# CC3K+ FINAL DESIGN DOCUMENT

Mingyu PARK (m25park) & Karen LI (k265li) & Yukki YAO (y25yao)

## *I. OVERVIEW (OVERALL STRUCTURE)*

### *i. classes and relationships*

1. *Observer*
    - 1.1 TextDisplay (**View**)
2. Game (**Controller**)
3. *Subject*
    - 3.1 Dungeon (**Model**)
    - 3.2 Floor
        - 3.2.1 Chamber (owned by Floor)
    - *3.3 Cell*
        - 3.3.1 Door
        - 3.3.2 Floortile
        - 3.3.3 Passage
        - 3.3.4 Stair
        - 3.3.5 Void
        - 3.3.6 Wall
    - *3.4 Character*
        - 3.4.1 *Player*
            - 3.4.1.1 Dwarf
            - 3.4.1.2 Elves
            - 3.4.1.3 Human
            - 3.4.1.4 Orc
            - 3.4.1.5 Mingyu
        - 3.4.2 *Enemy*
            - 3.4.2.1 Dragon
            - 3.4.2.2 Merchant
            - 3.4.2.3 other general enemies
    - *3.5 Item*
        - 3.5.1 *Major*
            - 3.5.1.1 Compass
            - 3.5.1.2 BarrierSuit
        - 3.5.2 Treasure
            - 3.5.2.1 Normal & SmallHorde
            - 3.5.2.2 MerchantHoard
            - 3.5.2.3 DragonHoard
        - 3.5.3 Potion
            - 3.5.3.1 RestoreHealth, BoostAtk, BoostDef
            - 3.5.3.2 PoisonHealth, WoundAtk, WoundDef
4. Other utilities
    - 4.1 GameState
    - 4.2 Direction
    - 4.3 PCType
    - 4.4 EnemyType
    - 4.5 Exceptions
        - 4.5.1 Invalid
        - 4.5.2 PlayerDead

### *ii. important super classes and subclasses*

1 *Observer*

An abstract class used to observe Subjects, only has one pure virtual method *notify(string action)* which allows subjects passing action messages to observers, invoked by Subject's *notifyObservers(string action)*.

1.1 TextDisplay (**View**)

A concrete class inherited from Observer TextDisplay monitors (observes) Dungeon, all Floors, all Cells, Characters, and Items. Having called by Game when PC Character (PC) has completed its action for the turn, it updates its display field and outputs formatted game information.

Important fields:

1) Dungeon *dungeon
2) vector<vector<char>> display
3) string actions

Important methods:

4) notify(string action)
5) operator<<

Description:

1) a pointer to the model, initialized as nullptr, Game will set it in Game's constructor
2) a 25 rows & 79 columns 2D vector that stores **current** floor's cells' symbol representation, updated when

game is running (GameState is INGAME)

3) a string stores all actions in current move, updated by overriding method notify(string action), and will be reset every time after printing.
4) inherited from its super class Observer, appends parameter action to field actions.
5) a friend output operator that prints out game information based on different game states:
   a) START:
      prints out a welcome page (read from welcome.txt, stored in vector<string> welcome_msg) that asks user to make choice between command line play and easy WASD play; prints out this option again when user restarts
   b) START_COMMANDLINE or START_CURSOR:
      prints out relevant welcome screen respectively (show options and guidelines on how to play); asks users to choose a race (default is Human)
   c) INGAME:
      prints out the 25 by 79 display with 5 additional lines showing PC status and actions with updated information from field Display
   d) WON / LOST
      prints out appropriate won/lost page (read from won.txt and lost.txt, stored in won_msg and lost_msg); prints out PC's score

2 Game (**Controller**)

A concrete class that controls and manages the entire game where both TextDisplay and Dungeon are owned by Game. It reads input from user then it manipulates both TextDisplay and Dungeon accordingly.

Important fields:

1) Dungeon dungeon
2) TextDisplay display
3) string filename

Important methods:

4) play()

Description:

1) ownership of the model Dungeon
2) ownership of the view TextDisplay
3) stores map information as a text file name, initialized with default.txt as default if no additional command line argument provided for customization; would apply the customized file otherwise
4) controls routine of the game with two helper method playCommandline() and playCursor() based on choices made by user input

*3 Subject*

An abstract class that is observed by Observers. It attaches its Observer to field Observers and notifies its Observers when changes are made.

Important fields:

1) vector<Observer *> observers

Important methods:

2) notifyObservers(string action)
3) attach(Observer *observer)

Description:

1) stores all registered (attached) observer pointers
2) notifies all attached observers with action message
3) attaches a new observer pointer into field Observers

## 3.1 Dungeon (**Model**)

A concrete class inherited from Subject that acts as a wrapper with all information for the entire game.

