Final Design Document

BB7K

q65liu & s275wu

Part I: Description of Classes

1. Board
   It is the heart of the game, it contains the main command loop, implements the main logic of the game and does all the board initialization job. It has a "owns a" relationship with Player class, TextDisplay class and Square class.
2. Player
   It implements the player part of the game. It contains everything needed for a player, its savings, its name, its char representation on the board, how many gyms it owns, etc. It has a "has a" relationship with Square class because each player has a pointer pointing to its current location square.
3. TextDisplay
   It is the visual display part of the game, which contains the fun text display of the board, the beginning greeting and ending message. Every time a player changes its location or player buy/sell improvement, Board will call notify() method to update the textdisplay. One notify is for improvement and one is for player. There are several map which helps the display. Also, we generated a lot of fun images (for example, dice, scroll, coffee, geese, train) which will be shown throughout the entire game.
4. Square
   It is the Square class, which is the parent class of OwnableProperty and UnownableProperty. Since there are a lot of pure virtual methods, so it is an abstract class. It contains a pointer pointing to the gameboard.
5. OwnableProperty
   It is one of the subclasses of Square, and the parent class of Residence, Gym and AcademicBuilding. It is also an abstract methods because it contains pure virtual methods and leaves some pure virtual methods from Square class unimplemented.
6. Residence/Gym
   Two of the sub classes of OwnableProperty. The methods implemented here are very similar. Both are concrete classes.
7. AcademicBuilding
   It is one of the subclasses of OwnableProperty. Since it can have improvements and monopoly block, so it has more methods to implement and more fields. It is a concrete class.
8. UnownableProperty
   It is one of the subclasses of Square, and the parent class of Slc, NeedlesHall, Dctims, CoopFee, CollectOsap, GoToTims, GooseNesting and Tuition. It is also an abstract methods because it contains pure virtual methods and leaves some pure virtual methods from Square class unimplemented.

9. Slc/NeedlesHall/DcTims/CoopFee/CollectOsap/GoToTims/GooseNesting/Tuition
   All are subclasses of UnownableProperty. All are concrete. The action method are implemented differently here.

Part II: Description of Aspects

1. <roll>: for roll command, we have a method roll in Board which returns an int, instead of generating 2 dice results in the roll method. We make one roll method generating only one dice and return the side of that dice. So it would be easier to do the comparison between dices. The roll method for testing mode is just asking for input.
2. Purchase property: The tuition is handled through several if statements, for example, if has 1 gym, then pay this much, if has 2 gyms, pay that amount, etc. If decline to purchase, then auction begins on that property.
3. <quit>: quit command is similar to asset command, should be useful while user is typing command such as roll, next… However, if are doing actions in squares, such as calling for options for tuition square, quit won't work. So does asset.
4. Ctrl-D: eof is handled throughout the program. It shouldn't be any SegFault due to Ctrl-D.
5. Monopoly: If DC and MC both belongs to player A, which we say there is a monopoly. When player B passed DC, B needs to pay double tuition to A since the monopoly is owned by A. We actually do the paying twice, so the msg will look like "B pays A ---dollars. B pays A ---dollars".
6. Random generating: for rolling dice in non-testing mode, generating rim cups, and also generating outcomes in slc and needles hall square. <cstdlib>'s rand() is used. Also, in the beginning of the program, srand(time(0)) is called in order to provide a seed for rand().
7. Rim Cups: In order to make sure there are no more than 4 cups in the game, we used what we said in due date 1 document. In board, we have a counter for total rim cups, if any player gains a cup, we check if # of cups =4 yet and if not, we increment that counter. If any player uses a cup, we decrement the counter.
8. Dummie Methods: As shown in the UML, the Square class has a lot of pure virtual methods, for example: virtual void setMortgage(bool mortgageornot)=0; virtual void mortgage(Player & owner)=0; virtual bool getAcBuilding()=0; virtual void setMonoBlock(std::string mb)=0; virtual int getImprovement()=0; virtual void improve(Player &p)=0; virtual void sellimprove(Player& p)=0;
   The reason why we put those in the Square class is because in Board, we used an array of 40 Square* called squareblock to keep track of each square, and in our init() in Board, we initialize the array like below:
   squareblock[0]=new CollectOsap("COLLECTOSAP");
   squareblock[1]=new AcademicBuilding("AL","Arts1",40,50,2,10,30,90,160,250);
   squareblock[2]=new Slc("SLCBOTTOM");
   ….
   squareblock[39]=new AcademicBuilding("DC","Math",400,200,50,200,600,1400,1700,2000);
   Later when we were implementing, the compiler complained that Square doesn't have a lot of

the methods implemented in AcademicBuilding or Gym. So we left to 2 options: rewrite the code or hack this out. We don't have enough time to redo the code so we added those methods to Square as pure virtual. However, if this doesn't apply to that class, we make those methods private. For example, void mortgage(Player &p) should not work for any of the UnownableProperty, therefore in UnownableProperty, we make this method private and it is an empty method which does nothing. Also, for void improve(Player &p), which Gym and Residence both cannot implement because they cannot be improved, we make this method private too in those two classes and it is also an empty method which does nothing. It turns out solved the problem.

