Chess Engine Design Documentation

Table of Contents

Section 1. Introduction

In this project we implemented a chess game engine that supports the instantiations of many different types of open-information two-player board games, for example Chess and Shogi[1]. We start with a proper motivation for what comes to our program design. Let's think about what are the main components of a Chess-like game, and what abstract concepts we may derive.

- The board. A typical game happens on a rectangular-sized board, with varying sizes for each type of game. A subset of the squares can be occupied by pieces. At any given time, only one piece can occupy a square. A game on the board typically starts with the full set of pieces in a fixed initial pattern.

- The pieces. Each piece has a unique identifier, and a set of rules that the piece is allowed to follow in order to act on the board. Each player starts with an equal set of pieces, and the initial positions of the pieces are always in a symmetric pattern. In any situation, one piece can occupy one square only; friendly pieces, in almost all situations, can only capture the opponent's pieces by taking over their position on the board, except Pawn's En Passant in Chess. Each player owns a dominant piece (the King), where the player must protect their King from getting captured by their opponent. The game ends when a King gets captured, or is in a situation that a capturing is unavoidable (a checkmate). There is also the concept of promotion, although specific rules vary between types of games.

- The moves. This is a concept very closely related to pieces, but still not quite the same. In essence, a player always makes a move on their turn, by changing the location of a piece of their choice, and applying any capturing or promotions. Some moves are more sophisticated than others. A King in Chess has the opportunity to perform a Castle with a Rook, and a Pawn may perform En Passant in addition to their usual capture moves. In Shogi, a player can turn captured pieces into their own and re-deploy them onto the board, of course under certain limitations of the game[2].

[1] See https://en.wikipedia.org/wiki/Shogi for an introduction to the rules of Shogi.
[2] Similar to how CrazyHouse is played as a variant of Chess. https://en.wikipedia.org/wiki/Crazyhouse

Essentially, what we have understood about the relations between these three components is that a board must be in control of the pieces and their locations, a piece must have a distinct set of rules that can be played, and a move must follow those rules to change the state of the board. We want to translate these high-level relations into class diagrams that can be implemented in C++ to reflect the ownership and association of each component of the game.

Some house-keeping rules: In our program there are a few customs we followed. For one, we use `std::string` to represent a location on the board e.g. for Chess, `a1` represents the lower-left corner and `h8` represents the up-right corner. Second, we use ascii characters `W` and `B` to refer to which player is taking their turns, and we will define them as `WHITE` and `BLACK` macro variables. Third, we use `char` to represent each piece's name. For example, `K` is used to represent a King, and `Q` a Queen, etc. In our text-based UI we use black and white foreground colors to represent the team of each piece, instead of distinguishing them with lower or upper cases, while the latter is still needed for handling setup mode.

Section 2. UML Overview

Although we change a few implementation bits of the UML diagram from before, the overall structure that underlines the relations discussed above is still the same. And our previous document contains a high-level description of the main classes that we are implementing here. Now we want to clarify on their relationships, and how we managed to use C++ and OOP design principles to put all that into action.

We decided that a `Board` class is going to handle the state of the game, in particular which pieces of whom are where[3]. That's why we made the `Piece` objects owned by the `Board` through `std::unique_ptr`. And we provide a public interface method `const std::unique_ptr<Piece>& Board::operator[](const std::string&)` that provides read-only access to the pieces. For a use case example, we introduced a new class in our second version of UML called `Vision`. This is a helper class that provides the `Player` class with a "vision map" on the game board, indicating which squares the player's pieces can "see". Its public method `void Vision::RefreshVision(Board*, char)` takes a `Board` and a player (`WHITE` or `BLACK`) and refreshes that player's vision. Basically it reads every piece on the `Board` that belongs to the player, and loops through all valid moves to update a vision counter[4].

We moved quite a few methods in our first draft of UML under `Board` to `Piece`. For example, we used to have `bool Board::IsEnPassant(const std::string&, const std::string&)`. After we introduced the read operator `[]`, we moved this method inside `Pawn` class and let itself make the decision by reading the `Board`. Another similar one is `bool Board::IsCastling(const std::string&, const std::string&)`, which is moved inside `King`. And `bool Board::ValidMove(const std::string&,const std::string&)` is renamed and moved inside `Piece`. These tiny improvements are a few examples that we have made to improve the overall low coupling-high cohesion design of the program.

An intuitive way to understand how we distinguish what's handled by `Board` and what's handled `Piece` and its many subclasses is whether we need to invoke the no-hidden-information property of a chess

---

[3] See `src/board.h` for detailed interfaces of this class.
[4] See `src/player/vision.cc` for details.

game. If there's something I need to know from my opponent, then the requests are handled by the `Board`. Otherwise, all internal states and decisions belonging to one player (like telling if a move is valid, or maintaining visions on the board) is managed by `Piece` or `Player`.

