CSCI 3010 - Fall 2018 - Muzny - In-class Activities -- Lecture 9

Player.h

```
struct Position {
    int row;
    int col;

    bool operator==(const Position &other) {
        return row == other.row && col == other.col;
    }
};

class Player {
public:
    Player(const std::string name, const bool is_human);

    std::string get_name() const {return name_; }
    int get_points() const {return points_; }
    Position get_position() const {return pos_; }
    bool is_human() const {return is_human_; }

    void ChangePoints(const int x);

    void SetPosition(Position pos);

    std::string ToRelativePosition(Position other);

    std::string Stringify();
private:
    std::string name_;
    int points_;
    Position pos_;
    bool is_human_;
}; // class Player
```

Maze.h

```
enum class SquareType { Wall, Exit, Empty, Human, Enemy,
Treasure };

std::string SquareTypeStringify(SquareType sq);

class Board {
public:
    Board();

    int get_rows() const {return 4; }
    int get_cols() const {return 4; }

    SquareType get_square_value(Position pos) const;

    void SetSquareValue(Position pos, SquareType value);

    std::vector<Position> GetMoves(Player *p);

    bool MovePlayer(Player *p, Position pos);

    SquareType GetExitOccupant();

private:
    SquareType arr_[4][4];

    int rows_; // might be convenient but not necessary
    int cols_;

    // you may add more fields, as needed
};  // class Board

class Maze {
public:
    Maze(); // constructor

    void NewGame(Player *human, const int enemies);

    void TakeTurn(Player *p);

    Player * GetNextPlayer();

    bool IsGameOver();

    std::string GenerateReport();

private:
    Board *board_;
    std::vector<Player *> players_;
    int turn_count_;
};  // class Maze
```

CSCI 3010 - Fall 2018 - Muzny - In-class Activities -- Lecture 9

1) (1 point) Annotating Player.h and Maze.h:
   a) Draw a square around the definitions of the constructors for the Player, Board, and Maze objects.
   b) Draw a circle around the fields for the Player, Board, and Maze objects.
   c) Indicate any methods that you think should not be public (and tell us why).

2) (3 points) Critiquing the design of the “Maze” - game
   a) Methods: should do 1 thing and do it well and they should avoid long parameter lists and lots of boolean flags. Which, if any, methods does your group think are not designed well? For each one, write down why you think it isn’t designed well. Is there a method that you think is designed well? Why?

   b) Fields: should be part of the inherent internal state of the object, their values should be meaningful through the object’s life, and their state should persist longer than any one method. Which, if any, fields does your group think should not be fields? For each one, write down why you think it shouldn’t be. Is there a field that you think definitely should be a field? Why?

   c) Objects/classes: objects should be cohesive (one single abstraction), complete (provide a complete interface), clear (the interface should make sense), convenient (should make things simpler in the long run), and consistent (names, parameters/returns, ordering of parameters, behavior should be consistent). Are there objects that you think are designed poorly? How so? What specifically would you change?

Trait	Player	Board	Maze
cohesive			
complete			
clear			
convenient			
consistent			

3) (1 points) Re-design the “Maze” game. You don’t need to write compilable code, but you should indicate what objects you would have, what methods they would each have, and what fields they would have. Be sure to highlight differences between your maze game design and the one that we used for homework 1.