

Design Chess

Let's design a system to play online chess.

We'll cover the following

- System Requirements
- Use case diagram
- Class diagram
- Activity diagrams
- Code

Chess is a two-player strategy board game played on a chessboard, which is a checkered gameboard with 64 squares arranged in an 8×8 grid. There are a few versions of game types that people play all over the world. In this design problem, we are going to focus on designing a two-player online chess game.



System Requirements

We'll focus on the following set of requirements while designing the game of chess:

1. The system should support two online players to play a game of chess.
2. All rules of international chess will be followed.
3. Each player will be randomly assigned a side, black or white.
4. Both players will play their moves one after the other. The white side plays the first move.
5. Players can't cancel or roll back their moves.
6. The system should maintain a log of all moves by both players.
7. Each side will start with 8 pawns, 2 rooks, 2 bishops, 2 knights, 1 queen, and 1 king.
8. The game can finish either in a checkmate from one side, forfeit or stalemate (a draw), or resignation.

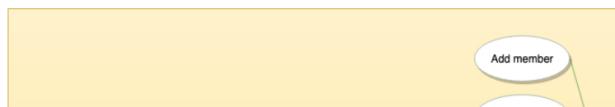
Use case diagram

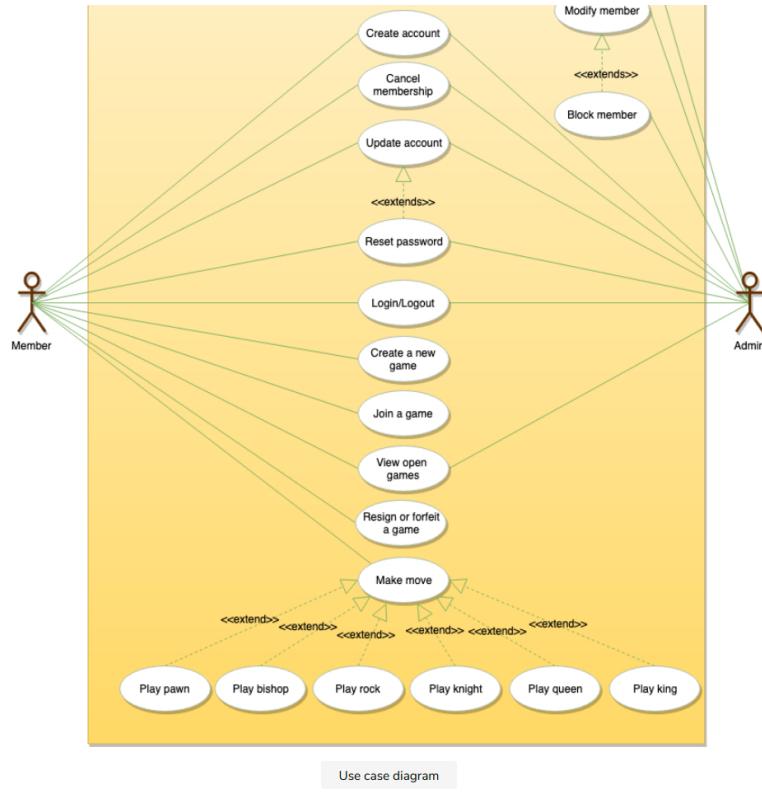
We have two actors in our system:

- **Player:** A registered account in the system, who will play the game. The player will play chess moves.
- **Admin:** To ban/modify players.

Here are the top use cases for chess:

- **Player moves a piece:** To make a valid move of any chess piece.
- **Resign or forfeit a game:** A player resigns from/forfeits the game.
- **Register new account/Cancel membership:** To add a new member or cancel an existing member.
- **Update game log:** To add a move to the game log.





