Roger Zhang, Shriya Kulkarni, Yuri Na

CS246 A5 Design Document - Chess

Overview

(describe the overall structure of your project)

- Each game that a user plays with a certain setup of the board and choice of pieces is a 'game instance'. The main() function in our program supports multiple game instances, till EOF is reached.
- Each gameInstance() function creates a ChessGame that contains the user's chosen configuration of the board (customized by giving command "setup", else the default standard board). It also creates two vectors to store the pointers to pieces that the user creates. Since these objects are within the scope of gameInstance(), they can be used for playing a round of chess, after which they are deleted, allowing for a fresh board and set of pieces for the next round.
- The command interpreter in our program is divided into 3 distinct parts, based on the commands it needs to support within its scope. gameInstance() allows a user to set up their board and begin a game, setup() allows them to customize the round they will be playing, and gameRun offers them the commands "move" and "resign" which actually change the state of the board as part of a round.
- Within setup, pieces are created and pushed into the piece vectors for black and white players. They are also set on the tiles of the board. In case no setup is done prior to giving the command "game", defaultSetup() is executed.
- Once the "game" command is given, 2 observer objects are constructed based on the player's choice (out of 4 possibilities - a human and 3 computer levels, each with access to its pieces). The ChessGame object and the two Observer objects (players) are then sent to gameRun which carries out the actual game by switching between the white and black players' turns (by executing the pickMove() method of each player).
- Together, gameRun() & pickMove() keep a round of the game running, till a stalemate or checkmate is encountered 9or a player resigns).
- Throughout this, castling & en passant has been accounted for and legal moves for each piece are kept track of so no wrong moves can be played by any player.
- Eliminated pieces just have their isCaptured field changed to 1 so they cannot be played again, nor can they be captured again.
- Once a checkmate or stalemate is encountered, or a player resigns, the winner is returned to gameInstance and then main to update the overall score. This process repeats till EOF.
- Then the final scoreboard is then printed to indicate the end of the program.

Roger Zhang, Shriya Kulkarni, Yuri Na

Design

(describe the specific techniques you used to solve the various design challenges in the project)

1. **Challenge: We needed a way to have a game with its own conditions and a board to exist. However, the board also needed to be affected by the moves of other elements like the game pieces, caused by the players.**

   For this, we used the observer pattern, where our subject is a ChessGame class, and its observers included Human and Computer. This allowed us to keep track of attributes pertaining to a game like a field which indicates whether the white king piece is in check. So that when each game is executed we can have an instance of a ChessGame which holds its information. Then every time the ChessGame changes its state, all the concrete observers (Human and Computer) will be notified. This then allows players to view the state of the game and interact with it.

   Challenge: There were numerous separate but highly related elements we wanted to include, such that their classes would have effectively the same fields but different behaviors in their methods.

   To enable this, we made use of inheritance. For instance, 6 classes (Pawn, King, Queen, Rook, Bishop, Knight) inherit from their parent class 'Piece'. Within the observers, Human and Computer and the levels within the latter (Level1, Level2 and Level3) are dependent on one of its fields. This also allowed us to reduce the splitting of our command interpreters as we did not have to account for possibilities dependent on the player through conditional statements, rather, the difference in each player's methods determined the gameflow.

2. **Challenge: With so much to work with, we needed to ensure that no fields that we did not intend to change would be changed in error. The high number of elements also meant that the program could possibly be slowed down by objects that needed to be altered.**

   To ensure this, we kept as few public fields as possible and created accessor methods such as 'getPlayerTurn' and 'getLegalMoves'. We also implemented methods to set fields/switch them and update them as needed, instead of directly accessing them. As for addressing the potential issue with speed and memory, we decided to use pointers and references to objects instead of passing any piece or board by value.

3. **Challenge: Since the command interpreter had to be able to take in 8 commands in total and each process that followed the commands was long and intensive, we needed a way to keep track of their effects in a convenient manner that would be easy to parse.**

   Here, we adopted principles from modular programming and split our implementation into 3 major parts for the command interpreter, depending on the scope of the commands. The individual files made finding errors easier too.

Roger Zhang, Shriya Kulkarni, Yuri Na

Resilience to Change

(describe how your design supports the possibility of various changes to the program specification)

Our design makes it convenient to add or remove any required components as per program specifications. Some examples are as follows:

1. The size and shape of the board can be changed with minor changes to buildBoard, namely by adjusting the limits of its loops.
2. Additional levels of computer or levels of expertise of humans (beginner/intermediate/expert) can be conveniently added by just implementing different pickMove() methods for them.
3. Since no captured pieces are deleted before the end of the game, in the situation that any have to be reinstated (such as while enabling the facility to undo a move), no new pieces need to be created, only a status and coordinate change is required.
4. Due to the separation of gameRun() and pickMove(), one can easily add extra chances for any of the players in question.
5. Since both players maintain a vector of each other's pieces apart from their own, any specification that uses the other player's future moves to determine one's move, this can be easily implemented. Such a process can be used to make a computer smarter, by letting it foresee the other's most likely next move.
6. It is convenient to remove conditions to allow for more flexibility (disallowing castling or allowing a set up to be such that a king is already in check).

