图示

描述已自动生成

Figure 1 the Unified Modeling Language (UML) figure of this design

● how many contracts your design has (if multiple) and what each does

**Answer:**

As shown in the fig.1, currently there are 11 contracts in this design which are AbstractGame, Chess, Board, Square, Piece, King, Queen, Bishop, Rook, Knight, and Pawn. There responsibility are listed below:

**AbstractGame:** Actually, this contract just defines the basic behaviors of a two-players, adversarial, round-based game. It contains the typical method stubs that this kind of game may need, such as the initialization function, the join function, the quit function and so no. The specific API could be seen in the figure.

It is worth to mention that this design follows the pattern of “challenge and response”. It brings about two advantages. Firstly, since the specific rule of this kind of games could be quite strange, so it is almost impossible to abstract a function to realize the check of whether certain player has won the game or not. Following this pattern takes us away from truly define a function to implements the check. Secondly, since we are designing the game on chain, we need to take the cost into consideration. If we did declare a function to conduct the calculation of for instance, whether this move of player A did approach “checkmate”, this would take lots of gas. What is worse is that, if we did follow the move-and-check pattern, we need to do this calculation after every move. From everyone’s point of view, this is not worthwhile and would scare our user away from this game.

Thus, to tackle this problem, it would be better that we switch to the pattern of “challenge and response”. It means that, we ask the person who thought that he/she is going to win to claim that they are going to win, and then, let’s leave the chance to the opposite side. If they could make a valid move or other valid action within a limited time, if means that this claim is invalid. Thus, they can continue the game.

**Chess:** This contract implements the AbstractGame contract and is the core contract in this design, it defines all the logistic needed in the chess game and it takes the responsibility of interacting with our users.

**Board:** This contract is used to mimic the checkerboard in the real check game, it contains an 8x8 array and would be used to hold all the squares.

**Square:** This contract is used to describe each cell on the board, it would be used to hold an object of Piece.

**Piece:** This contract is the used to define the interface of all kinds of pieces may need. The subsequent six kinds of pieces are all the subclass of Piece, they defines the features of each kind of Piece and overrides the method in Piece.

**King:** This contract defines the feature of the piece of “King” and defines the special move “Castling”.

**Queen:** defines the feature of “Queen”.

**Bishop:** defines the feature of “Bishop”.

**Knight:** defines the feature of “Knight”.

**Rock:** defines the feature of “Rock”.

**Pawn:** defines the feature of “Pawn”.

● what custom data structures each contract should define (and what each does)

**Answer:**

Several custom data structures are defined in these contracts, the specific introduction is listed below:

**AbstractGame:** This contract defines three enumeration structures to describe the different states and results in this kind of game. The first one GameState is used to demonstrate that currently the game is at which stage. It contains “JoinOne”,”GuessFirst”,”Running”,”TimeOut” and “GameOver”, these five constant values. Their usages are clear with the name. “JoinOne” describes that one player has joined the game, and he/she is waiting for the second player. “GuessFirst” means that there has been enough players and they are playing a little game to decide which guy play first with the white piece. To realize the fairness in the playing order, I decided to include matching pennis game into the guess first stage, details about this issue would be introduced in the subsequent question. Besides, “Running ” describes the most typical state of this game, at this stage, players are supposed to move their pieces round by round and try their best to approach the so-called “checkmate” to win the game. However, to keep the fairness, we need to pay attention that some player may deliberately leave the game when they find that they are going to lose. A limitation to the time of each round and each request is needed to be set. “TimeOut” is the symbol that use to represent this situation. No matter for which reasons, one side of player run out of his/her time, the opposite is able to win the game directly. Then, in that case, the game would switch to the next state, “GameOver”.

The second enumeration defined in this contract is “Turn” used to describe which turn is it in this round, i.e., which side should act.

The third enumeration is “Result”, it defines three kinds of states which are “White side win”, ”Black side win” and “Draw” respectively.

**Chess**: This contract defines a custom struct named “Player” containing all the relevant information about one player, such as their address, name, balance, and color of pieces. Currently, the member in this struct is just a rough version, it can be added according to specific goals and designs. For instance, if we want to record all the player ever joined, and rank them with their scores, it would be better that we add a new member named “score”, and add a mapping variable, mapping (address => player) to keep this information into the Chess contract.

● the *public* API of your contract(s), including all functions/variables/events

**Answer:**

All the API in each contract has been shown in the Fig.1, here I would detailly explain the goal and function of these methods/variables/events.

