure that changes to the tile's properties or the addition of new features can be made independently of other components.

Each `Piece`, such as `Pawn`, `King`, `Queen`, `Rook`, `Bishop`, and `Knight`, is derived from the base `Piece` class and contains specific movement logic implemented through the `isValidMove` function. This polymorphic design allows us to easily add new types of pieces or modify existing ones without affecting the overall system. Each `Piece` also has a hasMoved property, which helps determine special move eligibility, such as castling for the King and Rook or en passant for the Pawn. By isolating piece-specific logic within their respective classes, we enhance the cohesion of our design and reduce coupling between the different pieces and the game engine. This modular approach ensures that any changes to the movement rules or the addition of new pieces can be seamlessly integrated into the system.

Displays

The design of our display system of our chess program utilises the Observer pattern to effectively decouple the game logic from the user interface, allowing us to achieve low coupling and high cohesion. The `Board` class, which encapsulates the game state, serves the subject that maintains the list of observers. These observers are notified whenever there is a change in the game state. The `GraphicDisplay` and `TextDisplay` classes serve as concrete observers, each implementing their own update logic to render the game in a graphical or text-based format. By decoupling the game logic from the display logic, we achieve low couponing, where changes to the game state management does not require changes to the display components. This separation ultimately enhances the maintainability and scalability of our application. Additionally, high cohesion is achieved within each class. The `GraphicDisplay` is solely responsible for graphical rendering using an `Xwindow` object that uses the `X11` library for graphics operations, while `TextDisplay` manages console-based output. We also employ a buffer in `GraphicDisplay` to track the current board state, which allows for minimal redrawing and optimising performance by only updating necessary parts.

Changes From DD2

The general structure of our design remained the same from DD2, with a few minor adjustments and fixes after we made some realisations during implementation. Initially, our UML diagram outlined that certain commands, like `StartGame`, `AddPiece`, and `Move` would be responsible for creating players and pieces. We realised that this did not make sense - the purpose of commands should only be to communicate what the user desires, and it should not take on the responsibility of owning game components. Instead, we adjusted it so that all

commands simply report to `Game`, and that `Game` should be responsible for players and `Board` responsible for piece creation and movement.

Further, we also created the `Move` struct and distinguished it from a `MoveCommand` - a `MoveCommand` represents a move command from a user, whereas `Move` represents an actual move on the board that may or may not be made. Finally, in order to support check and checkmate checking, we only realised the complexity of the implementation after writing the code. The main idea was that we had to simulate board movements without actually modifying the real board, since checkmate and stalemate must look one move ahead. As a result, we needed to create temporary boards along with the `Board` copy constructor. After some testing, we realised that since the board had tiles which had pieces, we needed to implement a copy constructor for those as well. Otherwise, the temporary boards would be touching memory related to the real board which would cause a ton of correctness issues. All the copy constructors for these classes were not accounted for in DD2.

Resilience to Change

Interpreter & Commands

The `Interpreter` class promotes significant resilience to change through its abstract design. Firstly we introduce the `Interpreter` class separate from the `Game` class such that if we need to handle user input differently, it is not necessary to change the `Game` class, which should just be responsible for handling game logic. This also motivates our design decision on why the `Human` and `Computer` classes use the `Interpreter`. Our design enforces that the `Human` and `Computer` classes do not directly interact with the user's commands. Rather, they interact through a common interface of concrete `Command` classes, which remain impervious to the actual form of user commands. If instead the user decides to interact with the program through something other than std::cin, it will still be sufficient to only change the Interpreter class. In conclusion, this design of the `Interpreter` and `Command`s promotes a high resilience to change through the abstraction of the user's input commands, such that dependent classes such as `Game`, `Human`, and `Computer` will not need to change.

Players

The player system is highly resilient to change. Adding new player types or modifying existing strategies can be achieved with minimal disruption to the rest of the codebase. For instance, introducing new levels of computer bots would involve creating new classes that extend `Computer` and implementing the `computeMove` method with the desired strategy. The rest of the game, such as the game loop and user interface, would obviously remain unaffected. Additionally, by encapsulating relevant data for a move within `Move`, it centralises the representation of a move, which supports modifications and extensions. Should additional attributes be needed in the future, like timestamps or annotations for the moves (in the case we want to implement move tracking), we can also add these to the struct without altering the

interfaces of the methods that use it. Moreover, the separation of move generation logic in the `Computer` class allows for independent testing and optimization of different strategies without impacting the overall game flow. The design also accommodates changes in the game rules or board configurations, as the move generation algorithms rely on the `Board` to inform them on the valid moves on the board. This adaptability ensures that the player system can evolve alongside the game, supporting new features and enhancements.

