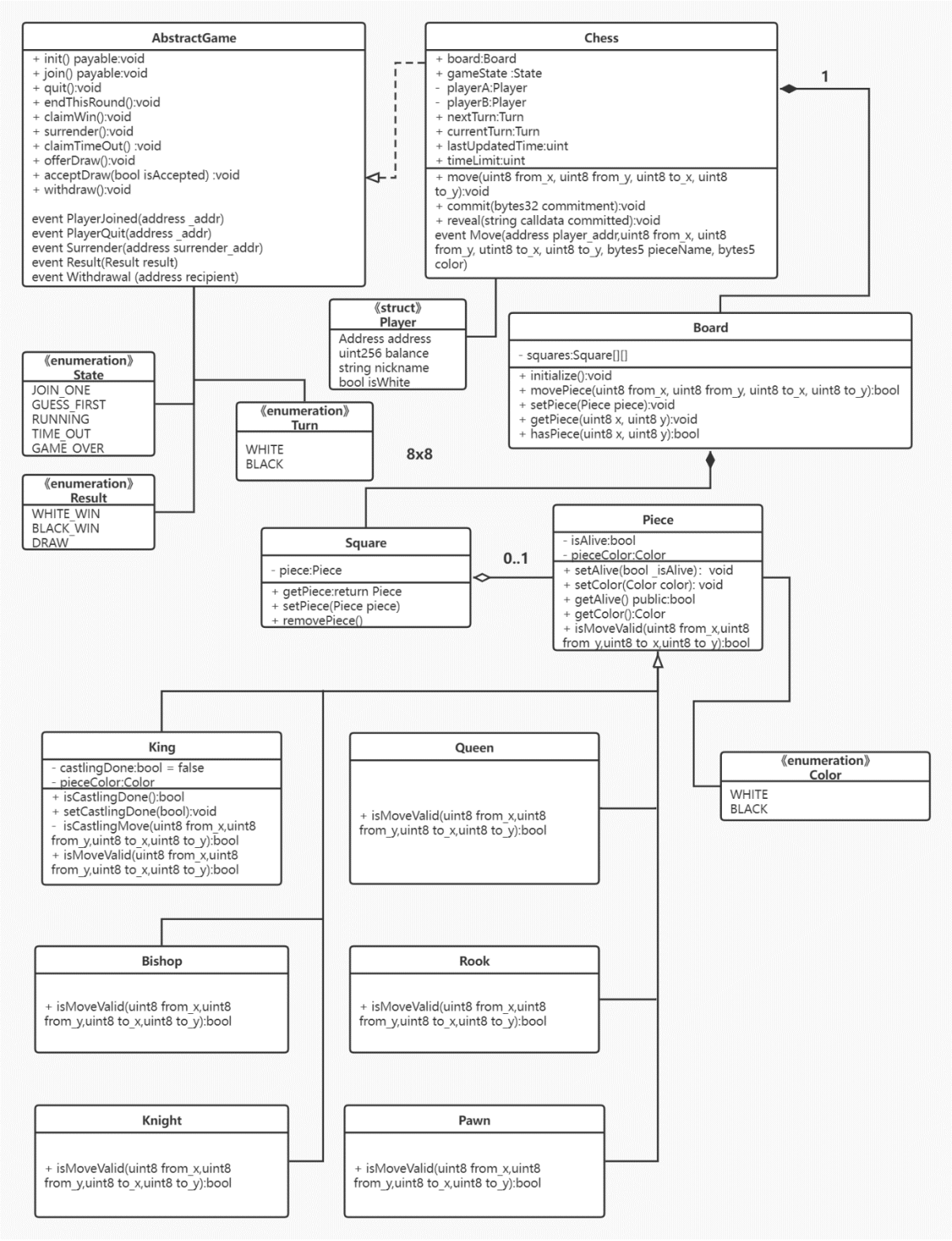

Figure 1 the Unified Modeling Language (UML) figure of this design

● 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 pennies 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
```