Important fields:

1) Observer *observer
2) Player *player
3) vector<Floor> floors
4) int level
5) GameState state

Important methods:

6) load(string filename)
7) move(Direction dir)
8) use(Direction dir)
9) attack(Direction dir)
10) enemyTurn()
11) setPlayer(PCType race)
12) restart()

Description:

1) a pointer to the observer of this game (TextDisplay); to be assigned to PC and the floors
2) a pointer to PC of this game which will be shared among all floors and only deallocated and reallocated at the start/restart of games
3) contains 5 floors (not pointer) representing each floor in the game, these floors live for the entire life of the program (not destroyed if restart); only destroyed by quitting the game
4) with range [0, 5], representing current floor level; if reaches 5 to indicate game is finished and PC won
5) the state of game; used by Game and TextDisplay to indicate the phase of the game (for additional information, please refer to 4.1 GameState)
6) loads the game from file where it passes filename to all floors; if the file is default map file, it also calls each floor generate methods and choose one floor to place the barrier suit
7) passes the move command to current floor level, if that floor returns true (i.e. PC reaches the stair at that floor), then it will increment the level field, and if level reaches 5, it will change game state to WON; If the call to current floor's move results in exception Invalid (for additional information, please refer to 4.5.1Invalid) thrown, it will notify its observers the details (exception message) of the exception, and rethrow it to Game
8) passes the use command to current floor level, it may throw two types of exceptions, one is Invalid, indicating the invalid use and another is PlayerDead (for additional information, please refer to 4.5.2 PlayerDead), indicating that PC dies after consuming the potion in that direction; in both cases, it will notify its observers with meaningful message for what happened; only rethrows these exceptions to Game
9) passes the attack command to current floor level; catches Invalid exception, if any, notifies Dungeon's observers the detail and rethrows the exception to Game
10) called by Game when move/use/attack commands are successfully executed (and game state is still INGAME after move/use/attack), it then asks current floor to let that floor's enemies randomly move (or attack, if nearby PC); catches the PlayerDead exception, if any, then sets the game state to LOST
11) allocates a new PC on heap depends on specific PCType (for additional information, please refer to 4.3 PCType) and sets the PC to current floor (i.e. the first floor since it is now at the start of the game); notifies observers in Dungeon the spawn of PC and changes the game state to INGAME
12) invoked by Game if at any game phase restart command is read from input; resets the state to START, deletes its PC, resets the level, and invokes reset() method for each floor

## 3.2 Floor

A concrete class inherited from Subject that contains all elements that belong to a single floor.

Important fields:

1) Observer *observer
2) int level
3) vector<vector<Cell *>> cells
4) vector<Chamber> chambers
5) vector<Enemy *> enemies
6) vector<Item *> items
7) Cell *stair
8) Player *player

Important methods:

9) dfs(vector<vector<int>> &map, int row, int col, int chamber)
10) assignChamber(vector<pair<int, int>> &tiles, vector<vector<int>>> &map)
11) setPlayer(Player *p)
12) load(string filename)
13) generate1()
14) placeBarrierSuit()
15) generate2()
16) enemyTurn()
17) move(Direction dir)
18) use(Direction dir)
19) attack(Direction dir)
20) reset()
21) peek(int row, int col, Direction dir)

Description:

1) a pointer to the observer of this game (TextDisplay); to be assigned to each element (Cell, Enemy & Item) living on this Floor
2) represents index in vector<Floor> floors in Dungeon; level of this Floor
3) a 2D vector (25 rows by 79 columns) of Cell pointers that stores Cells for this floor; initialized as nullity; allocated in Floor's *load()*
4) stores 5 Chambers for this Floor
5) stores all Enemies spawned for this Floor (no matter already dead or not)
6) stores all Items spawned for this Floor (no matter already used or not)
7) a pointer to the Stair of this Floor; initialized as nullptr; allocated by Floor's *load()* if read from file, by Floor's *generate1()* otherwise
8) a pointer to PC of the game; initialized as nullptr; assigned by Dungeon through Floor's *setPlayer()*
9) Deep First Search algorithm that calculates the association between Floortiles (within same Chamber as Cell <row, col>) and Chamber indexes it
10) loops through all unassigned Floortile (in {row, col} form) and calls *dfs()*
11) called by Dungeon, indicating that PC has reached this Floor thus activating its activities accordingly (all states remain unchanged if PC is not on this Floor)
12) loads the Floor elements from file (default.txt by default); if default.txt, then only Cells are allocated and calls *assignChamber()*, all Floor elements are allocated otherwise
13) first phase of random generation; generates PC location, Stair location, and Treasures, then if called by Dungeon to *placeBarrierSuit()* (if it is chosen) it places it otherwise not; continues to second phase of random

