Design Chess

Let's design a system to play chess

We'll cover the following:

- System Requirements
- Use Case Diagram
- Class Diagram
- Activity Diagram
- 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.

Chess

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

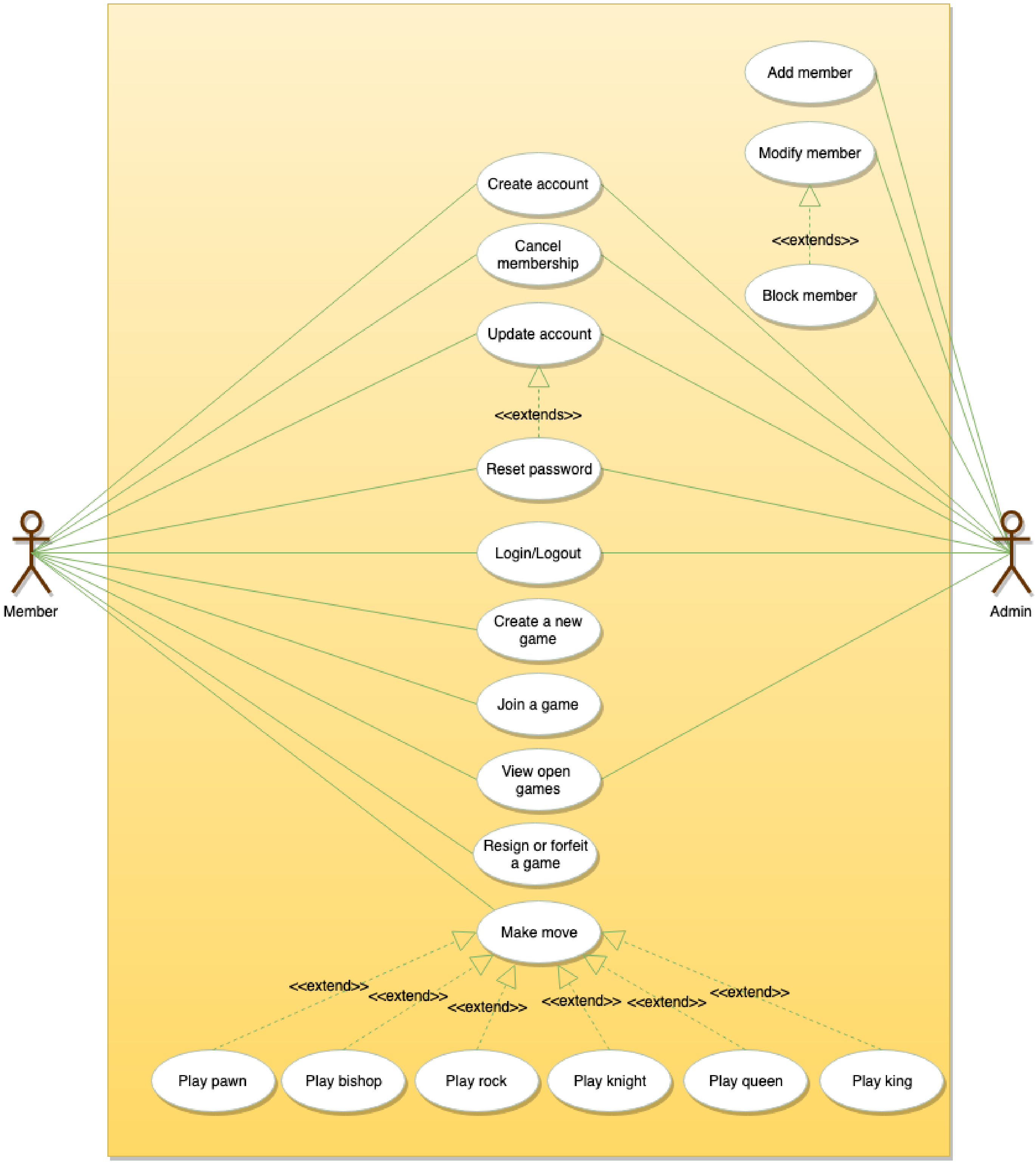
We have two actors in our system:

- **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.

Here is the use case diagram of our Chess Game:



Use Case Diagram for Chess

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.