9. Command line args: We added a if else statement to make sure command line args can be given in any order. In board, we have 2 bool for testing and loading, and both have get methods. In main, If(argc==2&&strcmp(argv[1], "-testing") ==0){setTesting(true); setLoading(true);} And there are more else if. For loading, we used ifstream, checked if the file is readable, if cannot be opened, a message will be printed. In order to make loading more easily, we make the ctor of player convenient for loading,

   Player(string name, char namec, int cups ,int savings, int loc, bool inline, int lineturn, Board*gb);

   There is a method in Board called load(string filename) which will load from input file.

   There is a if statement check if the location is 10, if not, it is not in line and lineturn will be 0.

   If yes, another if to check if is inline, if not in line, then inline will be false and lineturn will be 0. If in line, then inline will be true and lineturn will be taken.

   For loading buildings, we just call set methods for setOwner(), setImprovement(). We check if it is a gym or residence, if not, then setMortgage() if there is a -1.

   For testing, we just use cin to take inputs from user.

10. <save>: We used ofstream to create a file with given name, and then write to it. It is basically go through the array of Player* and array of Square* in Board and then call their get methods for private fields. Also, we used if statements to check if a player is in DcTims, if so, check again if it is in line or not, if so, then need to write lineturn too. For buildings, we used if statements to check if building has an owner, if no, then write BANK, then if the building is mortgaged, then we write -1.
11. <asset>: It should print savings, total worth, properties in monopoly form, print # of cups owned. This cannot work when player is deciding in Tuition square.
12. Owe money in auction: If a player wins in the auction in his turn but he or she does not have enough savings to pay, he or she has to immediately pay back all the amount owed to the bank or declare bankrupt. However, when a player(player A) wins in the auction in another player's(player B) turn and the winner(A) in auction does not have enough savings to pay, it is still player b's turn to call the command. Player A will pay back the money he or she owed in the auction, or declare bankrupt in his or her turn. Player A cannot call roll before paying back.
13. <trading> : check if each player has the corresponding property, check if it is trading money for money, check property has improvements: In each academicbuilding class, there is a private field, improvements, representing the improvements level. The method to get this field(getimprovements) can tell the improvements level of a academic building.

    swap:

1) money for property: Player A -money Player B + money → current owner of property becomes A.

2) propert1 for property2: current owner of property 1 becomes B -> current owner of property 2 becomes A

3) property for money: Player A + money Player B – money -> Current owner of property becomes B

14. Check property is monopoly: In each academic building class, there is a private filed, currentowner, which is a pointer to a Plyer representing the current owner of the building. If currentowner of the academic buildings in the same monopoly block points to the same player, they are monopoly.
15. <mortgage> : In each academicbuilding class, there is a private field, improvements, representing the improvements level. The method to get this field(getImprovements()) can tell the improvements level of a academic building. If it is grater than 0, it cannot be mortgaged
16. <unmortgage> :
17. Sent to Tims Line: There is a boolean field called intimsline. It is true when a player is sent to DC time line.Another integer field called intimsline indicating the times the player has been in the DC Tims line when intimsline is true. If intimsline of a player is true, he or she can just go out of DC Tims when rolling doubles. The option that if he or she wants to pay $50 or use a cup will be given. If the player choose to use a cup, it will be checked that he or she has enough cup to use. If so, he or she will get out of DC Tims and the number of cups will minus one. Also, if the player does not have $50 when paying $50 is chosen, he or she needs to gets money to pay back to bank or declare bankrupt. If the player stays for less than three turns, he or she can go next and wait. However, in the third turn, he or she has to make a decision.
18. Detect winner and ends the game: there is an array of Player pointers in the board. When a player bankrupts, the corresponding position in the array will be deleted and set to null. Every time when a player declares bankrupt, it will be checked if there is only one non-null Player pointer in the array. If so, the only player left wins the game and the winning message will be printed.
19. Auction: We have 2 int arrays called stillinauction and quitauiction and an int called quitnum. Every time a player draws, the quitnum will increase one and quiauction[quitnum] will be the index of current player in auction and stillinauction[currentauction] will be -1. So, when the quitnum is equal to the total player minus 1. There will be a winner in auction and find the player index such as stillinauction[index] != -1. That player will be the winner.
20. Tuition with monopoly : If monopoly is owned and property has no improvement. We first let call the action() in that square which deducts the tuition once, if there is a monopoly and this building has no improvement, we call the action again which will deduct the tuition for the second time.
21. SLC/Needles Hall: We first call rand() to generate a number from 0 to 99, then if the number is 50, which is 1% chance, then we will give the player a cup (if there are <4 cups). Then it will call rand() again to generate numbers, and do certain tasks.

Part III: Answers to questions (1-4 reanswer due-date 1, 5-6 wrap up questions)

Q1: After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a gameboard? Why or why not?

