Felipe Lima - 109290055

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();


	 friend std::ostream& operator<<(std::ostream& os, const Board
&b);

private:
	SquareType arr_[4][4];

	int rows_;
	int cols_;

};  // 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();
	friend std::ostream& operator<<(std::ostream& os, const Maze
&m);

private:
	Board *board_;
	std::vector<Player *> players_;
	int turn_count_;

};  // class Maze
```

1) Annotating Player.h and Maze.h:

a) Draw a square around the constructors for the Player, Board, and Maze objects.

b) Draw a circle around the fields (class attributes) for the Player, Board, and Maze objects.

c) Underline any methods that you think should not be public. (Briefly) Explain why you think that they should not be public.

2) Critiquing the design of the "maze" game:

a) Methods: should do 1 thing and do it well. They should avoid long parameter lists and lots of boolean flags. Which, if any, methods does your group think are not designed well? Is there a method that you think is a good example of being well-designed? which?

Our group decided all the methods were designed fairly well.

Good examples: for the most part all of them are good exemples.
void ChangePoints(const int x);

b) Fields: should be part of the inherent internal state of the object. Their values should be meaningful throughout 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? Why not? What is an example of a field that definitely should be a field? why?

Should not be fields: Board class: rows_ and cols_

Examples of fields that should be a field is “name_” and “points_” in Player

c) Fill in the following table. Briefly justify whether or not you think that a class fulfills the given trait.

Trait	Player	Board	Maze
cohesive (one single abstraction)	Yes fulfills the given trait. Does one thing.	Could be better. Should focus only on the creation of the board	Yes fulfills the given trait. Although it should be called “Game”
complete (provides a complete interface)	Yes fulfills the given trait. Everything a player needs that is related only to the player is in this class	Yes fulfills the given trait. Everything a player needs that is related only to the player is in this class	Yes fulfills the given trait. Everything a player needs that is related only to the player is in this class
clear (the interface makes sense)	Yes fulfills the given trait.	Yes fulfills the given trait.	Yes fulfills the given trait.
convenient (makes things simpler in the long run)	Yes fulfills the given trait. If anything would be added in the future it would be fairly easy.	Yes fulfills the given trait. If anything would be added in the future it would be fairly easy.	Yes fulfills the given trait. If anything would be added in the future it would be fairly easy.
consistent (names, parameters, ordering, behavior should be consistent)	Yes fulfills the given trait.	Yes fulfills the given trait.	Yes fulfills the given trait.