Watopoly Design

Members: Hongyu Ding(h44ding), Samuel Chen(x732chen), Shihong Pan(s43pan)

1. Overview

1.1 Game

Our program has a class called **Game** which consists of four classes: **Square**, **InputManager**, **Graphics**, and **Player** (see sections below for more details). **Square** is the superclass of **Building** and **nonProperty** which stores basic information of a square on the board. **InputManager** is used to read command-line arguments and facilitate **Game**'s need of reading saved files and accessing commands and arguments. **Graphics** is used to print and update the board and game messages. **Game** itself controls the progress of the entire game, such as save&load, update game each turn, and trade. It also includes enumeration of GameState {NO_GAME, PRE_GAME, IN_GAME, WON_GAME, LOST_GAME} and GameMode {NORMAL_GAMEMODE, TESTING_GAMEMODE}.

1.1.2 Important Methods

// Store all information of the game into a file called filename.

void saveGame(string filename)

// Load game from a file called filename.

void loadGame(string filename)

// Initiate trade process with another player.

void trade()

// Initiate auction of a building.

void auction(shared_ptr building)

// Initialize the game and all squares; set the game state to PRE_GAME.

void init()

// Process input in different game state. In PRE_GAME, receives input to set up player information or load a game. In IN_GAME, // process different commands.

void processInput()

// Update the squares and game status in each turn.

void update()

// Use **Graphics** to refresh and print the board.

void render()

1.2 Square

Each **Square** object stores its name, unique id, coordinate on the board, vector of shared pointers to players, and an additional pointer to the new player who landed on it in the current turn. The id is used to check if the players land on the square in each turn. The coordinate is only used by **Graphics** to print the board. Each turn we notify each square about the position of each player (this is done by **Game**) so that each square could update the players stored in it. **Square** also includes two pure virtual methods (*update*, *render*) which are implemented by different classes of buildings and non-properties.

1.2.1 Important Methods

// Each turn, update the players who are still on the square and store the new player who just landed on it in the current turn. // mplayers is the vector of all players in the game.

void updatePlayers(vector<shared_ptr> mplayers)

1.3 Player

Each **Player** object stores its name, symbol (used to display on the board), position (used to match the id of squares), balance, and other member variables related to the player's status and owned assets (will be mentioned below in relevant sections). **Player** has member methods for various commands and actions in the game, such as mortgage, unmortgage, and buy buildings.

1.3.1 Important Methods

// Mortgage the building if the player owns the building, the improvements are cleared and the building is not already mortgaged.

void mortgage(shared_ptr building)

// Unmortgage the building if the player owns the building, has enough balance and the building is not already unmortgaged.

void unmortgage(shared_ptr building)

// Buy the building and increase the number of corresponding type of building stored in the player.

void buy(shared_ptr building)

// Pay tuition to the School depending on the option chosen ($300 or 10% of total worth), return the amount paid.

int payTuition(string option)

// Send the player to DC Tims Line by setting the position of the player as the id of DC Tims Line.

void gotoTims()

1.4 InputManager

InputManager stores a line of input strings, command specified by the line, and the arguments following the command. It has similar methods to istream, but is designed specifically for this project to facilitate the storage of and access to command and arguments. It also provides a method to extract words in a line which help to load game from files.

1.4.1 Important Methods

// Read a line of command: command <args ...> and store the command and arguments. If fail, return false.

bool readline()

// Extract words in a string and store them into a vector of string.

static vector split(const string str)

1.5 Graphics

Graphics stores screen height, screen width, game messages, and a 2-dimensional vector of char which represents the board. Its member methods provide funtionality of printing and refreshing the board, modifing "pixels", drawing image from a file, printing game messages, and writing strings at certain position on the board.