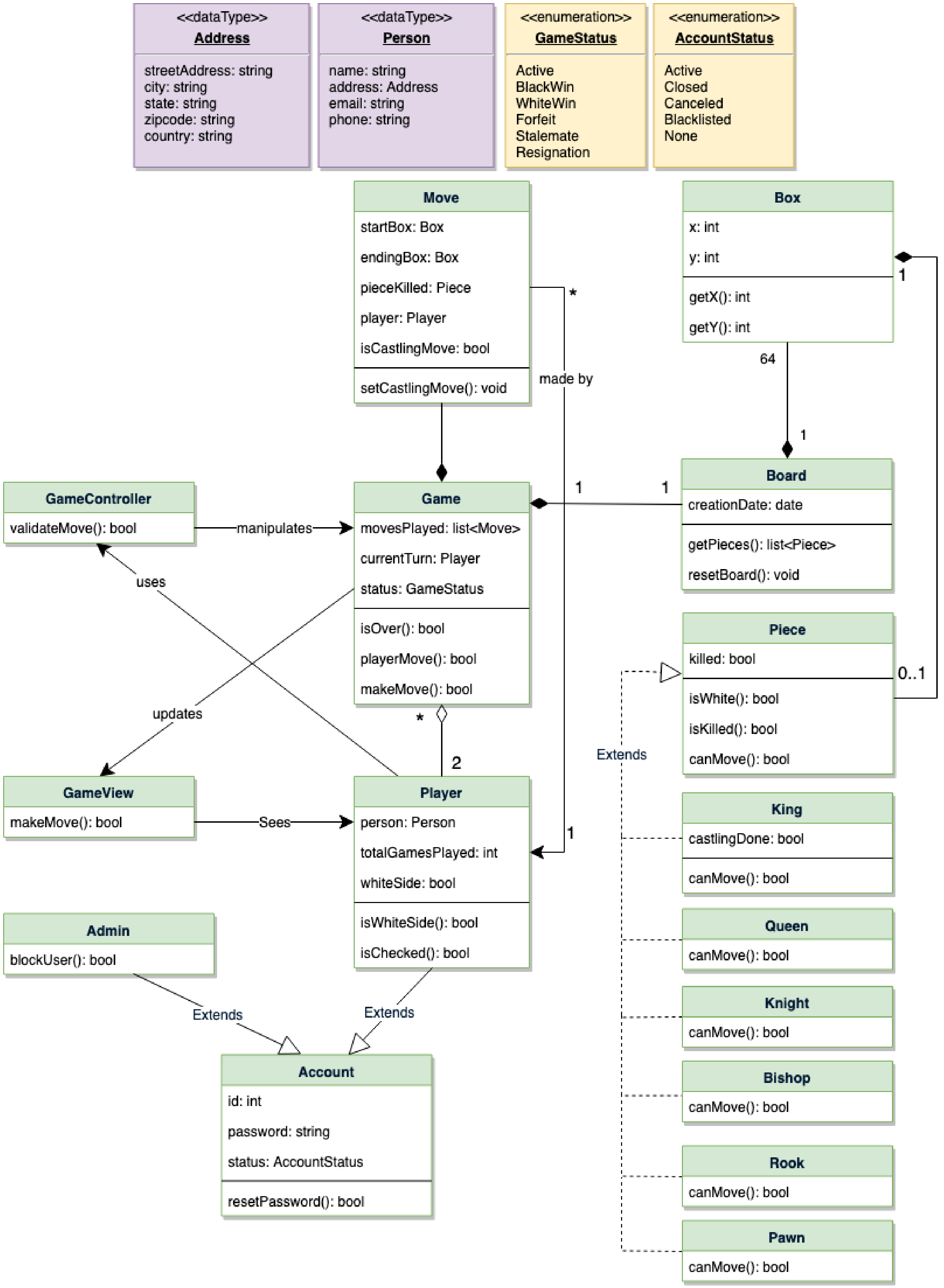
Board: Board is an 8x8 set of boxes containing all active chess pieces.

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.

