Design Chess

Let's design a system to play online chess.

We'll cover the following

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.

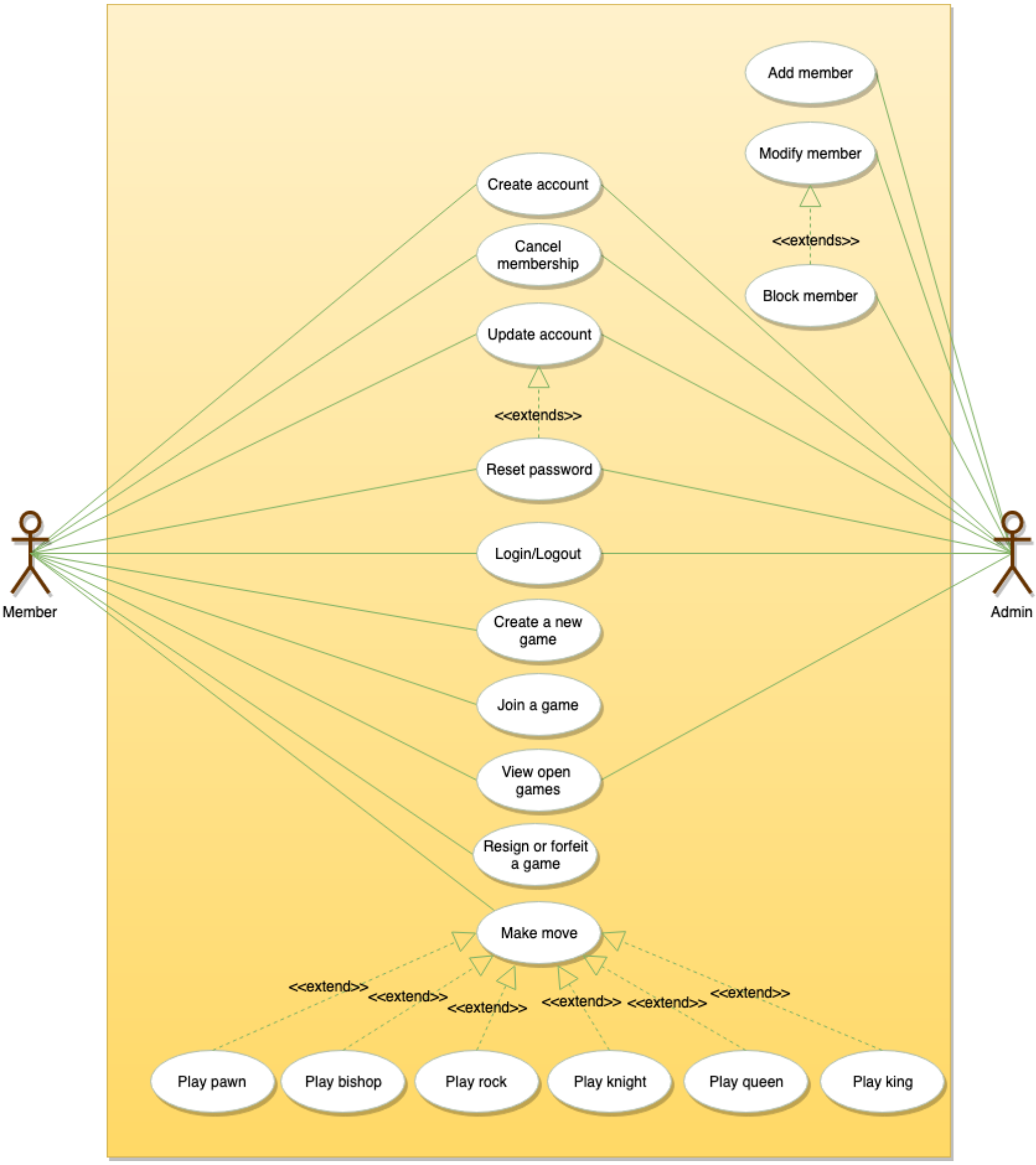
Use case diagram

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.