Game

The Game class demonstrates significant resilience to change through its modular design. By delegating specific responsibilities to other components like `Board` and `Player`, the Game class minimises coupling as much as possible. This design allows for easy modification or extension of game rules, player types, or display methods without requiring substantial changes to the game class itself. For instance, introducing new game modes or victory conditions would primarily involve adjusting the game loop and condition checks within the Game class, leaving the core game logic intact. The use of polymorphism in player handling (`Human` and `Computer` subclasses) allows for seamless integration of new player types or AI strategies without altering the main game flow. Furthermore, the Game class's reliance on the Command pattern for user input processing enhances its adaptability, as new commands can be added or existing ones modified with minimal impact on the overall game structure.

This flexibility extends to potential future enhancements, such as implementing an undo feature or integrating a book of standard openings, which could be incorporated by adding new methods or expanding existing ones within `Game`, while maintaining its fundamental structure and interactions with the game loop and chess logic. The modular nature of the game state system would also facilitate the integration of these features, allowing for seamless tracking and reporting of game states even as new functionalities are introduced.

Board, Tiles, and Pieces

The resilience to change for these components is achieved through several design choices. Starting with the `Board` class, the object maintains a collection of `Tile` objects using a vector of vectors, which can easily be resized or modified if the board size or configuration needs to change. The methods in the `Board` class relating to piece manipulation are all designed to be generic and applicable to any piece. This generality allows for new piece types or changes in movement rules to be incorporated without altering the fundamental operations of the board.

The `Tile` class encapsulates the concept of a board position and manages the piece it contains. Its design is straightforward and focused, making it easy to extend or modify. For instance, if additional properties or behaviours need to be associated with tiles (such as special tile types for other game modes), these can be added without affecting other parts of the system. The `Tile` class’s methods for setting and retrieving pieces are designed to work seamlessly with the `Piece` class hierarchy, ensuring that any changes to how pieces are represented or managed can be accommodated with minimal impact.

The `Piece` class and its derived types exhibit a high degree to change through the use of polymorphism and encapsulation. Each specific piece type implements the `isValidMove` method to define its unique movement rules. This polymorphic design allows the board to handle all pieces uniformly, invoking the `isValidMove` method on any piece regardless of its type. If new piece types need to be added or existing movement rules need to be altered, these changes can be made within the respective piece classes without requiring modifications to the board or tile classes. The `Piece` class also includes methods to manage piece-specific state, such as whether a piece has moved or special conditions like en passant eligibility for pawns and castling for kings. This encapsulation ensures that piece-specific logic is localised, further enhancing the system's modularity and maintainability.

Displays

The design of the display system is robust and resilient to change due to its modular structure and adherence to the Observer pattern. Since the **Board** class communicates with its observers through a well-defined interface, adding new types of displays or modifying existing ones can be done with minimal impact on the core game logic. For example, integrating a new web-based interface would involve creating a new observer class that subscribes to the **Board** updates and implements the necessary rendering logic for a web environment. The rest of the program remains untouched, which demonstrates the flexibility and extensibility of our design. This modular approach also supports changes in the game rules or board dimensions, as the display logic operates on abstract interfaces and general operations rather than specific, hardcoded values. The use of polymorphism allows for dynamic behaviour changes at runtime, allowing the system to adapt to display preferences or configurations. Furthermore, the display design ensures that new features such as additional game modes or enhanced graphical effects can be incorporated without significant rewrites. This adaptability ensures that the display system can evolve alongside the game, which supports future developments and improvements while maintaining the integrity and functionality of our program.

Project Specification Questions

Q1:

A trie would be an optimal data structure to implement a book of openings since it represents the branching nature of different chess openings and allows for efficient lookups using prefix matching. In the trie, each node represents a specific arrangement of the board and will also contain a field of whether ending at the specific node results in an opening. The root node is the original board setup, and subsequent nodes contain the state of the board after a specific move has been performed. Edges are represented by the type of piece and its starting/ending position after a move, transitioning the board from one state to another. For instance, the edge labelled "Pe2e4" would lead from the root node to a child node representing the board state after White's king pawn has moved to e4. From this node, we might have multiple edges representing

Black's possible responses, such as "Pe7e5" or "Pc7c5", leading to nodes that represent the Ruy Lopez and Sicilian Defense openings, respectively.