<u>Answers to Questions</u>

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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Assume that we're taking the default setup as the condition under which our game starts off. We can then implement a computer vs computer game, and record the first dozen or so of the moves the two players make. This would be a set of accepted opening moves and responses to opponent's moves. We know these are accepted/acceptable, because after setup and after each move, the vector of legal moves that can be played is updated and a computer's moves are restricted to this vector.

To make a whole book of the same, this process can be repeated multiple times - possibly with different levels of computers (which would alter the complexity of opening move sequences). Due to the randomness of moves, the chance of an opening sequence being repeated is negligible, but one could still check that the moves are indeed different.

To make the most of this process, all combinations of levels can be tried out, and we could also extend this to allowing a human to play against a computer or two humans to play against each other and record those moves.

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 can be done by implementing a stack (with the type std::vector<Record>) of previous moves, say ***history*** - each element is a Record object, where Record is a class with the fields:

- Box movedFrom
- Box movedTo
- Piece * pieceMoved
- Piece * pieceCaptured (set to nullptr if none).

For each move played, a Record object is created and pushed to history. To undo a move, the topmost element from the stack will be taken (***history***[(***history***.size() - 1]) and an algorithm equivalent to executing the command "move <movedTo> <movedFrom>" will be played (i.e. the opposite of the move recorded).

If any piece was captured, it will be reinstated at the box 'movedTo', and the isCaptured field of the piece will change back to 0 with a switchCapture() method that will need to be created (though no new pieces need to be created since captured pieces were never deleted). At the end of changing the board's state with these counter-moves, the we execute ***history***.pop_back();

The same thing happens for any number of undos - the topmost element of history is considered, a counter-move is made based on its fields on the current state of the board (already updated with any previous undos). The element is then popped.

Moreover, since after every move, legal moves are updated, just playing the opposite move will suffice and no special change will have to be made to the state of other pieces that will be affected by undoing a move (the present program takes care of that).

Question: 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.

1. The very structure of the gameboard will change - In four-handed chess, a special board is used so the structure of the gameboard itself will be changed to have 160 tiles (3 more ranks on each side of a standard chess board - in our program, this can be done by changing the limits of buildBoard appropriately). This will also change the colors of the pieces (adding 2 more vectors with Piece * elements for 2 more players) and the effect of the 'setup' command, which will now require all queens to be on black tiles in addition to the conditions to have all kings on the board and none in check.

2. The program will have to follow a playing order - Players will make their moves in a clockwise direction so a list of players will be maintained in that order through which an indicator will iterate repeatedly to determine whose turn it is. It will have to be set beforehand but not as a constant, so it can change as players get eliminated from the game through checkmates and stalemates. This also affects playerTurn field and switchPlayerTurn() - the latter will be changed to setToNextPlayer().

3. A points tracker will have to be implemented for each instance of a game that contributes to the final scoreboard, since four-player chess tracks points that may count towards determining a winner. For instance, checkmating an opponent gives a player 10 points, while stalemating them gives 10. An indicator (isActive = 1 by default) will be introduced as a field in each observer, and it will be switched to 0 as a player goes out of play. Counting of points will be conditional on isActive being 1 so points will not be counted for capturing an eliminated player's pieces, in accordance with the rules of the game. All of these scoring rules will have to be accommodated.

4. The game ends in different situations: In four-player chess, stalemating leads to elimination of a player and not the end of a game, unless there are only two players left.

Finally, depending on the guide being followed to implement four-player chess, changes in rules may be required, such as defaulting pawn promotion to queen, or disallowing en passant. In such cases, we would only have to update the program with very minor eliminations like removing the functionality to allow an en passant (status of -1) in Piece::updateLegalMoves() or the functionality for the user to choose what they wish to update their pawn to in Human::pawnPromote().

Roger Zhang, Shriya Kulkarni, Yuri Na

<u>Final Questions</u>

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?

- Communication is of utmost importance. The way we ensured this was by heavily commenting on our code and by describing changes in branches on GitHub. Moreover, we always got the other team members to look at our pull requests before merging anything. This ensured that everybody knew about any new developments and changes to the program that would later affect their part. It also ensured that errors were caught earlier than otherwise.
- The UML a project starts off with is extremely important too. It determines the journey of each individual in the team and serves as a common reference. Inconsistencies in a UML will cause numerous structural errors, leading to a need to overhaul it completely.
- Multiple people in the team should be responsible for each part of the program, such that if one of them is not available to work on it, the other can.
- Everybody's schedules and times (and the lack of available time) should be accounted for early on, and certain hours in which everyone can work on the project should be set.

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

- Definitely thought more about the UML we started off with. We believe that if more work was put into that, we would have an easier time later, with fewer overhauls of the structure.
- Started testing earlier and more rigorously so errors could be found sooner and debugging could be made a less intimidating process.
- Kept a more detailed record of the changes that were made at each stage and briefed each other regarding new developments more frequently so anyone could work on any part of the code, even if they did not lay down the foundation for it.
- Started coding earlier with more stringent deadlines, leaving more time for finishing touches and debugging.