```
        not take valid action to prove they still have solution, the player
        who claimed would win.
    */
    function claimWin() public;
    //  used for claim that time is over, the opponent player lose.
    function claimTimeOut() public;
    //  offer draw to the opposite player
    function offerDraw() public;
    //  answer whether accept the draw
    function acceptDraw(bool isAccepted) public;
    //  used to withdraw ether from the contract
    function withdraw() public;
    //  record the join of player
    event PlayerJoined(address _addr);
    //  record the quit of player
    event PlayerQuit(address _addr);
    //  record the address of surrender
    event Surrender(address surrender_addr);
    //  record the result
    event Result(Result result);
    //  record the address of recipient
    event Withdrawal (address recipient);
```

Chess:

```
contract Chess is AbstractGame{
   struct Player {
        address addr;
        uint256 balance;
        string nickName;
        bool isWhite;
     }
   Board public board;
   State public gameState;
   Player private playerA; // First player
  Player private playerB; // Second player
  Turn public currentTurn;
  Turn public nextTurn;
    // record the timestamp updated last time
   uint public lastUpdatedTime;
    // a limitation to each turn/response
    uint public timeLimit;
    // used to decide which player takes whilte
   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.

● how a game starts and ends

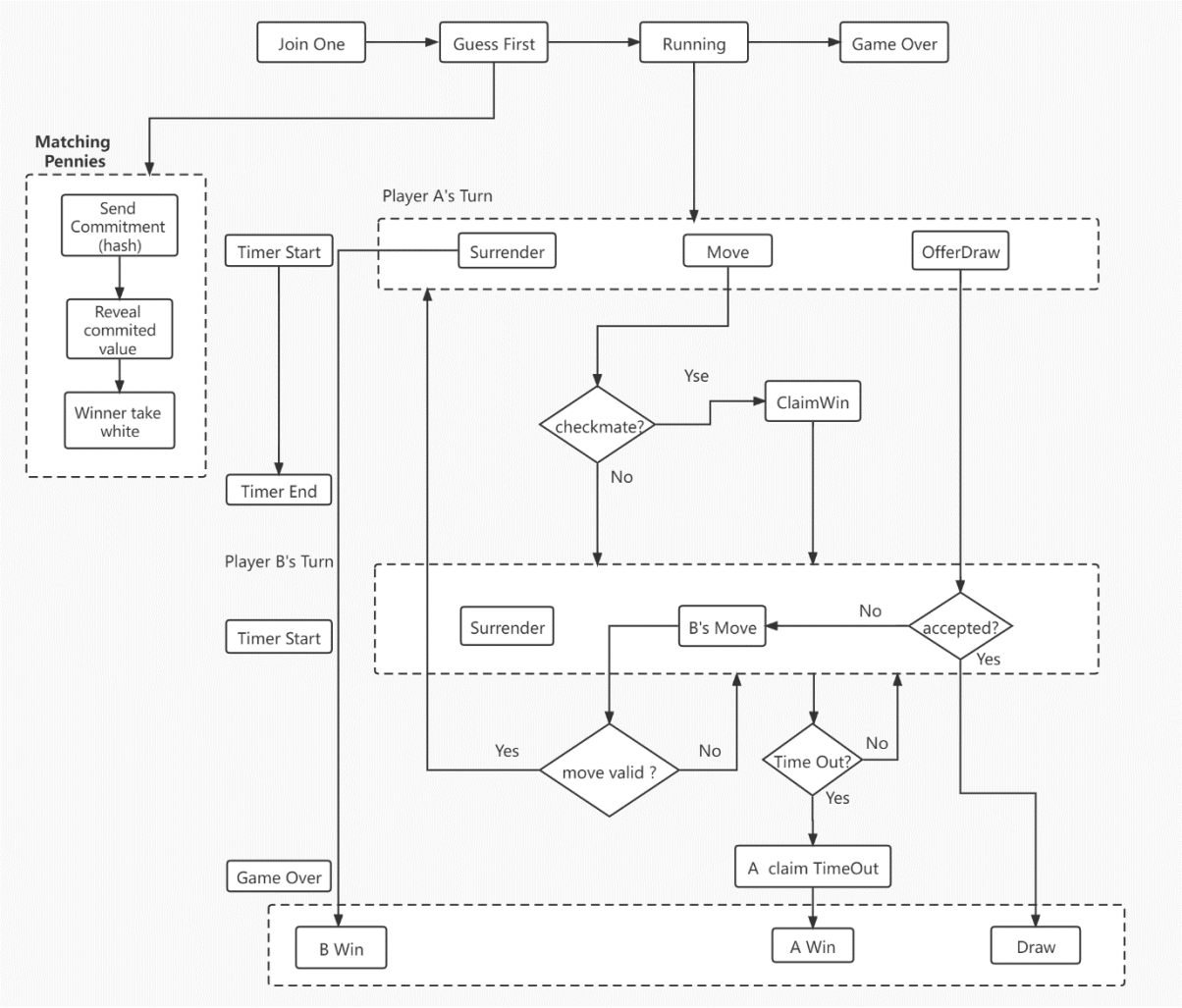

Figure 2 the Diagram of running flow

Answer:

As shown in Figure.2, the game start with the join of the second player. Then these two players need to play a matching pennies game to decide the color and order of their pieces. After this stage, the chess game get started and would keep running until the two players approach the end of this battle. Each player has a limited time in each round, if they run out of their time and do no take any valid action(typically it would be a valid move), they would lose the game, as their opponent is able to claim time out and win the game.

In each round, there are several actions that can be taken, “surrender”,”offerDraw”,”acceptDraw” and “move”. The main pattern is that if one player realize the “checkmate” after this move, he/she can call a function to claim that they are going to win, then it’s the opponent’s turn. If he/she is able to realize a valid move, it means that the opposite player still has chance. Otherwise, it means that it’s over. Besides, if two player both realize that they are in a stalemate, they can use the offerDraw and acceptDraw function to end the game.

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

Answer:

To move a piece, player only need to call the move function in the Chess contract. The method stub is shown below:

```
1.  function move(uint8 from_x, uint8 from_y, uint8 to_x, uint8 to_y) public;
```

Player only need to provide two coordinate pairs of the original position and the destination. Then the contract itself would make some calculation and decide that whether it's a valid movement or not.

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

Answer:

The corresponding variables and events are listed below. It can be seen that the state variable gameState is declared as public, thus, everyone who is interested in this game would be able to check the state and/or which player would move in this and next round. Besides, every movement would emit an event that records the original position, the destination, the color, and the piece.

```
1.      State public gameState;
2.      Turn public currentTurn;
3.      Turn public nextTurn;
4.      event Move(address player_addr,uint8 from_x, uint8 from_y, utint8 to_x, uint8 to_y,
5.             bytes5 pieceName, bytes5 color);
```

● 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

Answer:

There are certain design and coding patterns that is gas-saving which should be adopted and at the meantime there are also several gas-costly design that should be avoided.

Gas-Saving:

1. Short-circuiting

The meaning of Short-circuiting rule is that if we have two logical operand that need to be related with || or &&, let's put the cost-effective one at the first place, thus if it is true in the ||operation or it is false in the && operation, we do not need to pay for the expensive one. This is the so-called "short-circuiting".

2. Explicit function visibility

Actually, public and external functions differ in terms of gas usage. Since external function could only be called externally, thus its arguments would be read from calldata. This would be cheaper compared with public function. However, when it comes to the public function, since it would also be called internally, thus it's allocated to memory. In this case, during the designing, if we can make sure

that one function is open to external and would not be called internally, it would be better wo set it as “external” .

3. Proper data types

Try to use the most proper type, for instance if we are sure with the length of a string, compared with dynamic arrays, it would be more cost-effective to use a fix-sized bytesX.

4. Remove unnecessary libraries

Sometimes, it would be more cost-effective to implements an internal function in your contract instead of importing a library.

Gas-Costly:

1. Unreachable code
2. Repetition of assignment in a loop
3. Repetition of computation in a loop

Possible tradeoffs:

Sometimes, it’s quite common when we are designing a smart contract to make some trade-off between fairness, security, and efficiency.

For instance, in order to make sure that your contract would not get stuck or to avoid the risk associated with ether transfers. It would be better that you follows the pull over push pattern. This would definitely cost more gas than a simple transfer while the latter leaves a potential hazard which is not acceptable. This is the same with the check-effects-interaction pattern which is also implemented to resist the re-entrancy attack.

Besides the trade off on coding, a more general trade-off on the architecture would be whether it is necessary to realize a board in the chess game. Actually, there are more effective chess system ever appeared in the world such as the 0x88 system, which is quite efficient. If we do not need to implement a true board, it would be better to use a string to records and simulate the board and still keep its function in validation the movement of piece.