**AbstractGame:**

1. // used to initialize the game, turn the State to JOIN\_ONE
2. function init() public payable;
3. // used for second player to join the game, turn the State to GUESS\_FIRST
4. function join() public payable;
5. // used to quit the waitting for the first joined player
6. function quit() public;
7. // used to end each turn and switch between players
8. function endThisRound() public;
9. // used for player to give up
10. function surrender() public;
11. /\* used for player to claim that they are winner, if the opponent could
12. not take valid action to prove they still have solution, the player
13. who claimed would win.
14. \*/
15. function claimWin() public;
16. // used for claim that time is over, the opponent player lose.
17. function claimTimeOut() public;
18. // offer draw to the opposite player
19. function offerDraw() public;
20. // answer whether accept the draw
21. function acceptDraw(bool isAccepted) public;
22. // used to withdraw ether from the contract
23. function withdraw() public;
24. // record the join of player
25. event PlayerJoined(address \_addr);
26. // record the quit of player
27. event PlayerQuit(address \_addr);
28. // record the address of surrender
29. event Surrender(address surrender\_addr);
30. // record the result
31. event Result(Result result);
32. // record the address of recipient
33. event Withdrawal (address recipient);

**Chess:**

1. contract Chess is AbstractGame{
2. struct Player {
3. address addr;
4. uint256 balance;
5. string nickName;
6. bool isWhite;
7. }
8. Board public board;
9. State public gameState;
10. Player private playerA; // First player
11. Player private playerB; // Second player
12. Turn public currentTurn;
13. Turn public nextTurn;
14. // record the timestamp updated last time
15. uint public lastUpdatedTime;
16. // a limitation to each turn/response
17. uint public timeLimit;
18. // used to decide which player takes whilte
19. function commit(bytes32 commitment) public;
20. // combined with commit function
21. function reveal(string calldata committed) public;
22. // used to realize the move of piece from A point to B point with its x,y index
23. function move(uint8 from\_x, uint8 from\_y, uint8 to\_x, uint8 to\_y) public;
24. // each move would emit this event as a record in log
25. event Move(address player\_addr,uint8 from\_x, uint8 from\_y, utint8 to\_x, uint8 to\_y,
26. bytes5 pieceName, bytes5 color);
27. }

Board, Square contract are very simple as they just provide the setter and getter methods that may be needed.

**Piece:**

1. // defines the color of this piece
2. enum Color{WHITE, BLACK}
3. // a bool variable that shows whether this piece is alive
4. bool private isAlive;
5. // the color of this piece
6. Color private pieceColor;
7. // records the name of this piece
8. bytes5 pieceName;
9. // the setter and getter methods for fields
10. function setAlive(bool \_isAlive) public;
11. function setColor(Color color) public;
12. function getAlive() public returns (bool isAlive);
13. function getColor() public returns (Color color);
14. // used to judge whether this move is valid of not
15. function isMoveValid(uint8 from\_x,uint8 from\_y,uint8 to\_x,uint8 to\_y) returns (bool);

**King:**

1. contract King is Piece{
2. bool private castlingDone = false;
3. function isCastlingDone() public returns (bool);
4. function setCastlingDone(bool) public;
5. // used to judge whether this move is castling move
6. function isCastlingMove(uint8 from\_x,uint8 from\_y,uint8 to\_x,uint8 to\_y)
7. private returns (bool);
8. // override the method inherited from Piece
9. function isMoveValid(uint8 from\_x,uint8 from\_y,uint8 to\_x,uint8 to\_y)
10. override returns (bool);
11. }

Other types of Pieces are similar with King, the main feature of these subclass is that they all override the isMoveValid method to judge whether specific move is valid to this type of piece.

More details about these API could be seen on my [GitHub](https://github.com/Alan7921/BDL-Assignment4).

● how a game starts and ends

● how a player interacts with the game’s contract(s) to make a move

● if and how a player is notified about the game’s state and moves

● proposals for using design and coding patterns that increase gas fairness and efficiency, as

well as possible tradeoffs the coder will need to decide upon

● a *secure* randomized process to choose which player gets to play white in every game

● regarding time limits, you don’t have to follow the exact real-world rules (e.g. strictly

3/10/100 minutes per player), but you should design your own timing rules that ensure a

game does not run forever; your report should explain your choice in detail, taking into

account all types of possible attacks (that we discussed across all lectures)

● a description of which parts of the game (if any) could be performed off-chain, to reduce

cost, and how this could be done in a way that retains the trust and security guarantees of

an (entirely) on-chain execution