1.5.1 Important Methods

// Clear all contents in the 2-dimensional vector of char by replacing each char with ' '.

void clear()

// Print the board with default dimension (screen height and screen width); print game messages if any.

void render()

// Print the board with the default screen width and specified height.

void render(unsigned int height)

// Modify one pixel of the board.

void draw(char value, int x, int y)

// Modify the board with the contents read from a file (useful for printing welcome themes and preset board).

void drawImage(string filename)

// Write string(e.g. name of the player) to the board at the specified coordinate.

void write(string content, int x, int y, int width)

// Add game messages which will be printed by render().

void addMsg(string str)

1.6 Building

Building is a subclass of **Square**. In addition to the information stored in **Square**, **Building** stores its purchase cost and owner. It also records whether it is mortgaged. Since each kind of square behaves differently based on the player's information, the implementation of methods *update* and *render* is left to the subclasses of **Building**(i.e. **Academics**, **Gym**, **Class**).

1.6.1 Important Methods

// Set the building to be mortgaged or not mortgaged; if failed, do nothing.

void setMortgage(bool SetOrNot)

1.7 Academics

Academics is a subclass of **Building**. In addition to the information stored in **Building**, **Academics** stores string of the monopoly block it belongs to, level of improvement achieved, improvement cost at the current improvement level, and a vector of tuition corresponding to different improvement levels. It implements *update* to achieve the funtionality of paying tuition and buying the building. It also implements *render* to add strings representing existed improvements, players in it, and its owner to the object of **Graphics** gfx, if any.

1.7.1 Important Methods

// Add improvement level by 1. Return true if succeeded, false otherwise (if exceed the highest level).

bool addImprovement()

// Reduce improvement level by 1. Return true if succeeded, false otherwise.

bool removeImprovement()

// Update the players currently in the building; charge tuition or initiate buying for the new entered player; add game messages // to **Graphics** gfx, if any.

void update(vector<shared_ptr> players, shared_ptr gfx)

// Add name, current improvement level, the symbol of players in the building , and the symbol of building's owner into **Graphics** // gfx, if any.

void render(shared_ptr gfx)

1.8 Gym

Gym is a subclass of **Building**. It implements *update* in a similar way as **Academics**, except that the way of calculating usage fees is special according to the rules. It implements *render* in a similar way as **Academics**, except that it does not add string representing improvement level to gfx since **Gym** has no improvement available.

1.8.1 Important Methods

// Get usage fee by rolling two dices and check whether the owner, if any, owns the two gyms.

unsigned int getUsageFee() const

1.9 Residences

Residences is a subclass of **Building**. It implements *update* in a similar way as **Academics**, except that the way of calculating rent is special according to the rules. It implements *render* in a similar way as **Academics**, except that it does not add string representing improvment level improvement level to gfx since **Residences** has no improvement available.

1.9.1 Important Methods

// Get rent by checking number of residences owned by the owner of this residence; return 0 if this residence is not owned.

void getRent() const

1.10 nonProperty

nonProperty is a subclass of **Square**. **nonProperty** does not store anything in addition to the information stored in **Square**. Same as **Building**, the implementation of *update* and *render* is left to the subclasses of **nonProperty**(i.e. **DCtims**, **GoToTims**, **CollectOSAP**, **GooseNesting**, **CoopFee**, **NeedlesHall**, **SLC**, **Tuition**).

1.11 DCtims

DCtims is a subclass of **nonProperty**. **DCtims** stores the index of current player (player that rolls dices in the current turn). It implements *update* by first calling updatePlayers(mplayers). If there is new player being sent to DC Tims Line by any means other than landing on it, *update* adds a game message to gfx. Otherwise, *update* adds game messages of available options for the current player to gfx if the current player is already in DC Tims Line. **DCtims** implements *render* in a similar way as **Academics**, except that it does not add improvement and owner name to gfx since they are unavailable to non-properties.

1.12 GoToTims

GoToTims is a subclass of **nonProperty**. **GoToTims** implements *update* by first calling updatePlayers(mplayers). If there is new player landing on it, it calls the player's gotoTims() to send the player to DC Tims Line and relevant game messages is added to gfx. Otherwise **GoToTims** does nothing. It implements render in the same way as **DCtims**.

