Yun Zhou 300442776

Reflection

Assumptions

In order to make the implementation part better(i.e. code part), I read the specification carefully and find some ambiguous points that need to make assumptions based on that, these assumption points are shown below.

- The game is played by using **only one device**, that means players need to take turns to use the device to play the Cluedo, unlike the original game, it isn't flexible.
- The player will be represented as a **single Char** on the board, which means the player needs to identify the character through it, the weapon also use this assumption.
- For saving the cost, the initial position of each player is fixed include those characters that are not be used. That is, every character will be born at the **fixed locations** on the board even some of them are not be used.
- Room is pretty complex as it consists of many cells of locations, but it is treated as a single big location when the player tries to move in or move out of the room. Howsoever, the capacity of each room should be able to hold all players and weapons.
- The Room should have several entrance cells for players to move in/out. These entrance cells are away from the player starting cells. I observe this by the diagram from the specification of the assignment.
- The assumption of the player can't visit the same cell in one turn should also apply to Room cell as it's a single big cell when the player tries to move in/out. In other words, if the player tries to move out of the room at the same turn, he can't go out through the cell that he entered as it's already been visited.
- The player can make the accusation at the end of any turn. However, the player can only make the accusation once, it can either be a success(win) or failure(can only refute suggestions in this case).
- The player can only make a suggestion when the player is in the room and it's at the end of the turn(the player can choose end turn to make a suggestion).
- The possible solution is made when no one can refute the suggestion, but players can still accuse wrong cards. There is no restriction on that.
- The killer doesn't know his identity, every one must figure it out through suggestions.

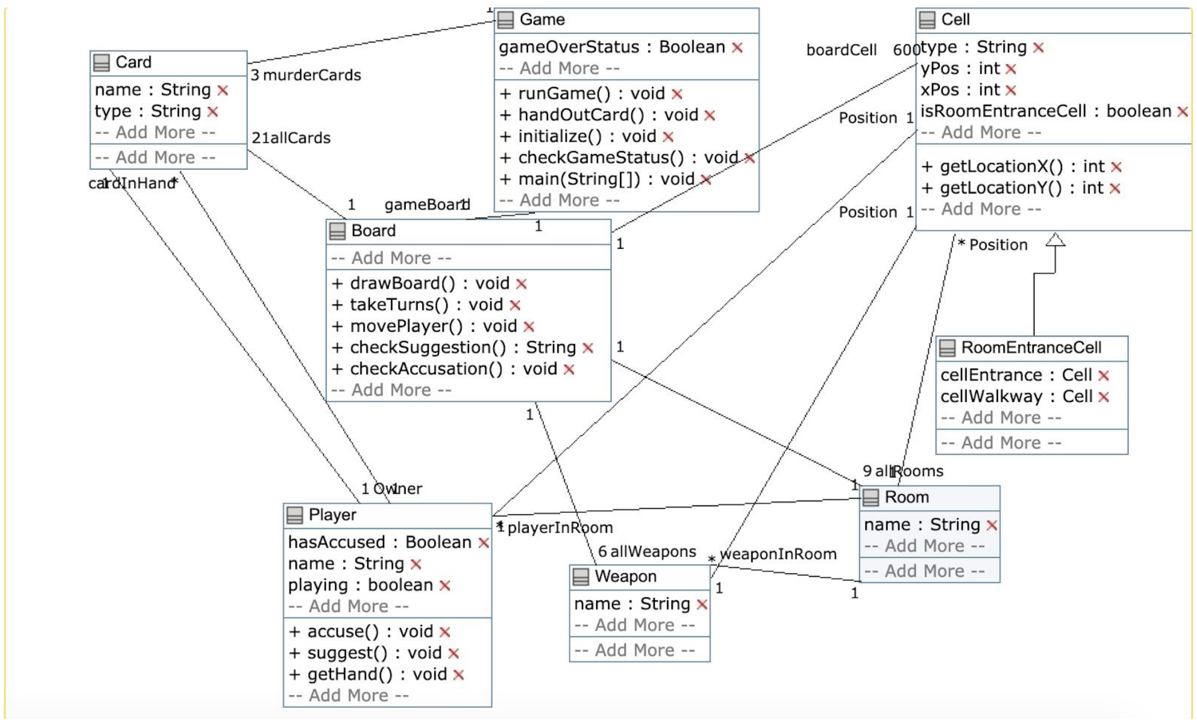
CRC Cards

CRC card is a high-level design and description which describes the purpose of each class and how they associate and collaborate. Based on the assumptions and the analysation of the requirement specification, we made the CRC cards together, it is useful and helps us a lot. It is useful because:

1) It teaches us how to work efficiently, especially group work. As it is at the highest level, we can easily change the idea and communicate with each other in order to make the design of cards better.

