A5 YANG LIN & YU YUXIAO

Overview of implementation

Firstly, we use a MVC model to implement the game. There are 3 main parts of our implementation:

Model: Player Board Square

View: Display

Controller: Controller

In the Controller part, we initialize a board, a display, a array of players and other needed tools in the filed. Through the controller, we could read command and execute the command like roll, next, trade, improve, mortgage, save, load etc, and then controller will notify relevant player and square to take action. As the change happened, the controller will notify Display to update the display. So, in this part, we use controller to notify players, board and player making change in such a way.

In the Model part, we implement an array of squares in Board to store information of squares. When we make a move, the board will notify the specific square, and then the square will give relevant reaction to players landing on it. In details, for the square part, we create two subclasses building and non-property using inheritance. Building is an abstract class and inherited by academic buildings, residences and gyms. Non-property is another abstract class inherited by other buildings with special features, such as SLC, Needles Hall, Dc Tims and so on.

In the View part, we implement a display class with a 2D array of char to print our

board. When any change need to be made, model will notify controller and then controller will notify display to make change of the 2D array.
To implement the improvement of academic buildings, we use a decorator pattern to store the implementing information. Thus, we could easily check if the player is monopoly in the block and make improvement of the academic buildings.

To implement the only four RURC card, we use a singleton to implement a RURC class, which could be created once in controller.

To implement the SLC and Needles Hall, we use a factory pattern to implement an abstract class to have a pure virtual method to generate a card.

Players store every piece of useful information for him/herself. That includes array of academics, residences and gyms that he/she contains. Therefore, with the control of Controller, player, block, board and square can work as a whole and lead to the final game.

Additionally, we use private, protected and public fields appropriately. As private and protected fields can limit accessibility to the class, they make the field “safer”. We use private fields as much as possible with proper use of public field.

Finally, for memory management, we new 40 squares with relevant blocks, players and the cards in heap. Through the game, we keep them unless they should be “popped off”(like a player declares bankruptcy). We choose to use pointers, so it is convenient and we don’t have to make a copy for object itself. Overview the whole memory, there is only 40 squares, an array of pointer to players and four RURC card in the heap, which makes it clear and easy to make memory management and avoid memory leak.

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

(1) Testing can improve the correctness of our code. It could help us defense all kind of request and input.

(2) Design a structure and decide what design pattern is vital before start code.

(3) Before finish all .cc implementation, we could implement all .h file.

(4) Corporation is important for large programs, as they are involved in a number of classes and is difficult for one person to finish. And since different people have different on problems, so communicating is significant. Collecting various ideas can contribute a lot to improve a program.

(5) Even when we need to finish a large program by ourselves, we can also implement the process as we do in a team. Analyzing the problem as a whole and then dividing, and finally collecting them again can be a good habit.

2. What would you have done differently if you had the chance to start over?

(1) Read all request first and think for the design carefully. The worst thing is that we found we didn't truly understand the game and find everything have to be start over after doing some useless work.

(2) Regard the "huge" game as a whole first, and then divide them into some part to implement. We can't start from some part at start, as for large program, every file should be connected in some way. From some side-part, we always found that we missed something when we implement a new class.

(3) Started as early as possible. This is really time-consuming, and we should know plans never catch changes.