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

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)

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

● 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