For example, the `Board` provides a very important and useful method `bool Board::IsRevealingKing(Piece*, const std::string&)`. Everytime a piece wants to make a move, it has to verify that such a move won't reveal an attack opportunity on its own King. To implement this method, we need to know some information about our opponent, say their vision on the board. On the other hand, the `Piece` manages methods like `bool Piece::CanMove(const std::string&)` that tells a piece if any moves can be executed by reading the board and invokes the above mentioned method[5].

So much for the `Board` and `Piece` relations. One more thing that needs some clarification is why we chose to give `Piece` an `Iterator`. It provides an **Iterator** interface to a `Piece` object, by wrapping around a quasi-abstract class `PieceIterator`, which has polymorphic implementations `SlideIterator` and `JumpIterator` that abstracts the two main move patterns in Chess and Shogi[6]. An `Iterator` loops through all squares that the current standing piece can see and move to, so as to enforce its move patterns. In our first UML, we created a separate `PieceIterator` for each piece in Chess, but later we felt like it's too redundant, and abstracted them into the two polymorphic subclasses above. Another benefit of this abstraction is that it allows an easy way to define any pieces we want, like the ones in Shogi, without explicitly constructing a `PieceIterator` for each new piece. We will bring this back again in a later section.

An `AbstractMove` controls the dynamics of the game (moving, capturing, promotion, castling, dropping, etc). It provides two interfaces `AbstractMove::MakeMoveOn(Board*)` and `AbstractMove::Undo()`, with different polymorphic implementations like `Move`, `Promotion`, and `Drop`[7]. We also made a separate subclass `KingMove` to handle castling in Chess, since it's the only case where two pieces were involved in one move. In our first UML draft we made `Move` a superclass and made morphic forms under `KingMove` and `PawnMove`. In our final draft we added an abstract layer `AbstractMove` to better capture the essence of what defines a move, and separately implemented its subclasses.

A `Player` controls the action on behalf of a player, be it human or computer. Parsing command line input is handled by `Player::TakeAction()`, and the execution step is managed by `Player::MakeMove()`. For `HumanPlayer`, the latter method wraps the command line input to an `AbstractMove` object, and hands it over to `Board` for execution (a more detailed description on this in next paragraph)[8]. For `ComputerPlayer`, it first makes some deliberate calculations and suggests a next move. The design of `ComputerPlayer::MakeMove()` is worth some mentioning: at first thought we wanted to implement more advanced computer levels as Decorators, with `ComputerLevel1` as a base class. However, this only brings more troubles to the downstream user (in our case, it's the `Board` who

[5] See `src/pieces/piece.cc` and `src/board.cc` for implementation details.
[6] See header files under `src/pieces/iterators`.
[7] See `src/moves/move.cc` at how a basic case is implemented.
[8] See `src/player/human_player.cc`.

creates the players). And we have no intent to let a Level 2 player make a move before a Level 4 player. So we abandoned the idea of a Decorator pattern, instead used Delegation. Now the `ComputerPlayer` acts like a mediator between the `Board` and the actual computer players (`ComputerLevel1…4`)[9]. It also enables the downstream users to construct a computer player with a one-liner[10].

Here comes the mechanisms behind a real-time game: upon startup, `void Board::AddHumanPlayer(char)` or `void Board::AddComputerPlayer(char, int)` is/are called, and the `Board` creates two `Player`s. Then the controller will keep calling `void Board::PlayerMakeMove()`, and the `Board` will call `Player::TakeAction()` to the player in turn. The `Player` will then take a command line input, maybe throw some errors on the way, and `Player::MakeMove()` is called. It parses an `AbstractMove` object, and calls `Board::MakeMove(std::unique_ptr<AbstractMove>)` to execute the move. The `Board` then will handle the checks/checkmates/stalemates after the move is executed successfully. But before that, the board will call `AbstractMove::MakeMoveOn(this)` to accept the move. After the move has taken effect, the `Board` will absorb and own the `AbstractMove` object, by storing it in a `std:stack`. That way, if `Player::TakeAction()` decides to undo a move, `Board::Undo()` will be called, and `AbstractMove::Undo()` is executed by the object on top of the stack. After successful execution of undo, the `Board` will simply pop the stack, and finish the undo sequence. Finally, the `Board` will swap the players if `Player::TakeAction()` is successful.