2) The responsibility part helps us to figure out what functionality should the class be implemented.
3) The collaboration part helps us to figure out what classes are associated with the current class. It’s useful when making the UML. Also, it provides classes we need in order to implement code functions.
    a) E.g. Player class needs to know how many Card instances they have in the **hand**. We can easily make the UML based on this.
4) It saves lots of time in the following steps as it’s pretty concise and straight forward. We will not be confused when we are not sure what to do next, because CRC cards can be observed by us to see whether this function is implemented or not. It’s pretty useful when we are making the UML as well as the coding part.
    a) E.g. For the Board Class, we know what to do by just looking at the CRC Cards.
5) CRC cards provide us with a general frame by summarizing requirements and assumptions. It gives us a direction to implement the function. It’s good but mistakes can be made and need to change it later in order to make the design perfect and complete.

Game Logic



UML is made based on the CRC cards, it sublime and implements the game logic.
Below is the description for each Class.
Game: It initialises everything in order for players to play the game. It includes set up everything like **Cell**s for a **Board** class in order to use the Board object to *draw*. Before each game start, it will ask the number of **Players** who are involved and hand out **Cards** evenly to each **Player** after random generate 3 murder **Card** which contains **Room**, **Weapon** and **Player**. After it’s been done, a map of **Board** will be constructed and then the run_Game method will be called and will not stop until the gameOverstatus is be reached(i.e. == true).

Yun Zhou 300442776

Board: it contains lots of information which includes: Lists of allWeapons, playingPlayers, murderCards, **RoomEntranceCell** and so on. It has a method for drawing boards and a method for **players** to play in turns. It will use the redraw() method to apply all changes during the **Game,** such as Player moves. It also contains the check methods for checking accusation and suggestion from **Player.** It will help to check the PlayingPlayerList all the time in run_Game() method. The game is over if the PlayingPlayerList is 0 which means every **Player** made a wrong accusation OR **Player** accuse the correct murder **Card**s by guessing through suggestions.
Player: contains the nickname of the player and the role name. Need to know one location **Cell.** It has hand **Cards** to accuse and suggest. These fields/function got the corresponding methods. All these methods will be used during the game which is the check methods on **Board**. The Player will be moved out of the PlayingPlayerList if the wrong accusation is made.
Card: contains the name and the type. The type can be **Room, Weapon, Player** and it can be owned by **Player** to refute the suggestion and check the accusation and so on.
Room: contains the name of the room and it also contains lots of **Cells. Players** and **Weapons** can stay in the Room and **Player** can make suggestions in it.
Cell: Used for drawing and determine the positions on the **Board**. It is used for **Player, Weapon, Room.** Also, it's related to **RoomEntranceCell** class.
RoomEntranceCell: It determines the Cell position for entering the **Room.**
Weapon: it has a weapon name and knows where is the location of the **Cell** on the **Board**.

Contributions and Challenges:

As our third teammate(Francis) is missing, we only have two guys to do the whole assignment, our contributions are the same. First, I make the first edition of CRC cards, my teammate Ruiyang thought some part could be changed, so we change opinion and make a discussion on that and finally construct 8 cards which means 8 classes. For the UML part, Ruiyang is responsible for that, but we also communicate with each other when CRC needs to be upgraded as the special relationship between UML and CRC. After the coding part is done, for consistency, we upgraded CRC and UML together.
For the coding part contribution, except the simple Class that is auto-generated, the remaining important parts we are about half and half. I am responsible for implementing several functions of game initialization, such as implement the GUI class which is responsible for asking how many players are involved in and let them choose the character to play. Also, I implemented fields and methods like randomly generate the murder cards and hand out cards evenly to the players. For the main method which is the game constructor method, as it is an overall view of summarizing all the objects and methods, this part we definitely work together.
Also, we've met lots of challenges. For example, Board class and Game class are intertwined and ambiguous, we can't always figure out the logic clearly. To make our life easier, we swapped some methods of classes that should stay in the original class. For instance, implements the takeTurn() method is the responsibility of the Game class, but we moved it into the Board class. This is a smart idea but the code looks too messy as we add up these methods at both classes. We are now afraid to delete the useless code as this may cause an unexpected error.

Yun Zhou 300442776

Satisfaction

In conclusion, I am satisfied with our work but there are still some points that can be upgraded. For example, the number of classes can be merged, code is not concise enough and the realization code of some functions should upgrade. For instance, the relationship of Player class and Card class is bidirectional, the Player needs to know Cards they have and Cards need to know their owner. I implement this by calling the add functions twice each time, it’s not good obviously. Also, the assumptions of Room is not implemented perfectly, not satisfied with it.