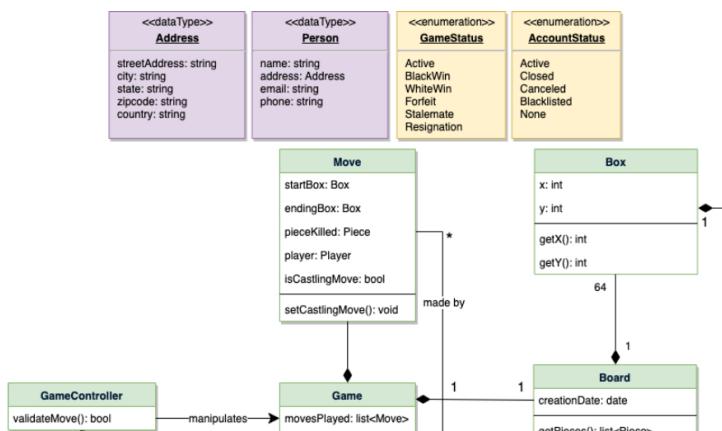
Use case diagram

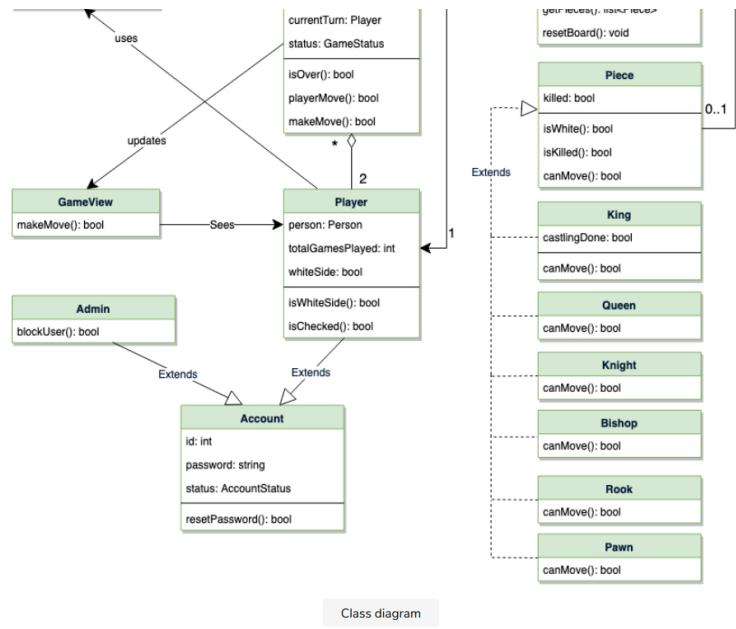
Class diagram

#

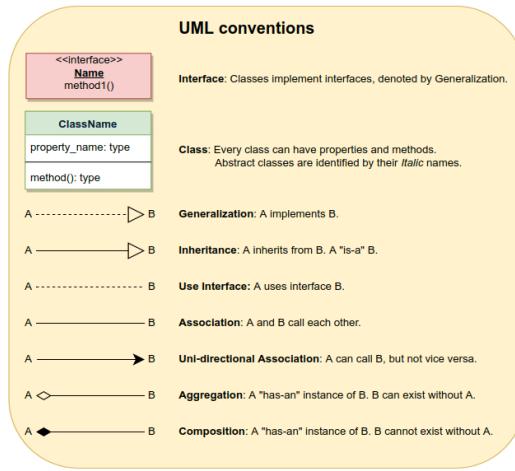
Here are the main classes for chess:

- **Player:** Player class represents one of the participants playing the game. It keeps track of which side (black or white) the player is playing.
- **Account:** We'll have two types of accounts in the system: one will be a player, and the other will be an admin.
- **Game:** This class controls the flow of a game. It keeps track of all the game moves, which player has the current turn, and the final result of the game.
- **Box:** A box represents one block of the 8x8 grid and an optional piece.
- **Board:** Board is an 8x8 set of boxes containing all active chess pieces.
- **Piece:** The basic building block of the system, every piece will be placed on a box. This class contains the color the piece represents and the status of the piece (that is, if the piece is currently in play or not). This would be an abstract class and all game pieces will extend it.
- **Move:** Represents a game move, containing the starting and ending box. The Move class will also keep track of the player who made the move, if it is a castling move, or if the move resulted in the capture of a piece.
- **GameController:** Player class uses GameController to make moves.
- **GameView:** Game class updates the GameView to show changes to the players.