Section 3. Design Principles

To begin with, we used the **Observer** pattern to handle the `Board` and `TextUI/GraphicsUI`, in the same manner as in Assignment 4. This ensures that our UI design is separate from the game mechanics, while ensuring that the UI is following real-time changes on the board no matter what is happening to the game.

For the `Board-Player-AbstractMove` relations described above, we implemented the **Command** pattern, where the `Player` acts as an Invoker, the `AbstractMove` as a Command, `Move` and other subclasses acts as a concrete implementation of a Command, which acts on the Receiver, in our case the `Board`. It ensures that the `Player` class can be implemented in a very clean and extensible fashion, by separating the decision process (handled by `Player`) from the execution process (handled by `AbstractMove`).[11] It also allows that once the player has figured out a move, it doesn't need to understand how the move is executed, by trusting that if they have parsed the correct implementation of an `AbstractMove`, it will take care of everything happening to the `Board`.

To implement the `ComputerPlayer` class, we used a **Delegation** pattern following the **Pimpl idiom**, by transferring responsibilities of a `ComputerPlayer` to `ComputerLevel1..4`, which takes the wheels behind the scene. For one, this allows the downstream user to have a relatively clean interface when creating and interacting with a `ComputerPlayer` object, without too much worrying about how each level

---

[9] See `src/player/computer_player.cc`.
[10] See `src/board.cc:141`.
[11] See `void Human::MakeMove()` for how a basic decision is implemented, and `void ComputerLevel1..3::MakeMove()` for a more sophisticated process.

of difficulty is implemented underneath. Second, it allows resource sharing among the implementation classes, by making them friends of the `ComputerPlayer` class. In some sense, the `ComputerPlayer` acts like a supervisor. It creates a workflow for `ComputerLevel1..4`, makes them work together to complete a task, meanwhile without interfering with how they actually work. And `ComputerLevel1..4` can then focus on completing their task, while staying clean of the details of interacting with stakeholders.

Last but not least, we used the **Iterator** pattern for `Piece` to faciliate high extensibility. We have also automated the creation process of a piece through template programming. We will describe in detail how this design pattern helps us to extend to arbitrary piece moving rules from Chess to Shogi, and potentially more games.

Section 4. Resilience to Change

We now describe how we extend Chess pieces to Shogi pieces through our implementation of `Iterator` and its implementation class `PieceIterator`. Recall that `Piece` uses an `Iterator` to loop through all potential valid moves on the board. The `Iterator` class maintains the interface inside `Piece`, while delegating `PieceIterator` with the actual work, another use case of **Pimpl Idiom**. To help motivate this, a piece in either Chess or Shogi can move by either sliding through a straight line, or jumping to nearby squares[12]. So we created two polymorphic templates `SlideIterator` and `JumpIterator` under `PieceIterator` to abstract these two types of behaviors. Upon construction, they take in an array of directions, with the length as a template variable[13]. The two templates differ only in how the method `void PieceIterator::operator++()` is implemented. To create a piece and enforce its moves, we simply construct either a `SlideIterator` or `JumpIterator` and pass it an array of valid move directions[14]. Moreover, instead of stopping at wrapping just one `PieceIterator` inside `Iterator`, we can wrap two or even more. This helps us to implement more sophisticated pieces with ease, for example, the Dragon King in Shogi[15]. With a few other tweaks on the `Piece` class, this essentially explains how we could create arbitrary pieces at speed, adapting from Chess to Shogi.

Another design feature is how `AbstractMove` and its many polymorphic subclasses essentially encapsulated the dynamics of what every chess-like game needs. Together with `Iterator`, these two components allowed board adaptability of downstream users to a specific set of game rules. As a fun fact, we can run computer players that are initially created for Chess to also play a Shogi game, since the two games share a lot of elements dynamic-wise, other than that Shogi supports Drop, which the computer players have not learnt yet.

To implement a fully functional Shogi playboard, we simply override a few mutator/accessor classes, and the most sophisticated methods like `Board::IsRevealingKing()` are shared with Chess. This adds to the fact that our Chess computer players can easily manipulate the game, no matter whether they're playing Chess or Shogi.

---

[12] In both scenarios, a jump is always un-intervenable. But in some other games, like Chinese Chess, some pieces can block other pieces from jumping over it.
[13] See for example `src/pieces/iterators/jump_iterator.h`.
[14] See for example `src/pieces/queen.h`.
[15] See `src/shogi/pieces/dragon.h`. In short, a Dragon combines the King and the Rook.

Section 5. Answers to Questions

Question 1: 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.