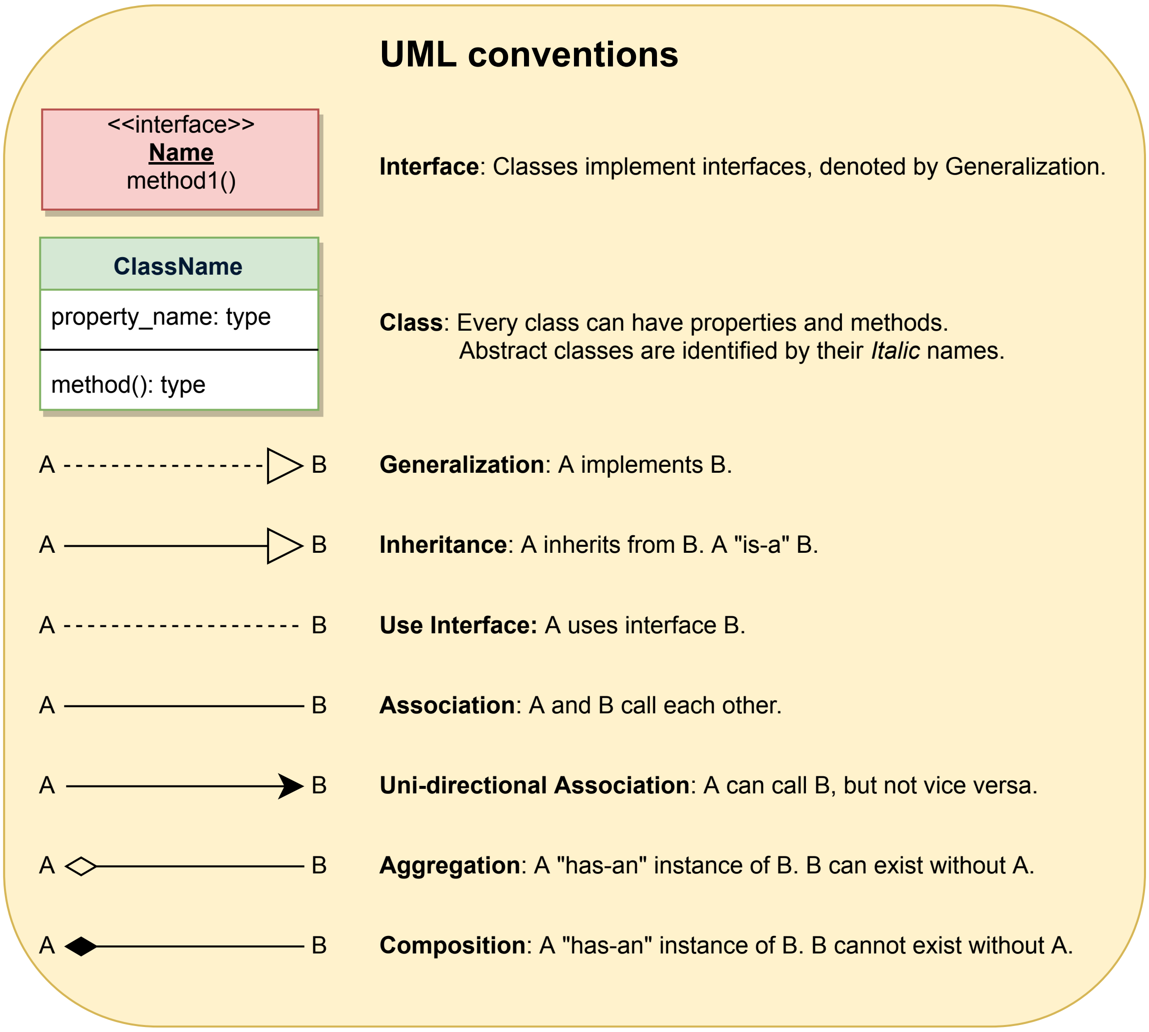
Move: Represents a game move, containing the starting and ending box. The Move class will also keep track of the player who made the move, if it is a castling move, or if the move resulted in the capture of a piece.

GameController: Player class uses GameController to make moves.

GameView: Game class updates the GameView to show changes to the players.



Class Diagram for Chess



Interface: Classes implement interfaces, denoted by Generalization.

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

Generalization: A implements B.

Inheritance: A inherits from B. A "is-a" B.

Use Interface: A uses interface B.

Association: A and B call each other.

Uni-directional Association: A can call B, but not vice versa.

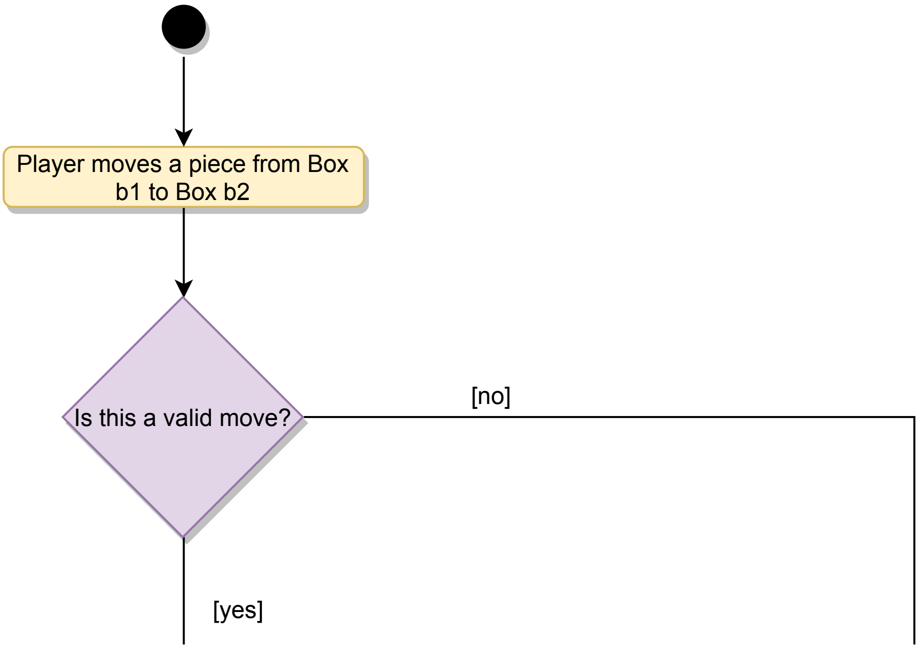
Aggregation: A "has-an" instance of B. B can exist without A.

