```
# quarto_state.py
# Sean Straw & Ari Cohen
from quarto_interface import *  # maybe split Board into logical info and display
    info?

class GameStatus():
    PLAYING = 1
    QUITTING = 2
    TIE = 3
    WIN = 4
    LOSS = 5

class MoveStatus():
    LEGAL_MOVE = 1
    ILLEGAL_MOVE = 2

# A move consists of two parts
# 1) placing the current piece into an empty square on the board
# 2) selecting a new piece to be the current piece to place
class GameMove():
    LINES = [[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15],
             [0,4,8,12],[1,5,9,13],[2,6,10,14],[3,7,11,15],
             [0,5,10,15],[3,6,9,12]]
    def __init__(self):
        self.row_placement = -1
        self.col_placement = -1
        self.piece = -1

    def get_row_placement(self):
        return self.row_placement

    def set_row_placement(self, row):
        self.row_placement = row

    def get_col_placement(self):
        return self.col_placement

    def set_col_placement(self, col):
        self.col_placment = col

    def get_piece(self):
        return self.piece

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

    def set_move(self, row, col, new_piece):
        self.row_placement = row
        self.col_placement = col
        self.piece = new_piece


class GameState():
    AVAILABLE = 1
    UNAVAILABLE = 0
    EMPTY = -1
```

```
    def __init__(self, interface_state):
        # arbitrarily pick piece 15 for first move, it doesn't matter which one
            is used
        self.available_pieces = [GameState.AVAILABLE]*15 + [GameState.UNAVAILABLE
            ]
        self.squares = [GameState.EMPTY]*16
        self.current_piece = 15
        self.interface_state = interface_state

    def reset(self):
        # arbitrarily pick piece 15 for first move, it doesn't matter which one
            is used
        self.available_pieces = [GameState.AVAILABLE]*15 + [GameState.UNAVAILABLE
            ]
        self.squares = [GameState.EMPTY]*16
        self.current_piece = 15

    def get_available_pieces(self):
        return self.available_pieces

    def remove_available_piece(self, index):
        self.available_pieces[index] = GameState.UNAVAILABLE

    def add_available_piece(self, index):
        self.available_piece[index] = GameState.AVAILABLE

    def get_current_piece(self):
        return self.current_piece

    def set_current_piece(self, new_piece):
        self.current_piece = new_piece

    def get_squares(self):
        return self.squares

    def get_square_piece(self, row, col):
        return self.squares[4*row + col]

    def set_square_piece(self, row, col, new_piece):
        self.squares[4*row + col] = new_piece

    def make_move(self, move):
        self.set_square_piece(move.get_row_placement(),
                              move.get_col_placement(),
                              self.get_current_piece())
        new_piece = move.get_piece()
        self.set_current_piece(new_piece)
        self.available_pieces[new_piece] = GameState.UNAVAILABLE

    def get_interface(self):
        return self.interface_state

def check_pieces_for_win(p1, p2, p3, p4):
    if (p1 < 0) or (p2 < 0) or (p3 < 0) or (p4 < 0):
        return False
```

```
        return ((((p1^p2) | (p1^p3) | (p1^p4))^15) != 0)

    # This checks that the move is legal
    # 1) the placement needs to be in a currently empty square
    # 2) the selected piece needs to be available - unused at the moment
    #        - the one exception is if all pieces have been placed
    # If the move is legal, it then looks for a win, tie, or end of game condition
    def check_move(main_game_state, move):
        row = move.get_row_placement()
        col = move.get_col_placement()
        #create copy so we don't disrupt anything when testing moves
        game_state = copy_game_state(main_game_state)
        piece_to_move = game_state.get_current_piece()
        new_piece_to_move = move.get_piece()
        availability = game_state.get_available_pieces()
        if game_state.get_square_piece(row, col) != GameState.EMPTY:
            # moving into an occupied square, so complain
            return [MoveStatus.ILLEGAL_MOVE, GameStatus.PLAYING]
        # if a real piece is moved, make sure it is available
        if (new_piece_to_move != GameState.EMPTY) and (availability[new_piece_to_move
            ] == GameState.UNAVAILABLE):
            return [MoveStatus.ILLEGAL_MOVE, GameStatus.PLAYING]
        # See if there is a win - if so, don't check new_piece_to_move
        squares = game_state.get_squares()
        squares[4*row + col] = piece_to_move # make the move so we can analyze the
            board
        for places in GameMove.LINES:
            if check_pieces_for_win(squares[places[0]], squares[places[1]],
                                    squares[places[2]], squares[places[3]]):
                game_state.set_square_piece(row, col, GameState.EMPTY)
                return [MoveStatus.LEGAL_MOVE, GameStatus.WIN]
        # no win, so make sure new_piece_to_move is legitimate if it is EMPTY
        if new_piece_to_move == GameState.EMPTY:
            # check to see if all pieces are used up
            available_pieces = []
            for i in range(16):
                if availability[i] == GameState.AVAILABLE:
                    available_pieces.append(i)
            list_length = len(available_pieces)
            if list_length != 0:
                # pieces were available, so complain
                return [MoveStatus.ILLEGAL_MOVE, GameStatus.PLAYING]
            # this is legal, but there are no moves left - so it is a tie
            return [MoveStatus.LEGAL_MOVE, GameStatus.TIE]
        return [MoveStatus.LEGAL_MOVE, GameStatus.PLAYING]

    def copy_game_state(game_state):
        new_game_state = GameState(game_state.interface_state)
        new_game_state.set_current_piece(game_state.get_current_piece())
        piece_list = game_state.get_available_pieces()
        square_list = game_state.get_squares()
        for piece in range(16):
            if(piece_list[piece] == GameState.UNAVAILABLE):
                new_game_state.remove_available_piece(piece)
            if(square_list[piece] != GameState.EMPTY):
                new_game_state.set_square_piece(piece/4,piece%4,square_list[piece])
```

```
        return new_game_state
```