# Watopoly Documentation

Group Members: Yunfei Zhao(y399zhao)  Yiyang Li(yy27li)



## Part I: Class and Subclasses

### Gameboard:

In our implementation, we have a class called Gameboard which has three classes: Player, Square, and Textdisplay. Player contains the player's information, such as the name and assets of a player. Square contains all the information of the building. Textdisplay is a class responsible for printing the gameboard, and it is attached to each players. When a player takes certain action, textdisplayed will be called to respond to that action. Gameboard takes control of the game. For example, it counts the total number of Roll Up the Rim Cup that have been given to players and it helps different players to trade.

***Important methods:***

```
void set_cur_player(std::shared_ptr<Player> p);
// set the current player of the game

void return_to_market(std::shared_ptr<Ownable> s);
// unbuilds all the improvements of a building and returns the building open market

void trade(std::shared_ptr<Player> a,std::shared_ptr<Player> b, int money, std::string c);
// a spends money to buy b's building and the building's name is c
void trade(std::shared_ptr<Player> a,std::shared_ptr<Player> b, std::string c, std::string d);
// a uses a's building c to trade with b's building d

int get_cur_roll();
```

// the Roll Up the Rim Cup that gameboard can still give to player
bool can_get_roll();
// determines if there is still Roll Up the Rim Cup left to give to players
void set_num_roll(int i);
// reset the number of Roll Up the Rim Cup that gameboard can still give to player

std::shared_ptr<Textdisplay> get_text();
// obtains a shared pointer to textdisplay

bool kick_out(std::shared_ptr<Player> p);
// resets the Player p's previous player's next player as p's next player, delete p from the vector of
// Players that gameboard has, returns true if the game has only one player left, false otherwise

1. Player

Player is a class that is used to store a player's infomation, such as money and assets. *Observe pattern* is used while implementing Player. Player has a private field called shared_ptr <Textdisplay> print_text. Every time when a player moves on the gameboard, print_text will be notified to make certain changes. Player also has a private field called vector <std::shared_ptr <Ownable>> buildings which is building list that contains all the building that the player owns. When a player bankrupts, all the building in buildings will be returned to open-market or given to some other player.

***Important methods:***
std::vector <int> give_me_a_num();
// randomly generates a number like rolling a dice

void notify_square(int row_last, int col_last, int row_now, int col_now);
// notify its privaye field <Textdisplay> print_textthe to change char in displayed in the gameboard
// "I stand on you"

void set_position(int square_num, bool sent_to);
//change square the player stands on, calls notify_square

void buy_building(std::shared_ptr<Ownable> a, int cash);
// buy building with cas

void notify_improvement(int row, int col);
// to notify textdisplay that an improvement has been built or unbuilt

int tuition();
// determines the amount that a player needs to pay when they stand on “TUITION” and choose not
// to pay $300

bool send_to();
// determines if a player is sent to DC tims line

bool can_leave_dc();
// determines if a player can be leave dc tims line

bool can_collect_osap();

// returns true if a player passes by or lands on COLLECT OSAP

## 2.Square

Square is a superclass which has two subclasses: Ownable and Nonownable. Ownables can be purchased buy players, including Academic buildings, residences and gyms; Nonownables such as TUITION cannot be purchased by players. Square contains the basic infomation of a land, such as square number, square location and square name.

***Important methods:***

```
bool is_ownable();
// determines if a square is ownable or not
int get_square_num();
int get_row();
int get_col();
// get_row and get_col get the location of the square on the game board
```

## 2.1.Ownable

Ownable is a superclass which has two subclasses: Academic and Nonacademic. Ownable contains infomation such as the the owner of the land and the rent a player need to pay when they land on this square.

***Important methods:***

```
std::string get_owner();
// get the name of the owner of the square

int get_rent();
// get the rent that a player needs to pay when they land on this square

bool can_own();
// determines if this square has owner, if it is return false, otherwise, returns true

void pay_rent(std::shared_ptr<Player> p);
// Player p pays the rent to the owner of the land

virtual void set_rent(int i = 0)=0;
// virtual method that resets the rent of a player need to pay when they land on this square.

void set_owner(std::shared_ptr<Player> p);
// sets the owner of the land

 bool is_mortgage();
// determines if a land has been mortgaged

 std::string get_monopoly();
// gets the monopoly that the building belongs to

int get_cur_improvement();
//get the # of improvement the building have, nonacademic return 0

bool check_block();
```

//check all ownable in the same monopoly if any of them have improvement(one have->true)

### 2.1.1. Academic

Academic contains all the information of an academic building, such as the improvement cost, the different tuition that other players need to pay when the number of improvement changes.