1.13 CollectOSAP

CollectOSAP is a subclass of **nonProperty**. **CollectOSAP** has a member method to add players to the players stored in it, since every player starts at **CollectOSAP** as the game starts. It implements *update* by first calling updatePlayers(mplayers). If there is new player landing on it, the player received $200 and relevant game message is added to gfx; otherwise it does nothing. The case of the player passing through **CollectOSAP** instread of landing on it is considered in **Game**, where we check if the distance of moving forward would exceed the distance between the player's current position and the end of the board circle (position 40, since there are 40 squares in total and the id/position of **CollectOSAP** is 0). **CollectOSAP** impelements *render* in the same way as **DCtims**.

1.13.1 Important Methods

// Add each initial player to **CollectOSAP** since all the players start from here.

void addPlayer(shared_ptr initPlayer)

1.14 GooseNesting

GooseNesting is a subclass of **nonProperty**. **GooseNesting** implements *update* by first calling updatePlayers(mplayers). If there is new player landing on it, the new player receives all the money in the bank, and game message of goose attack is added to gfx. **GooseNesting** implements *render* in the same way as **DCtims**.

1.15 CoopFee

CoopFee is a subclass of **nonProperty**. **CoopFee** implements *update* by first calling updatePlayers(mplayers). If there is new player landing on it, the new player loses $150 and relevant game messages are added to gfx. **CoopFee** implements *render* in the same way as **DCtims**.

1.16 NeedlesHall

NeedlesHall is a subclass of **nonProperty**. **NeedlesHall** defines a structure called NHOption({money, weight}) which stores amount of money gain or lost and a weight associated with it. It also defines a constant NHOption array of size 7 to stores the 7 different actions described in Table 3 in waterpoly.pdf. For example, -$200 with probability of 1/18 is stored as {-200, 1}, and -$100 with probability of 1/9 is stored as {-100, 2} (The total weight adds up to 18). Then we generates random integer between 1 and the total weight, minus each weight by order from this integer, and when the integer is below 0, we pick the amount of money associated with the current weight as the result. **NeedlesHall** implements *update* by first calling updatePlayers(mplayers). If there is new player landing on it, the new player gains or loses money based on the distribution table and relevant game messages are added to gfx, or in 1% chance receives a cup if the total number of cups owned by all players is less than 4. **NeedlesHall** implements *render* in the same way as **DCtims**.

1.16.1 Important Methods

// Pick an amount of money among the 7 options based on their probability (weight).

void getOption()

1.17 Tuition

Tuition is a subclass of **nonProperty**. **Tuition** implements *update* by first calling updatePlayers(mplayers). **Player** stores a bool variable representing whether need to pay tuition. If there is new player landing on it, the new player is set to "need to pay tuition", and game message of available options ($300 or 10% total worth) is added to gfx. The payment is processed by **Game**. **Tuition** implements *render* in the same way as **DCtims**.

1.18 SLC

SLC is a subclass of **nonProperty**. **SLC** defines an enumeration called SLCmove (see **1.18.1**) of different moves described in Table 2 of watopoly.pdf. It also defines a structure called SLCOption in the same way as how NHOption is defined, except that the amount of money is replaced by elements in SLCmove. Then a constant SLCOption array of size 8 is defined according to Table 2, and the method of generating one move from the array is the same as how getOption() works in **NeedlesHall**. **SLC** implements *update* by first calling updatePlayers(mplayers). If there is new player landing on it, the new player moves according to the move generated from the distribution table and relevant game message is added to gfx, or receives a cup in the same way as **NeedlesHall**. **SLC** implements *render* in the same way as **DCtims**.

1.18.1 SLCmove

enum SLCmove { NONE, BACK_3, BACK_2, BACK_1, FORWARD_1, FORWARD_2, FORWARD_3, GO_TO_DCtims, GO_TO_COSAP}

2. Design

2.1 Design Pattern: Obsever pattern

Here we apply observer pattern to **Game** and **Square**(and all its subclasses, same below). **Game** is the subject, **Square** is the observer. In each turn, **Game** calls *update* method of each **Square**, which has the funtionality of notifying all observers. The amount of information received by **Square** depends on the type of square it is. For example, **Square** itself has updatePlayers(mplayers) to receive the position of each player in order to update the players who are still on the square in the current turn. However, *update* is only implemented by subclasses of **Building** and subclasses of **nonProperty** since they need other players' information(status, balance, etc.) in addition to their positions.