Q2:

We would use a stack data structure to store a history of moves currently made. Whenever a player makes a move, information about the move will be pushed onto the stack. The information will describe the starting and ending positions of the piece that was moved, the information of the piece that was captured (if it exists), and other additional data such as whether the opponent is in check, etc. The way we would implement this is to create a `PlayerMove` class which would represent the elements on the history stack, which would be a part of the `Game` object. The `PlayerMove` class will store the information mentioned above. If a `Player` undoes a move, the `Game` will pop the top `PlayerMove` off the stack and use its information to restore the `Board` state. This will allow a player to undo an unlimited number of moves. If we will only allow one undo, we can use a variable to keep track of the last move made (still using a `PlayerMove` class) which we can reference in the case of an undo.

Q3:

Firstly, the board will need to expand in size, that is, extend each side of the standard 8x8 chessboard with 3 additional rows of 8 cells. Similarly, the text and graphical display will need to be changed slightly in order to display the larger board. Secondly, the game logic will need to be changed slightly. Two more player and score variables will be needed for the additional two players. Furthermore, the loop in the `Game` class will need to be altered slightly to cycle through the two additional players. We will also have to add more (enum) types to represent the additional colours. Thirdly, the algorithms for the computer agents will need to change. Since the four-handed chess board is different, the algorithms will need to be adapted to the new board. The way the interpreter parses the positions will also need to change. Since we can longer use the standard way to specify tiles on a board, we need to be able to parse a new positioning scheme.

The rules for how pawns move will have to change as well. Each pawn will need to have a `direction` property describing which direction it faces, which will determine how it moves. That is, the pawns of each player move in a different cardinal direction (up, down, left, right).

Extra Credit Features

In terms of extra credit features, we were able to implement the entire project by only using smart pointers. As a result, we did not have to explicitly manage our own memory, making for an easier and more straightforward development process while also ensuring that our program does not leak any memory. For the most part, avoiding raw pointers and using smart pointers instead was not too much of a challenge, as they were just simple replacements. However, our misunderstandings in the usage of smart pointers led us to some challenges when trying to adapt our Observer pattern to using smart pointers. We did not realise that the raw pointers being used in the Observer pattern did not express ownership, and so when we

attempted to make the switch, we faced a myriad of confusing problems. This resulted in a bit of a time sink researching things like extending our classes with `enable_shared_from_this()` and trying other unnecessary tactics to make it work. However, after researching and talking with classmates, we were informed that this switch was not necessary, and so we reverted to just using raw pointers in our Observer pattern.

Final Questions

Q1:

One lesson that this project taught us was how important constant communication is when it comes to developing in a team. Starting out, we all decided to independently work on our components, but quickly realised that there needed to be consistent discussion over how our components should work together. Ultimately, there has to be some level of coupling in our program, and so we had to work together to find a compromise on how different components talked to one another. While working on the project, we scheduled daily standup meetings to talk about what we accomplished, and what we needed help on. This was also helpful because some members' work depended on the completion of other members, and so it was nice to know where others were at with their work so that people could manage expectations and time properly.

Additionally, we also learned how valuable collaborative problem-solving can be. Whenever we were stuck on a design or implementation choice, we'd often discuss the problem and bounce ideas off of one another, which greatly aided in our development process. Whenever someone would encounter a bug, we would also try to help out and locate the source. This collective effort often led to clean and effective solutions that would have taken only one person much longer to come up on their own.

Q2:

If we were to start over, we would aim for a more consistent application of design patterns throughout our project. While we successfully implemented the Observer pattern for our display system and the Strategy pattern for our computer players, we may have missed other opportunities to leverage design patterns. A thorough exploration of design patterns during our initial planning phase could have led to an even more flexible and maintainable codebase. We would also invest more time upfront in understanding smart pointers. Our challenges with smart pointers, especially in the context of the Observer pattern, resulted in significant time loss. A deeper initial dive into smart pointer usage and best practices could have saved us time later.

Additionally, a more comprehensive testing strategy would have been beneficial. Implementing a robust testing framework from the outset, possibly using test-driven development, could have helped us catch and resolve issues earlier. Lastly, we would consider potential extensions, such as the four-handed chess variant, earlier in our design process. Although our current design is generally flexible, incorporating these considerations from the start could have resulted in an even more adaptable core structure. This proactive approach could save time and effort if we decide to implement such extensions in the future.