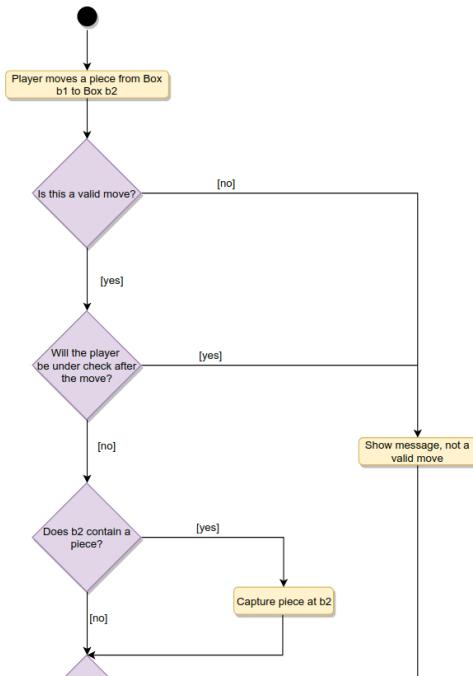
Class diagram

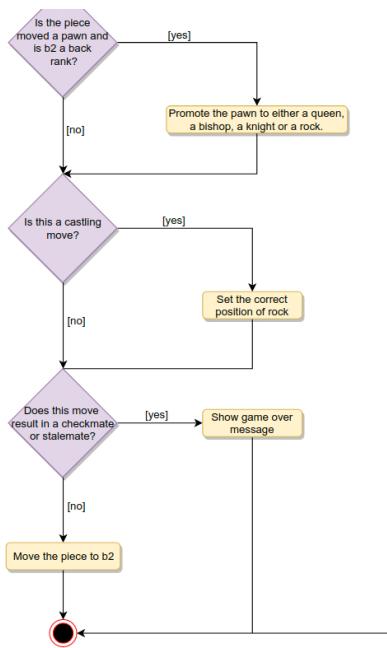


Activity diagrams

#

Make move: Any Player can perform this activity. Here are the set of steps to make a move:





Code

#

Here is the code for the top use cases.

Enums, DataTypes, Constants: Here are the required enums, data types, and constants:

Java	Python
<pre> 1 class GameStatus(Enum): 2 ACTIVE, BLACK_WIN, WHITE_WIN, FORFEIT, STALEMATE, RESIGNATION = 1, 2, 3, 4, 5, 6 3 4 5 class AccountStatus(Enum): 6 ACTIVE, CLOSED, CANCELED, BLACKLISTED, NONE = 1, 2, 3, 4, 5 7 8 9 class Address: 10 def __init__(self, street, city, state, zip_code, country): 11 self.__street_address = street 12 self.__city = city 13 self.__state = state 14 self.__zip_code = zip_code 15 self.__country = country 16 17 18 class Person(): 19 def __init__(self, name, address, email, phone): 20 self.__name = name 21 self.__address = address 22 self.__email = email 23 self.__phone = phone 24 </pre>	

Box: To encapsulate a cell on the chess board:

Java	Python
<pre> 1 class Box: 2 def __init__(self, piece, x, y): 3 self.__piece = piece 4 self.__x = x 5 self.__y = y 6 7 def get_piece(self): 8 return self.__piece 9 10 def set_piece(self, piece): 11 self.__piece = piece 12 13 def get_x(self): 14 return self.__x 15 16 def set_x(self, x): 17 self.__x = x 18 19 def get_y(self): 20 return self.__y 21 22 def set_y(self, y): 23 self.__y = y </pre>	