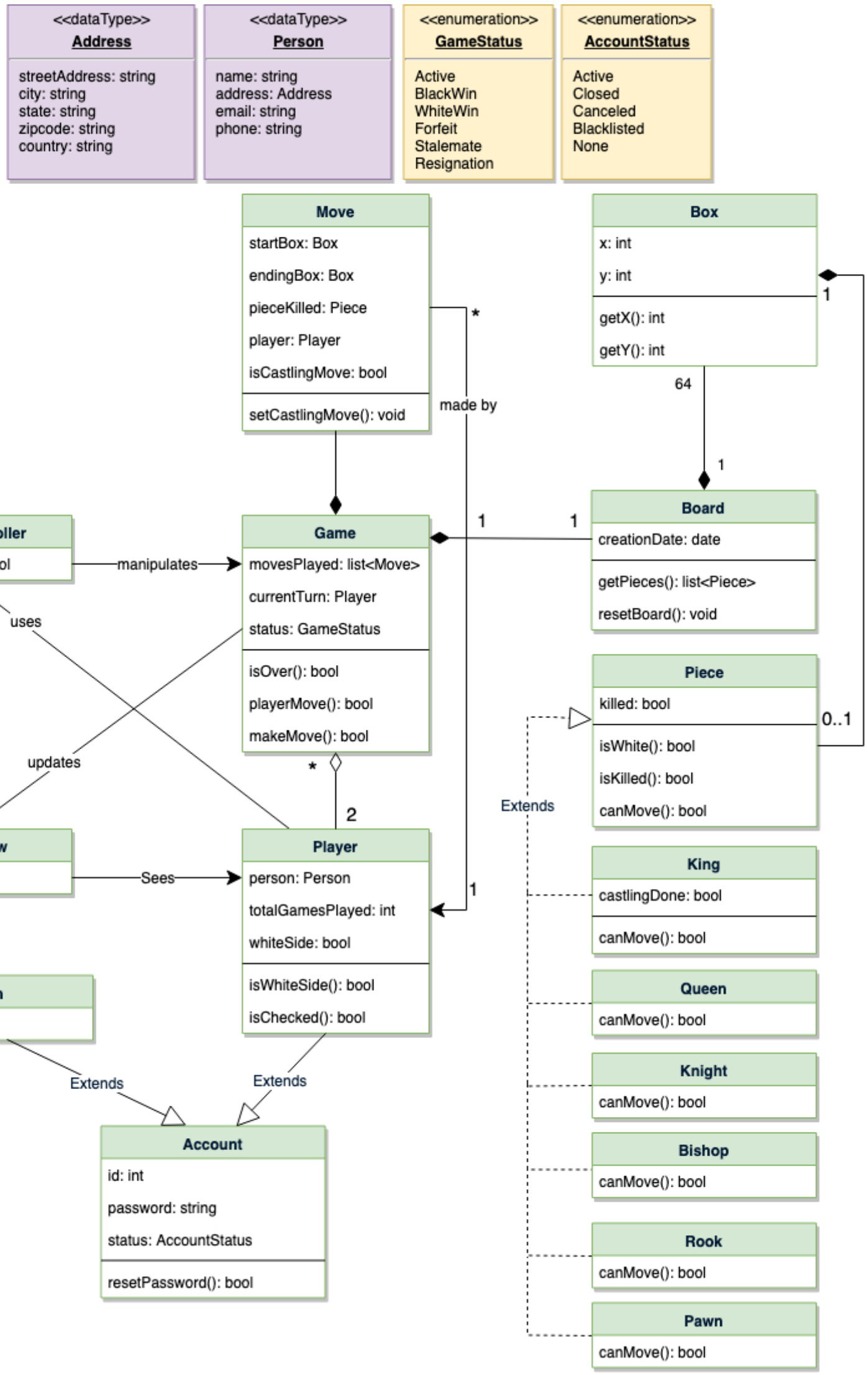
Use case diagram

Class diagram

Here are the main classes for chess:

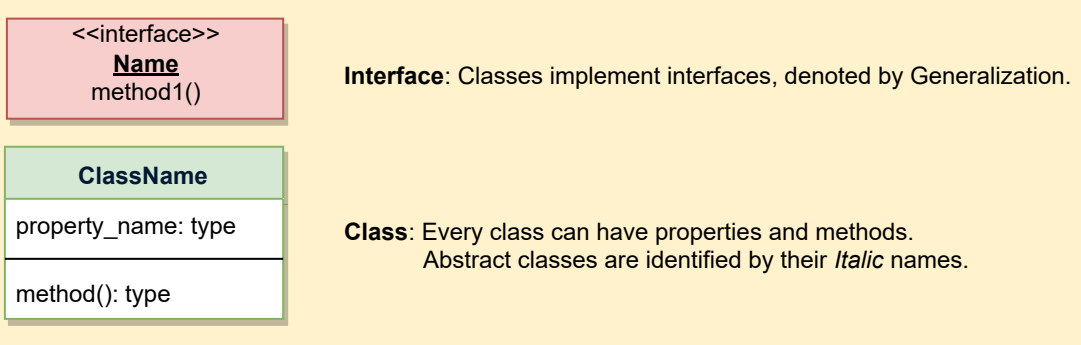
- **Player:** Player class represents one of the participants playing the game. It keeps track of which side (black or white) the player is playing.
- **Account:** We'll have two types of accounts in the system: one will be a player, and the other will be an admin.
- **Game:** This class controls the 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.