A1: Observer Pattern would be a good one but we didn't use it in our code. Instead, we just have several notify methods in TextDisplay to keep TextDisplay updated. For example, every time a player moves, it will call notify(string playercharname, string location). There is a map in TextDisplay which handles that. Also, whenever player bankrupts, it calls notifyremove(playername) so the player map in TextDisplay will erase that player. Whenever a improvement is bought or sold, it will call notify(string location, int improvement) in order to modify the improvement map in the TextDisplay. In this case, no patterns is necessary but gameboard is still implemented.

Q2: Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?`

A2: We didn't model SLC and Needles Hall closely to Chance and Community Cards. Because we think just a big if loop is simple enough for all the outcomes. However, if we really need to model SLC and Needles Hall using this way, we still think using design pattern will be too complicated. And would be the same answer we provided for due date 1: It will be much simpler if we have a Card class, and it will contain fields such as forwardmoves, backwardmoves, moneylost, moneygain… and in SLC and NeedlesHall we can set up an array of 52 Cards and each does different actions such as move back 3 steps, step forward 2 steps, lose 100 dollars, etc. And player will generating a number from 0 to 51, which will access the cards from that index.

Q3: What could you do to ensure there are never more than 4 Roll Up the Rim cups?

A3: Same answer to Due Date 1. In Board class, we have an private field which is a integer called rimscuptotal to keep track of the total number of Roll up and Rim cups existing in the game. When the game is initialized, rimscuptotal will be set to 0. Every time a player wins a Roll up and Rim Cup, It will call getCuptotal() to check if there are already 4 cups now or not. If not, that player will get 1 cup and the rimscuptotal will be incremented by 1 by calling setCuptotal(int cupamount). Whenever a cup is used, rimscuptotal will be decremented by calling setCuptotal(getCuptotal()-1). This way, we can makre sure there are never more than 4 cups.

Q4: Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

A4: As Due Date 1 answer, we didn't use Decorator Pattern to implement improvements because implementing it and maintaining it requires too much work. There are 2 parts of improvements: money part and visual representation on the board. Both parts are achievable if we use decorator pattern but it would be unnecessarily complicated. On due date 1, we thought Strategy Pattern will be much more straightforward. However we went for the even simpler way we metioned which is just having methods in AcademicBuilding such as sellimprove(Player & p), improve(Player &p), getImprovement() which returns the improvement level and setImprovement() which sets the improvement level. And in TextDisplay we implemented a notify method which would be called every time improvement level changes and then a print improvement method which takes the

improvement level integer and then prints that many I on the board.

Q5: What lessons did this project teach you about developing software in teams?

A5: Communications and daily meet up is very necessary. In the beginning we developed our parts independently. When we first came together and merge our code, we spent a long time changing variable names and methods calling names. Later on, we basically code together every day so this issue didn't appear any more. Also, we learnt to show our strength. Liu is good at logic and I am good at display and class relationship, so we divided our responsibilities reasonably. Lastly, we learnt to devote our time into this. None of us have any C++ experience before this class, so both of us started very early and spent at least 5 hours every day on this project and that's why we even had time to implement some of the bonus.

Q6: What would you have done differently if you had the chance to start over?

A6:

1. Since we basically sit next to each other and code together, so using git isn't necessary. However, if we can start over, we are definitely learn to use git to share our code.
2. We structs our classes first then come to the main logic, which caused a waste of time because later on we need to go back to those small classes to change functions again. If we can start over, we would definitely start with the main logic in Board class.
3. We didn't compile until we both finish our code, so when we first compiled it, the error message is so long that we spent 7 hours debugging for it. If we can start over, we would test and compile as we go.

Part IV: Updated UML

- We used to have a pointer in academicbuilding which points to TextDisplay, because we thought we wanted to use that to update improvement in the visual board. However, we only have a pointer in Board class to TextDisplay and Board is the only class that can communicate with TextDisplay.
- We used to have OwnableProperty*[28] which only keeps track of all the ownable squares. However now we just have a Square*[40] to keep track of all the squares, but when we only need to call OwnableProperty squares, we have a bool in Square indicating if it is ownable or not.
- We used to have a map<string, bool> assetlist in Player because it would be easier to print out the assetlist. However we realized it is unnecessary.
- We used to have improve() in Player: then we moved it to be a pure virtual method in Square, which is only implemented in AcademicBuilding and implemented as dummies private methods in Residence, Gym and UnownableProperty.
- We used to have a action method in OwnableProperty and an gameAction method in UnownableProperty. However, now we just have a single pure virtual action() in Square and each of the subclasses of OwnableProperty and UnownableProperty will have different implementations.

- We also have a notifyremove() method in TextDisplay which is called everytime a player bankrupts and needs to be removed from the visual board.

Part V: Final Notes

This project took us over 60 hours on this project and we are pretty proud of our work. Even though we didn't use any design patterns but we made this decision because we think using design patterns would make the implementations unnecessarily complicated. Each class has its own work, such as all the visual display, such as the beginning greeting and the ending image, the board it self are all in TextDisplay, the Board logic are all in Board, etc. We know that there will definitely be a better way to design this game, and we will keep optimizing it after it is due.

Thank you for grading ☺

Liu & Wu