Piece: An abstract class to encapsulate common functionality of all chess pieces:

Java	Python
------	--------

```
1 from abc import ABC, abstractmethod
2
3 class Piece(ABC):
4     def __init__(self, white=False):
5         self._killed = False
6         self._white = white
7
8     def is_white(self):
9         return self._white
10
11    def set_white(self, white):
12        self._white = white
13
14    def is_killed(self):
15        return self._killed
16
17    def set_killed(self, killed):
18        self._killed = killed
19
20    def can_move(self, board, start_box, end_box):
21        None
22
```

King: To encapsulate King as a chess piece:

Java	Python
------	--------

```
1 class King(Piece):
2     def __init__(self, white):
3         self._castling_done = False
4         super().__init__(white)
5
6     def is_castling_done(self):
7         return self._castling_done
8
9     def set_castling_done(self, castling_done):
10        self._castling_done = castling_done
11
12    def can_move(self, board, start_box, end_box):
13        # we can't move the piece to a box that has a piece of the same color
14        if end_box.get_piece().is_white() == self.is_white():
15            return False
16
17        x = abs(start_box.get_x() - end_box.get_x())
18        y = abs(start_box.get_y() - end_box.get_y())
19        if x + y == 1:
20            # check if self move will not result in king being attacked, if so return True
21            return True
22
23        return self.is_valid_castling(board, start_box, end_box)
24
25    def is_valid_castling(self, board, start, end):
26
27        if self.is_castling_done():
28            return False
29
30        # check for the white king castling
31        if self.is_white() and start.get_x() == 0 and start.get_y() == 4 and end.get_y() == 0:
```

Knight: To encapsulate Knight as a chess piece:

Java	Python
------	--------

```
1 class Knight(Piece):
2     def __init__(self, white):
3         super().__init__(white)
4
5     def can_move(self, board, start, end):
6
7         # we can't move the piece to a box that has a piece of the same color
8         if end.get_piece().is_white() == self.is_white():
9             return False
10
11        x = abs(start.get_x() - end.get_x())
12        y = abs(start.get_y() - end.get_y())
13        return x * y == 2
```

Board: To encapsulate a chess board:

Java	Python
------	--------

```
1 class Board:
2     def __init__(self):
3         self._boxes = [[[]]]
4
5     def Board(self):
6         self.reset_board()
7
8     def get_box(self, x, y):
9         if x < 0 or x > 7 or y < 0 or y > 7:
10             raise Exception("Index out of bound")
```

```

11 |     return self._boxes[x][y]
12 |
13 | def reset_board(self):
14 |     # initialize white pieces
15 |     boxes[0][0] = Box(0, 0, Rook(True))
16 |     boxes[0][1] = Box(0, 1, Knight(True))
17 |     boxes[0][2] = Box(0, 2, Bishop(True))
18 |     ...
19 |     boxes[1][0] = Box(1, 0, Pawn(True))
20 |     boxes[1][1] = Box(1, 1, Pawn(True))
21 |     ...
22 |
23 |     # initialize black pieces
24 |     boxes[7][0] = Box(7, 0, Rook(False))
25 |     boxes[7][1] = Box(7, 1, Knight(False))
26 |     boxes[7][2] = Box(7, 2, Bishop(False))
27 |     ...
28 |     boxes[6][0] = Box(6, 0, Pawn(False))
29 |     boxes[6][1] = Box(6, 1, Pawn(False))
30 |     ...
31 |

```

Player: To encapsulate a chess player:

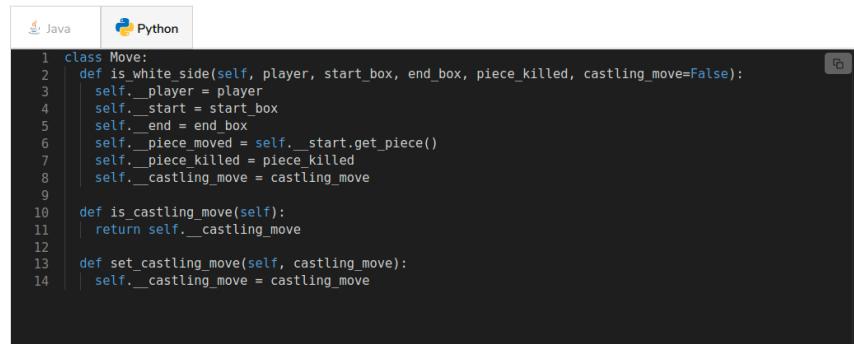


```

Java Python
1 class Player(Account):
2     def __init__(self, person, white_side=False):
3         self._person = person
4         self._white_side = white_side
5
6     def is_white_side(self):
7         return self._white_side
8

```

Move: To encapsulate a chess move:

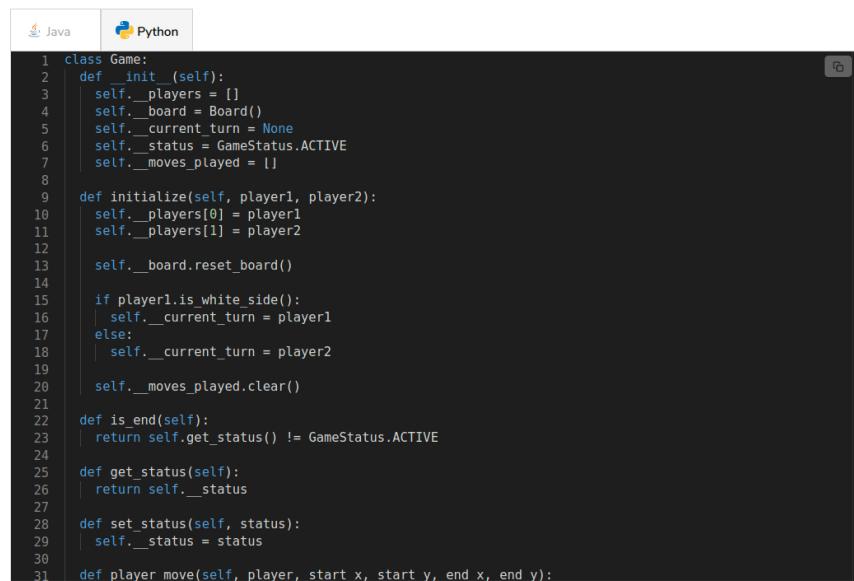


```

Java Python
1 class Move:
2     def is_white_side(self, player, start_box, end_box, piece_killed, castling_move=False):
3         self._player = player
4         self._start = start_box
5         self._end = end_box
6         self._piece_moved = self._start.get_piece()
7         self._piece_killed = piece_killed
8         self._castling_move = castling_move
9
10    def is_castling_move(self):
11        return self._castling_move
12
13    def set_castling_move(self, castling_move):
14        self._castling_move = castling_move

```

Game: To encapsulate a chess game:



```

Java Python
1 class Game:
2     def __init__(self):
3         self._players = []
4         self._board = Board()
5         self._current_turn = None
6         self._status = GameStatus.ACTIVE
7         self._moves_played = []
8
9     def initialize(self, player1, player2):
10        self._players[0] = player1
11        self._players[1] = player2
12
13        self._board.reset_board()
14
15        if player1.is_white_side():
16            self._current_turn = player1
17        else:
18            self._current_turn = player2
19
20        self._moves_played.clear()
21
22    def is_end(self):
23        return self._status != GameStatus.ACTIVE
24
25    def get_status(self):
26        return self._status
27
28    def set_status(self, status):
29        self._status = status
30
31    def player_move(self, player, start_x, start_y, end_x, end_y):

```

[← Back](#)

[Next →](#)

Design a Restaurant Management sys...

Design an Online Stock Brokerage Sys...

Mark as Completed