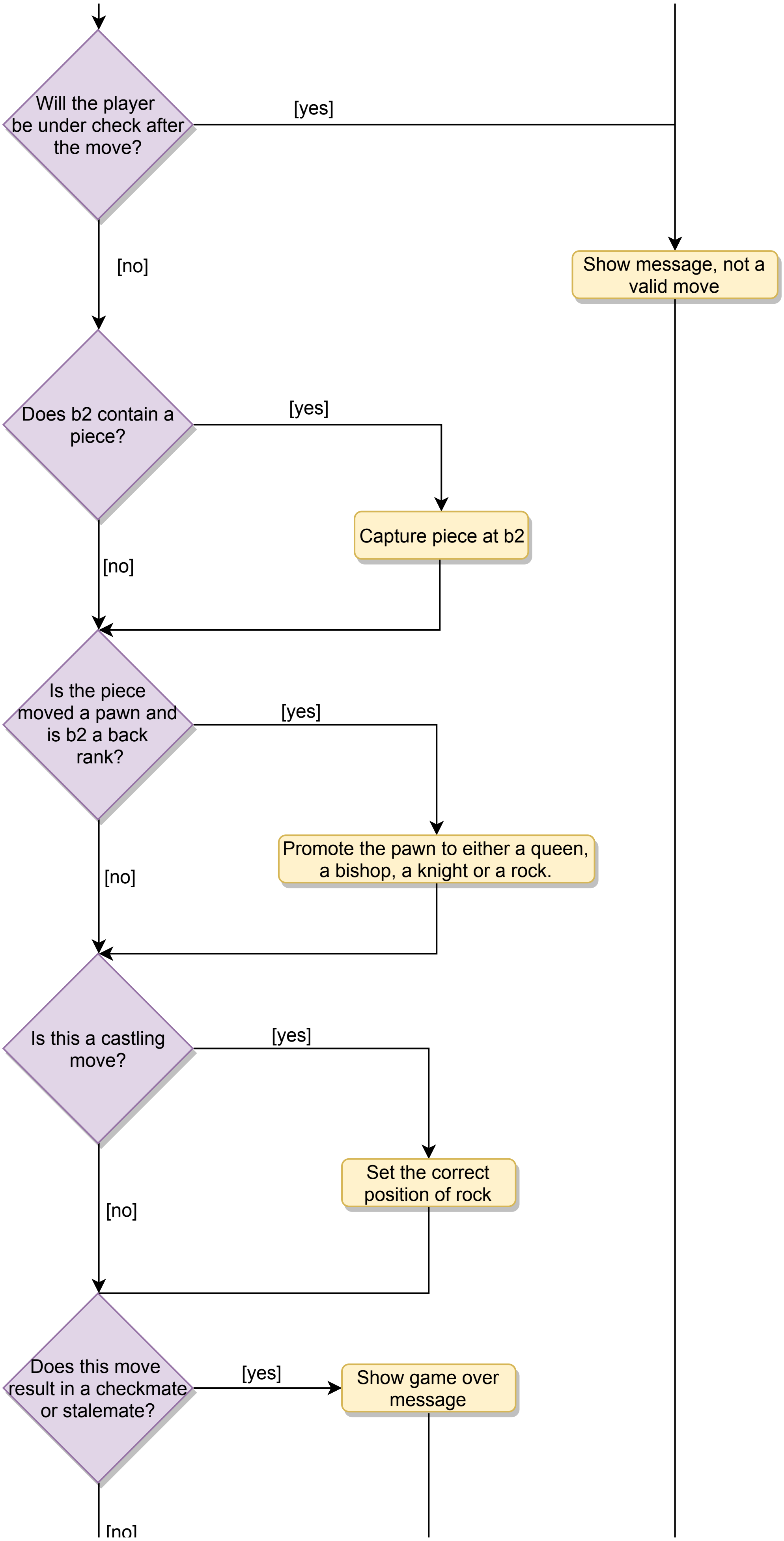
Composition: A "has-an" instance of B. B cannot exist without A.