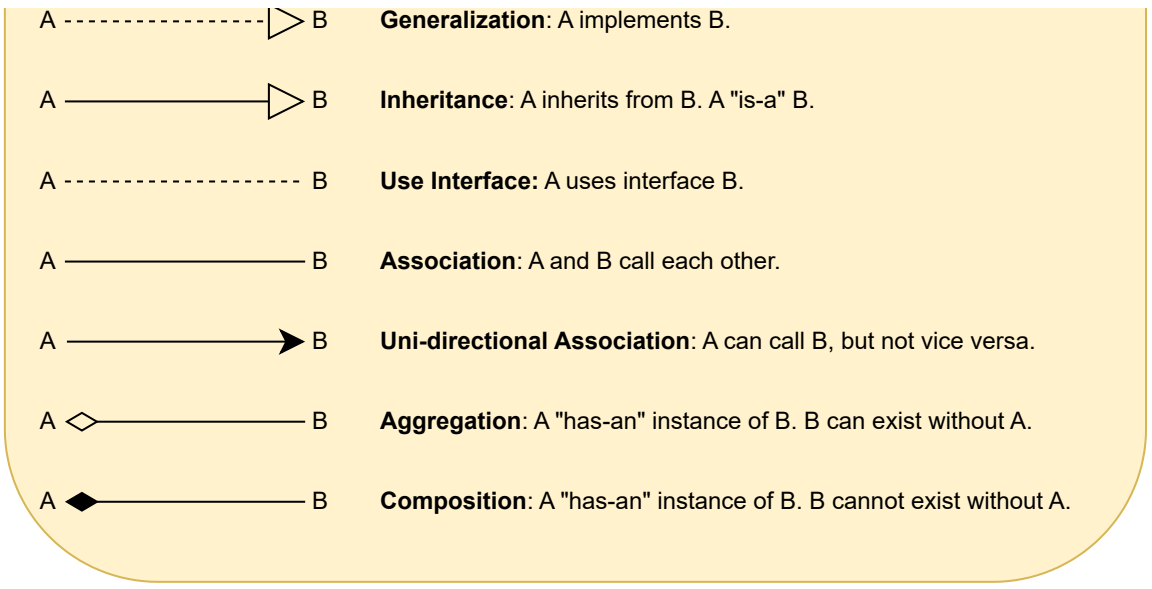
Class diagram

UML conventions



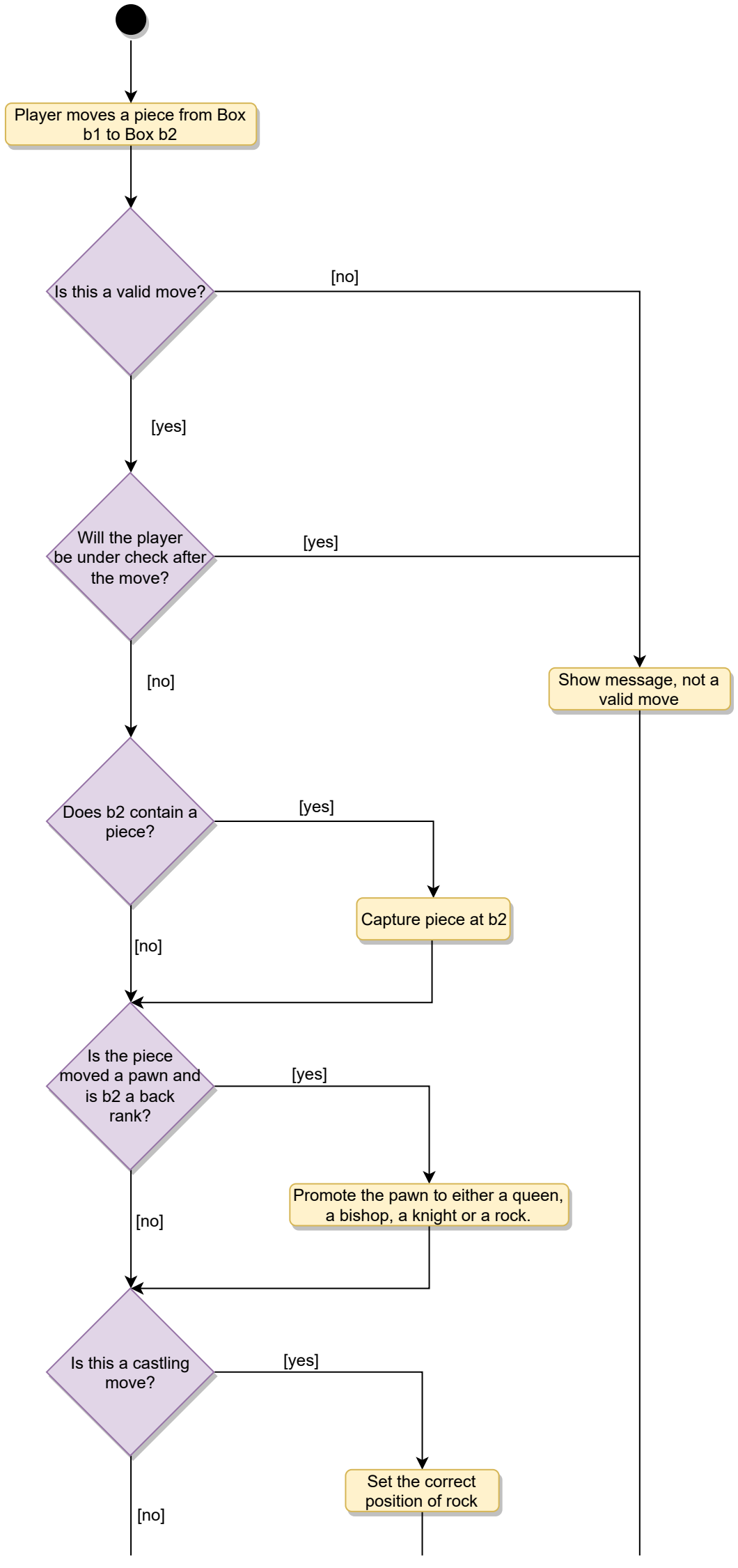
Interface: Classes implement interfaces, denoted by Generalization.