2.2 Design Patter: Template method

We apply template method design pattern to **Square**. While every kind of square (**Academics**, **DCtims**, etc.) inherits part of its update functionality (namely updatePlayers(mplayers)) from **Square**, it has its own implementatoin of *update* to modify the player's status or add messages since each square has different actions to perform.

2.3 Usage of InputManager

Our design of **InputManager** facilitates not only processing player's command, but also reading loadfile. Its functionality of extracting words and saving arguments separately makes the loading process and command reading more coherent and clear.

2.4 Usage of Game

We find that having a class that controls the overall flow of the game is favorable in this project. The class **Game** is created for this purpose. In fact, it is intuitive to create such class due to the need of actions such as rolling dice and save/load. **Game** deals with commands and organizes the interaction between **Square**, **Player**, and **Graphics**. Actions such as teleporting to certain position and rewarding money when passing by are implemented in **Game**. In this way, we ensure that each **Square** only needs to know necessary information of itself and of the players, and no information of other **Square** and the board. Moreover, we transfer part of works from some kinds of **Square** to **Game**. For example, the *update* of **DCtims** only adds game messages to gfx. The main works are handed to **Game**, which controls the following game progress and asking for player's actions. We also implement the functionality of bankrupt, trade and auction in **Game**. Such functionality which requires further interaction with player commands are centralized in **Game** such that the implementation becomes more natural and clear. We also use **Game** to create different game states (PRE_GAME, IN_GAME, WON_GAME, etc.) to divide the actions performed. For instance, in PRE_GAME we deal with players setup, file loading, and printing preset gameboard and themes; in IN_GAME we deal with normal game loop to process commands and update the board. This leads to a clear partition of functionality and avoid possible chaos in coding.

2.5 High cohesion, low coupling

Our design of modules ensure high cohesion and low coupling as all the codes of each class follow a single logic without having unnecessary connection with other classes. For example, each kind of squares only has connection with its superclass(**Building**, **nonProperty**). Also, **Player** only has connection with **Building** and **Game** for necessary interactions such as ownership and trade. In addition, **Graphics** focus on modifying "pixels" of the board and printing them and it is only related to **Game** since **Game** controls when to update and print. This proves high cohesion and low coupling. Another example of high cohesions is that every subclasses of **Building** and **nonProperty** only focuses on implementing its own functionality without having connection with other squares. Even **Game**, which necessarily performs various tasks to control the game progress, is partitioned into different sections where each section focuses on one functionality(such as update of the entire board, printing the board). The section of updating the game is further partitoined by different game states, where each game state has one single logic(set up game, iterate game, end game, etc.) and each game state does not interfere other game state.

3. Resilience to Change

3.1 Themes

Based on consumer's need, the themes of the board may need to change(such as Christmas special edition). This requires an overall renewal of the style of grids on the board and the replacement of welcome logo at the centre. In our project, we implement the functionality of drawing the board by reading input from file(using drawImage(filename)). Therefore, it is convenient for us as we can just copy the design of board into a text file, then run the program again to read and print out the board. We do not even need to recompile the program, since the program can read whatever the text file stores.

3.2 Adding squares & player modification

As mentioned above, we apply template method design pattern to **Square**. If in the future we need to add more squares of different kinds(e.g. a brand new engineering building), we could just add another subclass under **Building** or **nonProperty**, and the only work to do is to add its own version of *update* to satisfy the functionality it has, since all other interactions with **Player** are already defined. Moreover, sometimes we need to add more features to **Player** due to the need of increasing fun or the side-effect of adding squares. In this case, we only need to add member variables and methods to **Player** to support its interaction with the new squares, and add relevant codes in **Game** if the new features require interaction with player's command.