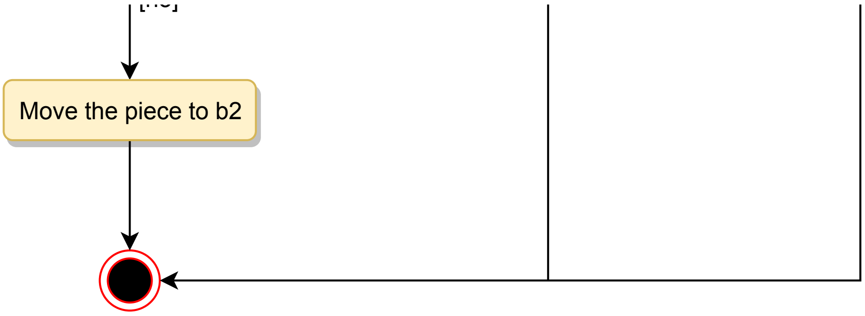
UML for Chess

Activity Diagram

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




Will the player be under check after the move?
[yes]
[no]
Show message, not a valid move
Does b2 contain a piece?
[yes]
Capture piece at b2
[no]
Is the piece moved a pawn and is b2 a back rank?
[yes]
Promote the pawn to either a queen, a bishop, a knight or a rock.
[no]
Is this a castling move?
[yes]
Set the correct position of rock
[no]
Does this move result in a checkmate or stalemate?
[yes]
Show game over message
[no]



Activity Diagram for Chess

Code

Here is the code for the top use cases.

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

```
class PieceType:
    ROOK = "rook"
    KNIGHT = "knight"
    BISHOP = "bishop"
    QUEEN = "queen"
    KING = "king"
    PAWN = "pawn"


CHESS_BOARD_SIZE = 8

INITIAL_PIECE_SET_SINGLE = [
    (PieceType.ROOK, 0, 0),
    (PieceType.KNIGHT, 1, 0),
    (PieceType.BISHOP, 2, 0),
    (PieceType.QUEEN, 3, 0),
    (PieceType.KING, 4, 0),
    (PieceType.BISHOP, 5, 0),
    (PieceType.KNIGHT, 6, 0),
    (PieceType.ROOK, 7, 0),
    (PieceType.PAWN, 0, 1),
    (PieceType.PAWN, 1, 1),
    (PieceType.PAWN, 2, 1),
    (PieceType.PAWN, 3, 1),
    (PieceType.PAWN, 4, 1),
    (PieceType.PAWN, 5, 1),
    (PieceType.PAWN, 6, 1),
    (PieceType.PAWN, 7, 1)
]
```

Board: To encapsulate a cell on the chess board:

```python
from copy import deepcopy
from .pieces import Piece, PieceFactory
from .moves import ChessPosition, MoveCommand
from .constants import CHESS_BOARD_SIZE, INITIAL_PIECE_SET_SINGLE, PieceType


class ChessBoard:
    def __init__(self, size=CHESS_BOARD_SIZE):
        self._size = size
        self._pieces = []
        self._white_king_position = None
        self._black_king_position = None
        self._initialize_pieces(INITIAL_PIECE_SET_SINGLE)

    def _initialize_pieces(self, pieces_setup: list):
        for piece_tuple in pieces_setup:
            type = piece_tuple[0]
            x = piece_tuple[1]
            y = piece_tuple[2]

            piece_white = PieceFactory.create(type, ChessPosition(x, y), Piece.WHITE)
            if type == PieceType.KING:
                piece_white.set_board_handle(self)
            self._pieces.append(piece_white)

            piece_black = PieceFactory.create(type, ChessPosition(self._size - x - 1, self._size - y - 1), Piece.BLACK)
            if type == PieceType.KING:
                piece_black.set_board_handle(self)
            self._pieces.append(piece_black)

    def get_piece(self, position: ChessPosition) -> Piece:
        for piece in self._pieces:
            if piece.position == position:
                return piece
        return None

    def beam_search_threat(self, start_position: ChessPosition, own_color, increment_x: int, increment_y: int):
        threatened_positions = []
        curr_x = start_position.x_coord
        curr_y = start_position.y_coord
        curr_x += increment_x
        curr_y += increment_y
        while curr_x >= 0 and curr_y >= 0 and curr_x < self._size and curr_y < self._size:
            curr_position = ChessPosition(curr_x, curr_y)
            curr_piece = self.get_piece(curr_position)
            if curr_piece is not None:
                if curr_piece.color != own_color:
                    threatened_positions.append(curr_position)
                break
            threatened_positions.append(curr_position)
            curr_x += increment_x
            curr_y += increment_y
        return threatened_positions

    def spot_search_threat(self, start_position: ChessPosition, own_color,
increment_x: int, increment_y: int,
                           threat_only=False, free_only=False):
        curr_x = start_position.x_coord + increment_x
        curr_y = start_position.y_coord + increment_y

        if curr_x >= self.size or curr_y >= self.size or curr_x < 0 or curr_y <
0:
            return None

        curr_position = ChessPosition(curr_x, curr_y)
        curr_piece = self.get_piece(curr_position)
        if curr_piece is not None:
            if free_only:
                return None
            return curr_position if curr_piece.color != own_color else None
        return curr_position if not threat_only else None

    @property
    def pieces(self):
        return deepcopy(self._pieces)

    @property
    def size(self):
        return self._size

    @property
    def white_king_position(self):
        return self._white_king_position

    @property
    def black_king_position(self):
        return self._black_king_position

    def execute_move(self, command: MoveCommand):
        source_piece = self.get_piece(command.src)
        for idx, target_piece in enumerate(self._pieces):
            if target_piece.position == command.dst:
                del self._pieces[idx]
                break
        source_piece.move(command.dst)

    def register_king_position(self, position: ChessPosition, color: str):
        if color == Piece.WHITE:
            self._white_king_position = position
        elif color == Piece.BLACK:
            self._black_king_position = position
        else:
            raise RuntimeError("Unknown color of the king piece")
```