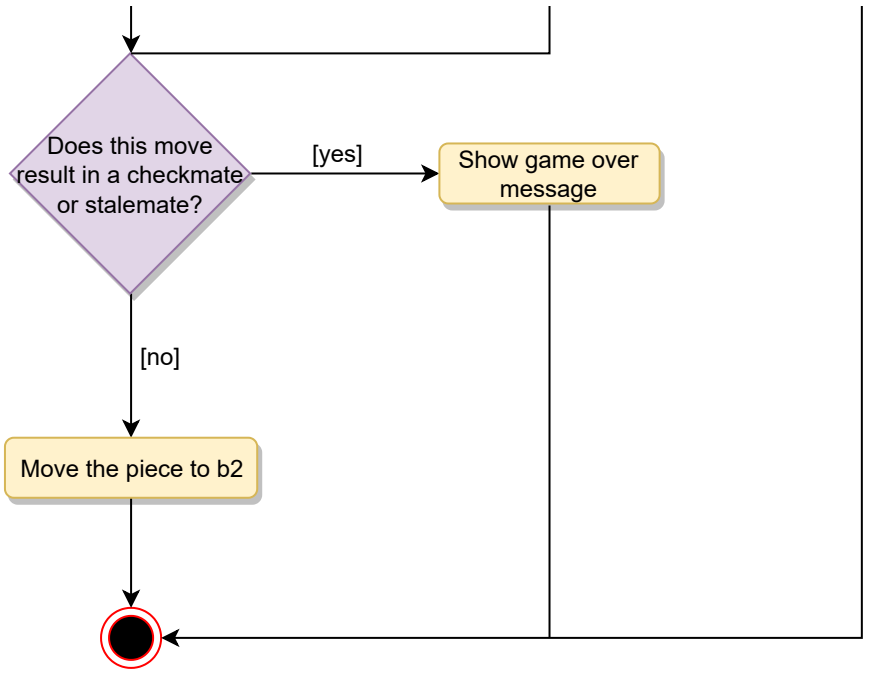
Class: Every class can have properties and methods.
Abstract classes are identified by their *Italic* names.



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

```python
class GameStatus(Enum):
  ACTIVE, BLACK_WIN, WHITE_WIN, FORFEIT, STALEMATE, RESIGNATION = 1, 2, 3, 4, 5, 6


class AccountStatus(Enum):
  ACTIVE, CLOSED, CANCELED, BLACKLISTED, NONE = 1, 2, 3, 4, 5


class Address:
  def __init__(self, street, city, state, zip_code, country):
    self.__street_address = street
    self.__city = city
    self.__state = state
    self.__zip_code = zip_code
    self.__country = country


class Person():
  def __init__(self, name, address, email, phone):
    self.__name = name
    self.__address = address
    self.__email = email
    self.__phone = phone
```

Box: To encapsulate a cell on the chess board:

Java Python

```python
class Box:
  def __init__(self, piece, x, y):
    self.__piece = piece
    self.__x = x
    self.__y = y

  def get_piece(self):
    return self.__piece

  def set_piece(self, piece):
    self.__piece = piece

  def get_x(self):
    return self.__x
```

```
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
```

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
```

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(Rook(True), 0, 0);
16     boxes[0][1] = Box(Knight(True), 0, 1);
17     boxes[0][2] = Box(Bishop(True), 0, 2);
18     #...
19     boxes[1][0] = Box(Pawn(True), 1, 0);
20     boxes[1][1] = Box(Pawn(True), 1, 1);
21     #...
22
23     # initialize black pieces
24     boxes[7][0] = Box(Rook(False), 7, 0);
25     boxes[7][1] = Box(Knight(False), 7, 1);
26     boxes[7][2] = Box(Bishop(False), 7, 2);
27     #...
28     boxes[6][0] = Box(Pawn(False), 6, 0);
29     boxes[6][1] = Box(Pawn(False), 6, 1);
30     # ...
```

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

```python
class Move:
  def is_white_side(self, player, start_box, end_box, piece_killed, castling_move=False):
    self.__player = player
    self.__start = start_box
    self.__end = end_box
    self.__piece_moved = self.__start.get_piece()
    self.__piece_killed = piece_killed
    self.__castling_move = castling_move

  def is_castling_move(self):
    return self.__castling_move

  def set_castling_move(self, castling_move):
    self.__castling_move = castling_move
```

Game: To encapsulate a chess game:

Java Python

```python
class Game:
  def __init__(self):
    self.__players = []
    self.__board = Board()
    self.__current_turn = None
    self.__status = GameStatus.ACTIVE
    self.__moves_played = []

  def initialize(self, player1, player2):
    self.__players[0] = player1
    self.__players[1] = player2

    self.__board.reset_board()

    if player1.is_white_side():
      self.__current_turn = player1
    else:
      self.__current_turn = player2

    self.__moves_played.clear()

  def is_end(self):
    return self.get_status() != GameStatus.ACTIVE

  def get_status(self):
    return self.__status

  def set_status(self, status):
    self.__status = status
```

Back

Design a Restaurant Management system

Next

Design an Online Stock Brokerage System

Mark as Completed