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.

Altering values shouldn't be accesses from any given state

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?

The take turn method is not designed well.

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?

Int rows and int cols are unnecessary fields. This is because the array needed a specific size. A fields that are necessary are x and y coordinates.

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)	No, there are several elements to the player class and it is the least abstract.	Yes	Yes, there are only two elements to the Maze feature, stringify and the type values
complete (provides a complete interface)	Yes, has plenty of features	Yes	No there is more that can be done
clear (the interface makes sense)	There are plenty of straight forward elements	No, there are functions that are unnecessary	Yes, there are small amount of features
convenient (makes things simpler in the long run)	Yes, all methods are easily accessable	Yes, breaks things down	No, could be broken down into more elements
consistent (names, parameters, ordering, behavior should be consistent)	Yes	Yes	No there could be more stored in here vs other classes