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

```python
from abc import ABC
from .constants import PieceType
from .moves import ChessPosition
from .king import King
from .queen import Queen
from .knight import Knight
from .rook import Rook
from .bishop import Bishop
from .pawn import Pawn


class Piece(ABC):
    BLACK = "black"
    WHITE = "white"

    def __init__(self, position: ChessPosition, color):
        self._position = position
        self._color = color

    @property
    def position(self):
        return self._position

    @property
    def color(self):
        return self._color

    def move(self, target_position):
        self._position = target_position

    def get_threatened_positions(self, board):
        raise NotImplementedError

    def get_moveable_positions(self, board):
        raise NotImplementedError

    def symbol(self):
        black_color_prefix = '\u001b[31;1m'
        white_color_prefix = '\u001b[34;1m'
        color_suffix = '\u001b[0m'
        retval = self._symbol_impl()
        if self.color == Piece.BLACK:
            retval = black_color_prefix + retval + color_suffix
        else:
            retval = white_color_prefix + retval + color_suffix
        return retval
    def _symbol_impl(self):
        raise NotImplementedError

class PieceFactory:
    @staticmethod
    def create(piece_type: str, position: ChessPosition, color):
        if piece_type == PieceType.KING:
            return King(position, color)

        if piece_type == PieceType.QUEEN:
            return Queen(position, color)

        if piece_type == PieceType.KNIGHT:
            return Knight(position, color)

        if piece_type == PieceType.ROOK:
            return Rook(position, color)

        if piece_type == PieceType.BISHOP:
            return Bishop(position, color)

        if piece_type == PieceType.PAWN:
            return Pawn(position, color)
```

King: To encapsulate King as a chess piece:

```
from .pieces import Piece
from .moves import ChessPosition


class King(Piece):
    SPOT_INCREMENTS = [(1, -1), (1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1,
-1), (0, -1)]

    def __init__(self, position: ChessPosition, color: str):
        super().__init__(position, color)
        self._board_handle = None

    def set_board_handle(self, board):
        self._board_handle = board
        self._board_handle.register_king_position(self.position, self.color)

    def move(self, target_position: ChessPosition):
        Piece.move(self, target_position)
        self._board_handle.register_king_position(target_position, self.color)

    def get_threatened_positions(self, board):
        positions = []
        for increment in King.SPOT_INCREMENTS:
            positions.append(board.spot_search_threat(self._position,
self._color, increment[0], increment[1]))
        positions = [x for x in positions if x is not None]
        return positions

    def get_moveable_positions(self, board):
        return self.get_threatened_positions(board)

    def _symbol_impl(self):
        return 'KI'
```

