```
# quarto_player.py
# Sean Straw & Ari Cohen

import random  # used for creating random moves
from quarto_interface import *
from quarto_state import *
from quarto_network import *


class GamePlayer():
    HUMAN = 0
    COMPUTER = 1
    NETWORK_HOST = 2
    NETWORK_CLIENT = 3

    MAXIMIZE = 1
    MINIMIZE = -1

    def __init__(self):
        self.type = GamePlayer.HUMAN
        self.level = 0
        self.time_limit = 10000000

    def get_type(self):
        return self.type

    def set_level(self, level):
        self.level = level

    def game_over(self, game_status):
        if self.type == self.NETWORK_HOST:
            signal_host_game_over(self, game_status)
            self.connection.close()
        if self.type == self.NETWORK_CLIENT:
            signal_client_game_over(self, game_status)
    def set_type(self, new_type):
        self.type = new_type
        if self.type == GamePlayer.NETWORK_HOST:
            [self.HOST_IP, self.HOST_PORT] = get_host_information()
            self.connection = connect_to_host(self)
        elif self.type == GamePlayer.NETWORK_CLIENT:
            self.HOST_IP = '0.0.0.0'
            self.HOST_PORT = get_port_number()
            self.HOST_PORT = start_host_server(self)

    def get_move(self, game_state):
        if self.type == GamePlayer.HUMAN:
            return get_human_move(game_state)
        elif self.type == GamePlayer.NETWORK_HOST:
            return get_network_host_move(game_state, self.connection)
        elif self.type == GamePlayer.NETWORK_CLIENT:
            return get_network_client_move(game_state)
        else:
            return get_computer_move(game_state, self.level)

def get_players_info(player0, player1, interface_state):
```

```python
        players = [player0, player1]
        data = get_players_information(interface_state)
        for index in range(2):
            if data[index*2] == "h":
                players[index].set_type(GamePlayer.HUMAN)
            elif index == 0 and data[index*2] == 'n':
                players[index].set_type(GamePlayer.NETWORK_HOST)
            elif index == 1 and data[index*2] == 'n':
                players[index].set_type(GamePlayer.NETWORK_CLIENT)
            else:
                players[index].set_type(GamePlayer.COMPUTER)
                players[index].set_level(data[(index*2)+1])
        return

    def get_computer_move(game_state, level):
        #First move doesn't matter, might as well not keep user waiting...
        if (len(get_good_pieces_and_squares(game_state)[1]) == 16) or (level == 1):
            return get_random_move(game_state)

        good_moves = simple_move_test(game_state, GamePlayer.MAXIMIZE, 1, 1, [0, 0])
        if(len(good_moves) != 0):
            if (level == 2):
                return [random.choice(good_moves), GameStatus.PLAYING]
            elif (level == 3):
                #Here's where I'd do more with good_moves (look further)
                #For now just do the same as level 2
                return [random.choice(good_moves), GameStatus.PLAYING]
        else:
            #No possible moves that won't let opponent win
            return get_random_move(game_state)

        #depth = len(get_good_pieces_and_squares(game_state)[0])
        #return test_moves(game_state, depth, GamePlayer.MAXIMIZE, depth)

    def simple_move_test(game_state, minimax,
                         max_depth, depth_left, root_move):
        good_pieces, good_squares = get_good_pieces_and_squares(game_state)
        good_moves = []
        for piece in good_pieces:
            for square in good_squares:
                comp_game_state = copy_game_state(game_state)
                comp_move = GameMove()
                comp_move.set_move(square[0],square[1],piece)
                [placeholder, move_status] = check_move(comp_game_state, comp_move)

                #First time through the loop
                if (depth_left == max_depth):
                    if(move_status == GameStatus.WIN):
                        if (minimax == GamePlayer.MINIMIZE):
                            return GameStatus.WIN
                        else:
                            return [comp_move]
                    elif (minimax == GamePlayer.MAXIMIZE):
                        comp_game_state.make_move(comp_move)
                        opponent_moves = simple_move_test(comp_game_state,
                                                          GamePlayer.MINIMIZE,
                                                          max_depth, depth_left,
                                                          [comp_move, 0])
                        if(opponent_moves != GameStatus.WIN):
                            good_moves.append(comp_move)
                else:
                    pass

        if (minimax == GamePlayer.MINIMIZE):
            return GameStatus.PLAYING
        else:
            depth_left -= 1
            return good_moves

    def get_random_move(game_state):
        move = GameMove()
        good_pieces, good_squares = get_good_pieces_and_squares(game_state)
        chosen_square = random.choice(good_squares)
        if(len(good_pieces) != 0):
            chosen_piece = random.choice(good_pieces)
            move.set_move(chosen_square[0],chosen_square[1],chosen_piece)
            return [move, GameStatus.PLAYING]
        else:
            chosen_piece = -1
            move.set_move(chosen_square[0],chosen_square[1],chosen_piece)
            return [move, GameStatus.WIN]

    def get_good_pieces_and_squares(game_state):
        avail_pieces = game_state.available_pieces
        all_squares = game_state.squares
        good_pieces = []
        good_squares = []
        for piece in range(len(avail_pieces)):
            if(avail_pieces[piece]):
                good_pieces.append(piece)
            if(all_squares[piece] == GameState.EMPTY):
                good_squares.append([piece/4, piece%4])
        return good_pieces, good_squares
```