3.3 Modification on existing squares

It is possible to modify the attributes of existing squares. For example, one may want to change improvement cost of academic buildings and probability of Needles Hall actions. In this case, we only need to change the value of parameter used when adding academic buildings to the vector of **Square** and change the weight of each options of Needles Hall actions.

3.4 Change in board content

Our design of **Graphics** enables us to modify the content of board easily since it is convenient to modify any text display on the board by calling draw and write methods of **Graphics** to add additional text. Nonetheless, the size of board is slightly less resilient to change, which will be discussed in **6.2**.

4. Answers to Questions

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

In the first version, our answer rejected to the use of Observer Pattern(although we argued that it is just "not fairly good"). However, after coming all the way through, we eventually apply the Observer Pattern(as discussed in **2.1**). The reason is that the game is turn-based. Each turn we must update the status of players and squares. Initially our thought was to only update the squares that has player on it. However, this became messy when we want to implement such logic. Our discussion concluded that it would be most direct and clear to provide player's information to all squares in each turn, and the amount of information received depends on the type of square. Moreover, since we already decided to use **Game** to control the overall progress of the game, we let **Game** to be the subject and let **Square** to be the observer. It makes sense to let **Game** packs up the information of players and sends notifications, since we want to pass information of all players to each square, and we want to reduce any unnecessary connection between different **Player**s and **Square**s.

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

Our answer is the same as the first version. Factory method would work, but it is not necessary to use the factory method. The factory method provides an interface for object creation, and the subclasses **SLC** and **NeedlesHall** decide which object (i.e. movement direction) to create, so it is modeled closely to Chance and Community Chest Cards. For example, according to different probability of movements, we can use factory method to create the movement randomly so that it is closer to the Chance and Community Chest cards. However, it is not necessary to use factory methods because all possible movements can be dealt with by simply changing the position of **Player** and do not have to become objects which occupies more memory resources. Thus we plan not to use it.

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

Our answer is the same as the first version. Decorator Pattern can be a good pattern to use when implementing improvements. The reasons are that there are two different improvements, but they share the same properties which is the cost. It is good that we can have different decorators according to the improvement of different faculty buildings. So that when players need improvements, we can simply call the function in decorator instead of manually modify the state of the current academic building.

However, since the cost and number of improvements is fixed, using a decorator pattern seems too much work. We can simply have two fields in the academic building that stores the cost and the number of improvements since improvements are limited up to 5, which would also work and is easier to understand.

Since the number of the improvements is fixed, we decide not to use the decorator design pattern.

5. Extra Credit Features

5.1 Goose bonus

We adds an int variable called bank in **Game** to record the money in the bank, and a bool variable in **Player** to record whether the player is qualified for the goose bonus.

5.2 Board themes change

As mentioned in **3.1**, the functionality of **Graphics** (drawImage) allows simple and convenient themes change.

6. Final Questions

6.1 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?

We worked as a team of 3. We learned the important of deciding an overall sturcture before writing any codes. It is important to build a well-constructed project structure so that any modifications at any time would not be a torture. In addition, github is brilliant tool for team collaboration. It helped us to track the history of modifcations, facilitated remote collaboration, and allowed merging codes and pushing local works. Such help taught us what a real team work in large program looks like, and how to make efficient communication by using the correct tool. Moreover, we learned that it is essential to make clear division of work and communiate any difficulties in order to let teammates to help. We had situation where the teammates' efforts overlapped on the same functionality, resulting in conflict codes which slowed down the progress. It would be better if we built a formal and efficient way of communicating change in codes and tracking where the changes take place. This would be done via github, we just need more experience of using it.

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

As mentioned in **3.4**, when it comes to modifying the size of board, our project shows less resilience to change. Indeed, it is easy to modify the dimensions in **Graphics**. But our set up for coordinate of squares on the board would require repetitive change to match up the dimensions of the square. It would be better if we could come up with a formula that regulate the gap between each square and set up a ratio between dimensions of the board and dimensoins of each square. In this way, if the size of board is modified, we only need to change the dimensions recorded in **Graphics**.