Queen: To encapsulate Queen as a chess piece:

```
from .pieces import Piece


class Queen(Piece):
    BEAM_INCREMENTS = [(1, 1), (1, -1), (-1, 1), (-1, -1), (0, 1), (0, -1), (1,
0), (-1, 0)]

    def get_threatened_positions(self, board):
        positions = []
        for increment in (Queen.BEAM_INCREMENTS):
            positions += board.beam_search_threat(self._position, self._color,
increment[0], increment[1])
        return positions

    def get_moveable_positions(self, board):
        return self.get_threatened_positions(board)

    def _symbol_impl(self):
        return 'QU'
```

Knight: To encapsulate Knight as a chess piece:

```
from .pieces import Piece


class Knight(Piece):
    SPOT_INCREMENTS = [(2, 1), (2, -1), (-2, 1), (-2, -1), (1, 2), (1, -2),
(-1, 2), (-1, -2)]

    def get_threatened_positions(self, board):
        positions = []
        for increment in Knight.SPOT_INCREMENTS:
            positions.append(board.spot_search_threat(self._position,
self._color, increment[0], increment[1]))
        positions = [x for x in positions if x is not None]
        return positions

    def get_moveable_positions(self, board):
        return self.get_threatened_positions(board)

    def _symbol_impl(self):
        return 'KN'
```

Rook: To encapsulate Rook as a chess piece:

```python
from .pieces import Piece


class Rook(Piece):
    BEAM_INCREMENTS = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    def get_threatened_positions(self, board):
        positions = []
        for increment in Rook.BEAM_INCREMENTS:
            positions += board.beam_search_threat(self._position, self._color,
increment[0], increment[1])
        return positions

    def get_moveable_positions(self, board):
        return self.get_threatened_positions(board)

    def _symbol_impl(self):
        return 'RO'
```

Bishop: To encapsulate Bishop as a chess piece:

```python
from .pieces import Piece


class Bishop(Piece):
    BEAM_INCREMENTS = [(1, 1), (1, -1), (-1, 1), (-1, -1)]

    def get_threatened_positions(self, board):
        positions = []
        for increment in Bishop.BEAM_INCREMENTS:
            positions += board.beam_search_threat(self._position, self._color,
increment[0], increment[1])
        return positions

    def get_moveable_positions(self, board):
        return self.get_threatened_positions(board)

    def _symbol_impl(self):
        return 'BI'
```

Pawn: To encapsulate Pawn as a chess piece:

```
from .pieces import Piece
from .moves import ChessPosition


class Pawn(Piece):
    SPOT_INCREMENTS_MOVE = [(0, 1)]
    SPOT_INCREMENTS_MOVE_FIRST = [(0, 1), (0, 2)]
    SPOT_INCREMENTS_TAKE = [(-1, 1), (1, 1)]

    def __init__(self, position: ChessPosition, color: str):
        super().__init__(position, color)
        self._moved = False

    def get_threatened_positions(self, board):
        positions = []
        increments = Pawn.SPOT_INCREMENTS_TAKE
        for increment in increments:
            positions.append(board.spot_search_threat(self._position,
self._color, increment[0], increment[1] if self.color == Piece.WHITE else (-1)
* increment[1]))
        positions = [x for x in positions if x is not None]
        return positions

    def get_moveable_positions(self, board):
        positions = []
        increments = Pawn.SPOT_INCREMENTS_MOVE if self._moved else
Pawn.SPOT_INCREMENTS_MOVE_FIRST
        for increment in increments:
            positions.append(board.spot_search_threat(self._position,
self._color, increment[0], increment[1] if self.color == Piece.WHITE else (-1)
* increment[1], free_only=True))

        increments = Pawn.SPOT_INCREMENTS_TAKE
        for increment in increments:
            positions.append(board.spot_search_threat(self._position,
self._color, increment[0], increment[1] if self.color == Piece.WHITE else (-1)
* increment[1], threat_only=True))

        positions = [x for x in positions if x is not None]
        return positions

    def move(self, target_position):
        self._moved = True
        Piece.move(self, target_position)

    def _symbol_impl(self):
        return 'PA'
```

Move: To encapsulate a chess move:

```python
class ChessPosition:
    def __init__(self, x_coord, y_coord):
        self.x_coord = x_coord
        self.y_coord = y_coord

    def __str__(self):
        return chr(ord("a") + self.x_coord) + str(self.y_coord + 1)

    def __eq__(self, other):
        return self.x_coord == other.x_coord and self.y_coord == other.y_coord

    @staticmethod
    def from_string(string: str):
        return ChessPosition(ord(string[0]) - ord("a"), int(string[1:]) - 1)


class MoveCommand:
    def __init__(self, src: ChessPosition, dst: ChessPosition):
        self.src = src
        self.dst = dst

    @staticmethod
    def from_string(string: str):
        tokens = string.split(" ")
        if len(tokens) != 2:
            return None
        src = ChessPosition.from_string(tokens[0])
        dst = ChessPosition.from_string(tokens[1])
        if src is None or dst is None:
            return None
        return MoveCommand(src, dst)
```