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

Answer:

I plan to use the matching pennies game to decide that which player play as which side as this kind of measure to the maximum keep the fair and security. Though it may cost extra gas, compared to the timestamp or the block.number, it could not be controlled by the adversary player.

Generally, the matching-pennies game could be divided into three stages. At the first stage, both these two players are required to send a hash value based on the nonce they chose plus the choice they made (0 or 1). Then at the second stage, they are required to reveal the cleartext of the nonce and the choice they used. The contract would automatically detect that whether someone has cheated or not and announce the winner of this guess. The winner would play first, in other word, he/she would take the

white pieces.

● 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)

Answer:

Currently, in order to prevent the DoS attack or any other attacks that aimed at getting the game stuck, I have added two public variables that relevant to time limitations as shown below:

```
1.    // record the timestamp updated last time
2.    uint public lastUpdatedTime;
3.    // a limitation to each turn/response
4.    uint public timeLimit;
```

It is defined that in each round, each player would have 5 minutes to conduct any actions he/she needs, including thinking, moving pieces, offering Draw, accepting Draw, or surrendering. Players are required to end their turn actively and take care of the left time of this round. Whenever one player has ended his round, the timer would update the instant and restart to counting for the next round.

Paying attention to the left time is regarded as a responsibility that players need to take. No matter which player in which case run out of his/her time, he/she would lose the game directly. The opposite player is able to use the "claimTimeOut" function to kill the game. This is also used to prevent the deliberate quit or pausing and would save the game from getting stuck.

However, this may result in one kind of potential attack that aims to use this rule to win the game directly. In theory, it is possible that the adversary can prevent the transaction of its opponent being packed on the chain by keep increasing the gas price. If the adversary keep attacking for more than five minutes, he or she would win the game without even one move. However, the cost of conducting this kind of attack is incredibly high, especially on the Ethereum chain. The key point here is balancing the possible earnings and the cost of conducting this kind of attack. If the adversary could only earn one ether with winning the chess, he or she would never pay maybe hundreds of ethers to cheat in this way.

Besides, when it comes to attacks, there are several hazards that are common in smart contracts. Firstly, underflow and overflow, solidity 0.80 has made the check of these hazards a default behavior. Thus, we do not need to include and call the SafeMath library anymore. Secondly, the re-entrant attack, since our chess contract is used to playing a chess game instead of depositing and transferring ether, it has little connection with this kind of attack. If we decide to add a bet for the game, we'd better follow the Check-Effects-Interactions Pattern to make sure that all the internal states/variables are changed first and require the player to withdraw their rewards instead of actively assigning ethers. DoS and front running have been discussed on the previous paragraph and have been handled carefully. Furthermore, since a truly secure randomized process has been included, thus the timestamp would not be manipulated, which prevent this kind of attack.

● 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

Answer:

Actually, from my point of view, I think that it would be better that we only use the blockchain to records the movement and events in the chess game, instead of requiring players calling functions to make transactions with the contract to realize a move or to take any actions. Currently, in order to realize a thorough, complete chess with inherited property, I have defined so many contracts. The deployment of these contracts would cost lots of gas. This is totally not that cost-effective.

A better solution is that let's exploit the advantages of blockchain. Firstly, let's think that what's the biggest advantages of this kind of technology that could be adopted in a chess game? That must be the immutability. Since no one can change the content that has been posted on chain, thus it would be the best place to record the result of each move. Thus, actually we do not need to implement a complete chess game on chain, instead we can realize a recorder or parser that records and parse the state and situation in the battle. The chess game could be conducted off-chain relying on a third-party chess client. After each move, the player generate a string literal that represent the move. There are many notation methods that could be used to describe the movement in chess. The most common one is portable game notation, which is also known as PGN. Following the rule of this descriptive text, all the events in a chess game could be completely represented. In order to prevent cheating behavior, before sending each string, a signature from the opposite player would be needed. The signature guarantee that both the two side are satisfied with this new state.

In this case, the contract only need to handle the initialization, the join, the guess first, the recording and the issuing of events(win,draw,surrender,tiemOut) and do not need to verify whether each move is valid or not anymore which would definitely save a large number of ethers.