***Important methods:***

void set_rent(int i = 0)override;
// based on the number of improvment the the buidling has to reset the rent of the building

void buy_improve();
void sell_improve();
// buy_improve and sell_improve will reset the owner's money, the number of improvement the
// building has, and they  also call a owner's method to notify texdisplay

### 2.1.2. Nonacademic

Nonacademic is a superclass which has two subclassesm Gym and Rez.

***Important methods:***

int num_monopoly();
// gives number of residences/gym the owner has

#### 2.1.2.1. Gym

***Important methods:***

void set_rent(int i)override;
// giving the number of sum of the two dice i, set the rent that the player needs to pay

#### 2.1.2.2. Rez

***Important methods:***

void set_rent(int i = 0)override;
// set the rent that the player needs to pay after checking number of residences that the owner has

### 2.2. Nonownable &&  2.2.1 Chance

Nonownable is a class used for good style, it only has a constructor and a destructor. Chance is a subclass of Nonownable. Chance itself also has two subclasses: SLC and Needles. Chance has a virtual method named std::string calculus(bool open), so that SLC and Needles both also have it. This method can ramdomly generates a result  based on some probability (like Chance and Community Chest cards).

2.2.1.1.SLC

***Important methods:***

std::string calculus(bool open)override;
// returns a string that calculus ramdomly generated based on certain probability
int go_to(std::shared_ptr<Player> p, std::string cal);
// returns the number of steps a player needs to move according to string cal

2.2.1.2.Needles

***Important methods:***

std::string calculus(bool open)override;
// returns a string that calculus ramdomly generated based on certain probability
void mutate_money(std::shared_ptr<Player> p, std::string m);
// resets the money that a player has according to a string m

## 3. Textdisplay

Textdisplay is a class responsible for storing the char that needs to be printed to the screen, so it has a private field vector <std::vector<char>> theDisplay.

***Important methods:***

void notified_char(int row_last, int col_last, int row_now,int col_now,int position,char c);
// last_square " ", cur_square "char"

void notified_improve(int row, int col);
// when an improvement is built, change theDisplay

void cancel_improve(int row, int col);
// when an improvement is unbuilt, change theDisplay

friend std::ostream &operator<< (std::ostream &out, const Textdisplay &td);
// prints the gameboard to screen

Part II:

## 1. Extra features

1. Use of shared pointers, no memery leaks!
2. Class Chance has a virtual method calculus used to randomly generate a result
3. Detailed response, for example, when a player tries to buy or sell improvement, if he/she fails, the reason why the improve command fails would be printed on screen

## 2.Changes from Due Date 1

We did not make significant changes after due date 1, but we did add some methods in some classes to help us get the information stored in private field. We also add some methods in some class to help us get control of the game. For example, in Gameboard, we added a method called kick_out(std::shared_ptr<Player> p), it deletes a player p from the vector of Players that gameboard has, returns true if the game has only one player left.

## 3. Plan of Attack

**Question 1:** After reading this subsection, would the observer Pattern be a good pattern to use when implement a game board? Why or why not?

Due Date 1 Answer: No, we think that game board is a class used for store all the necessary information, such as players, squares, and text-display, so the observer pattern should not be used while implementing the game board. However, observer pattern will be used between those the objects of players and text-display. For example, a player may notify other players or text-display, and a player can be notified by other players as well.

Due Date 2 Answer: No change.

**Question 2.** Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Com- munity Chest cards. Is there a suitable design pattern you could use? How would you use it?
Due Date 1 Answer: We did not use any design pattern to model SLC and Needles Hall. Instead, we used a super class called chance, and let SLC and Needles Hall be its subclasses. Both of them share a pure virtual method called calculus, which is used to generate a random card in the game.
Due Date 2 Answer: No change.

**Question 3.** Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?
Due Date 1 Answer: No, we think Decorator Pattern is not a good pattern to use. Decorator pattern is used to enhance an object at runtime. If improvements can be built without a limit and every time when an identical improvement is built, value added to each building is the same, Decorator Patter should be used. However, in our case, separate implementation is needed for each building and Decorator Pattern would not be a good choice.
Due Date 2 Answer: No change.

## 4. Answer to Final Questions

**1. What lessons did this project teach you about developing software in teams?**
While working on this project, both of us learned to share ideas and give support. This project is certainly time-consuming. It took three days for my partner and I to finish discussing all the implementation detail. During our discussion, we sometimes needed to persuade each other with our own ideas. Listening and expressing opinions are improtant in this process, since sometimes a better solution would just come to us when we shared our ideas. We ended up having a good design that we did not need to make many changes when we worked on implementation. It also took more than a week for us to finish the implementation. Sometimes, we would get frustrated when we could not the bug in our code. Having a partner is then really improtant since we would always get help and support from each other, so that we could make more progress than just working alone.

**2. What would you have done differently if you had the chance to start over?**
If we had a chance to start over, we might use more helper functions in our program.