Game: To encapsulate a chess game:

```
from copy import deepcopy
from .pieces import Piece
from .render import *
from .moves import *
from .board import ChessBoard


class ChessGameState:
    def __init__(self, pieces, board_size):
        self.pieces = pieces
        self.board_size = board_size


class ChessGame:
    STATUS_WHITE_MOVE = "white_move"
    STATUS_BLACK_MOVE = "black_move"
    STATUS_WHITE_VICTORY = "white_victory"
    STATUS_BLACK_VICTORY = "black_victory"

    def __init__(self, renderer: InputRender = None):
        self._finished = False
        self._board = ChessBoard()
        self._renderer = renderer
        self._status = ChessGame.STATUS_WHITE_MOVE

    def run(self):
        self._renderer.render(self.get_game_state())
        while not self._finished:
            command = self._parse_command()
            if command is None and self._renderer is not None:
                self._renderer.print_line("Invalid command, please re-enter.")
                continue
            if not self._try_move(command):
                self._renderer.print_line("Invalid command, please re-enter.")
                continue

            self._board.execute_move(command)
            if self._status == ChessGame.STATUS_WHITE_MOVE:
                self._status = ChessGame.STATUS_BLACK_MOVE
            elif self._status == ChessGame.STATUS_BLACK_MOVE:
                self._status = ChessGame.STATUS_WHITE_MOVE
            self._renderer.render(self.get_game_state())

    def _try_move(self, command: MoveCommand):
        board_copy = deepcopy(self._board)
        src_piece = board_copy.get_piece(command.src)
        if src_piece is None:
            return False
        if (self._status == ChessGame.STATUS_WHITE_MOVE and src_piece.color ==
Piece.BLACK) or \
                (self._status == ChessGame.STATUS_BLACK_MOVE and
src_piece.color == Piece.WHITE):
            return False
        if command.dst not in src_piece.get_moveable_positions(board_copy) and
\
                command.dst not in
src_piece.get_threatened_positions(board_copy):
            return False
        board_copy.execute_move(command)
        for piece in board_copy.pieces:
            if self._status == ChessGame.STATUS_WHITE_MOVE and \
                    board_copy.white_king_position in
piece.get_threatened_positions(board_copy):
                return False
            elif self._status == ChessGame.STATUS_BLACK_MOVE and \
                    board_copy.black_king_position in
piece.get_threatened_positions(board_copy):
                return False
        return True

    def _parse_command(self):
        input_ = input()
        return MoveCommand.from_string(input_)

    def get_game_state(self):
        return ChessGameState(self._board.pieces, self._board.size)
```

Render: To encapsulate a chess render:

```python
from .moves import ChessPosition


class InputRender:
    def render(self, game_state):
        raise NotImplementedError

    def print_line(self, string):
        raise NotImplementedError


class ConsoleRender(InputRender):
    def render(self, game):
        for i in reversed(range(0, game.board_size)):
            self._draw_board_line(i, game.pieces, game.board_size)
        self._draw_bottom_line(game.board_size)

    def print_line(self, string):
        print(string)

    def _draw_time_line(self, countdown_white, countdown_black):
        print("Time remaining: {}s W / B {}s".format(countdown_white,
countdown_black))

    def _draw_board_line(self, line_number, pieces, board_size):
        empty_square = "  "
        white_square_prefix = "\u001b[47m"
        black_square_prefix = "\u001b[40m"
        reset_suffix = "\u001b[0m"
        black_first_offset = line_number % 2

        legend = "{:<2} ".format(line_number + 1)
        print(legend, end='')
        for i in range(0, board_size):
            is_black = (i + black_first_offset) % 2
            prefix = black_square_prefix if is_black else white_square_prefix
            contents = empty_square
            curr_position = ChessPosition(i, line_number)
            for piece in pieces:
                if curr_position == piece.position:
                    contents = piece.symbol()
            square_str = prefix + contents + reset_suffix
            print(square_str, end='')
        print()

    def _draw_bottom_line(self, board_size):
        vertical_legend_offset = 3
        line = " " * vertical_legend_offset
        for i in range(0, board_size):
            line += chr(ord("a") + i)
        print(line)
```

Player: To encapsulate a chess player:

```
from .render import ConsoleRender
from .game import ChessGame


class Player:
    def play_chess(self):
        render = ConsoleRender()
        game = ChessGame(render)
        game.run()


if __name__ == "__main__":
    player = Player
    player.play_chess()
```