An opening move can be thought of as a sequence of concrete `AbstractMove` objects. So we may have a class `Openings`, which stores a few opening moves in linked lists, and upon construction merges the lists into a tree. It may support a method `Openings::MoveTaken(AbstractMove)` and `Openings::SuggestMove()`, the former takes a current move by a player and compares it with all child nodes of the current tree node, moves down the branches if matched, and the latter returns a random child node of the current node.

Question 2: 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 has been answered in detail above when we describe `AbstractMove` as well as its various polymorphic subclasses. We discussed how we implemented the `Undo()` for both `AbstractMove` and `Board`, when and how an undo is executed, and what data structure we used to support unlimited undos.

Question 3: 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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)

If we ever wanted to implement 4-player chess, the first thing we need is to add more players on the board, which means we need to modify how the `Board` manages the turns. Currently the `Board` has two private variables `player` and `opponent` which take the value `WHITE` or `BLACK` to indicate who is the current player. In 4-player chess, we may need two more colors `RED` and `GREEN`, and put player indicators in a `std::vector` so we can loop through players instead of performing `std::swap`.

Additionally, we already implemented moving directions in `PieceIterator` class, to control the direction a player's piece is facing. It's very quick to extend it to 4 directions. With this in mind, we may also modify `King::IsCastling` and `Pawn::IsEnPassant` to take consideration of directional changes.

Section 6. Extra Credit Features

For one, we already discussed how we made a Chess program into a chess engine to adapt to various board games like Shogi.

Another thing we did in our program is to use C++ smart pointers wherever we can, whenever we can. So far through our test runs we haven't seen a memory leak yet, and there is no explicit `delete` in the entire program. The caveat here is to properly design our class ownerships. For example, `Move` and `Board` both may own `Piece` at run-time, but not at the same time, so we used `std::unique_ptr<Piece>` in both classes[16].

We implemented `Undo` not only because it's an extra feature but as an organic component of the engine itself. The way we thought about this is for computer players to determine a move (and many other use cases), it's the best if we can actually execute a move, see what it brings to us, and undo it if necessary. Especially in `ComputerLevel4::MakeMove()` where we implemented an alpha-beta search algorithm, it is crucial for the alpha-beta searchers to apply a move before passing down the search tree, and undo the move when back-tracing[17]. In terms of implementation challenges, once we figured out the interaction of `Move` with `Board`, it boils down to moving `std::unqiue_ptr<Piece>` objects back and forth to handle capturing/undo capturing/promotion/undo promotion, etc. This also shows a good program structure design should be prioritized over code implementation, because the former sometimes determines how easily the latter can be done.

Section 7. Final Remarks

One thing we learned is that a good SOLID design triumphs over a clever implementation, especially at the early stage of developing a large program. When we started the project, we didn't immediately come up with all the design patterns and class interactions. Instead, we started with a simple Chess game with very basic rules, and iterated several times to make the code more adaptive and robust to change. The current stage is just a slice of its evolution, and we still feel like more could be done.

Section 8. Reference

Here is a presentation of the project structure, the reader may refer to it while reading this documentation.

```
src
├── board.h
├── controller.h
├── game.h
├── main.cc
├── moves
│   ├── abstract_move.h
│   ├── drop.h
│   ├── king_move.h
│   ├── move.h
│   └── promotion.h
├── pieces
│   ├── bishop.h
│   ├── iterators
│   │   ├── iterator.h
│   │   ├── jump_iterator.h
│   │   ├── pawn_iterator.h
│   │   └── slide_iterator.h
```

[16] Or we could have used `std::shared_ptr` everywhere to save some brain juice in sacrifice of a tiny bit of performance.

[17] See `src/player/computer_level_4.cc` for details.

```
│   ├── king.h
│   ├── knight.h
│   ├── pawn.h
│   ├── piece.h
│   ├── queen.h
│   └── rook.h
├── player
│   ├── computer_level_1.h
│   ├── computer_level_2.h
│   ├── computer_level_3.h
│   ├── computer_level_4.h
│   ├── computer_player.h
│   ├── human_player.h
│   ├── parser.h
│   ├── player.h
│   └── vision.h
├── shogi
│   ├── pieces
│   │   ├── bishop.h
│   │   ├── dragon.h
│   │   ├── gold.h
│   │   ├── horse.h
│   │   ├── king.h
│   │   ├── knight.h
│   │   ├── lance.h
│   │   ├── pawn.h
│   │   ├── rook.h
│   │   ├── shogi_piece.h
│   │   └── silver.h
│   ├── shogi_board.h
│   └── shogi_player.h
├── subject.h
├── chessboard.h
├── observer.h
├── text_ui.h
├── graphics_ui.h
├── exceptions.h
└── window.h
```