generation

14) randomly places BarrierSuit on this Floor, with a Dragon spawned nearby to protect it
15) second phase of random generation; generates Enemies and Potions
16) goes through Cells row by row in left most fashion, places any alive Enemy in an operation queue, then starts at the first Enemy in queue, randomly moves it (or attacks PC if nearby PC)
17) moves PC to dir Direction, returns true if PC reaches the Stair on this floor, false otherwise; throws Invalid exception indicating invalid move (INVALID MOVE!)
18) tries to use an item in dir Direction, if PC dies by doing so (used the PoisonHealth potion), throws PlayerDead exception; throws Invalid exception if that item is unusable or does not exist (INVALID USE!)
19) attacks dir Direction if there is any Character (Enemy in this game); if there's no target in dir, throws Invalid exception (INVALID ATTACK!)
20) called when game is restarted, deletes all elements belong to this Floor: Cells, Enemies, Items; clears vectors and resets Chambers (Chamber only has aggregation with Cell)
21) peeks in dir next to Cell <row, col> for potential items; if sees any items, appropriate string is returned (such as "PC sees an unknown potion.")

## *II. DESIGN*

### *i. Model-View-Controller Design Pattern*

The MVC builds the high-level structure of the entire program. **Controller**: Game class, the owner of View and Model, manages user input and transfers them into commands for Model and View. **View**: TextDisplay class, manages data collected from class Dungeon (Model) and prints formatted output. **Model**: Dungeon class, contains all data needed for the program, manipulates commands sent from Controller and manages the data, logic and rules. In Model, there are many classes that form different types of data.

### *ii. Observer Pattern*

This is used by TextDisplay (View) to monitor and observe Subjects, which is everything in the Model part of the structure, including Dungeon, Floor, Item, Character, and Cell. In this way, each Subject could send event massage directly to TextDisplay without knowing TextDisplay's details. If other type of View is added, attaching it to Subjects is also easier than any implementations without Observer pattern.

### *iii. Double Dispatch Pattern*

There are several places that Double Dispatch pattern is used.

❖ isMovable(): Cell & Enemy, Player

When a Character wants to move to a Cell, it passes that Cell into *Character::isMovable(Cell &c)*, then *Player::isMovable(Cell &c)* and *Enemy::isMovable(Cell &c)* call *c.isMovable(*this)*, after which Cell::isMovable(Player &p) and Cell::isMovable(Enemy &e) are invoked respectively, and some of concrete subclasses of Cell override *isMovable* to return a fixed value (true or false).

❖ isGeneratable(): Cell & Character, Item

When trying to place a Character or an Item on a Cell, Character::isGeneratable(Cell &c) and Item::isGeneratable(Cell &c) are called, then the same as isMovable, they pass themselves to Cell::isGeneratable(Character &c) and Cell::isGeneratable(Item &i), then Cell has corresponding methods to manipulate them respectively.

❖ isGeneratable(): Cell & Normal, SmallHorde & Character

This is a triple Dispatch, unlike above, Normal and SmallHorde classes override Item::isGeneratable(Cell &c) and pass themselves in, then Cell::isGeneratable(Normal &t) and Cell::isGeneratable(SmallHorde &t) are invoked, then Cell passes these two variables into Character::isGeneratable(Normal &t) and Character::isGeneratable(SmallHorde &t), if character in that Cell is not nullptr. Character::isGeneratable() method returns false by default, Player overrides and returns true, since Normal and SmallHorde gold could be spawn on the same cell that PC stands on during random generation.

- use() & usedBy(): various PC race & Item

  When PC trying to use an Item, Player's concrete subclasses' use(Item &i) are called, and they pass selves to Item's usedBy method where the method is overloaded. Depending on different PC races, one of Item's usedBy is called, e.g. Item::usedBy(Orc &o). Since all usedBy methods in Item are pure virtual, concrete classes implemented it to handle different PC abilities, e.g. in PoisonHealth::usedBy(Elves &e), e has been added 10 HP instead of -10 HP since all potions are positive for the PC race Elves.

*v. Non-Virtual Interface*

- In *Character::getAtk()* and *Character::getDef()*, NVI is applied.

  Class Character has two virtual private methods *getBoostAtk()* and *getBoostDef()*, these methods return 0 by default, and in *getAtk()* and *getDef()* these two methods are called to calculate boosted Atk & boosted Def, Player overrides these two methods to return true boostAtk & true boostDef values.

## *III. RESILLIENCE TO CHANGE*

- Ncurses support for WASD controls (This feature is implemented, please see *VI EXTRA CREDIT FIGURE* for details.)
- Inventory system for PC to hold multiple potions

  Class Player should add a new field (e.g. vector<Potion *> potions), and a public method that gets descriptions of each potion (e.g. "1. BA", "2. RH", etc.), if potion type is still unknown after added to inventory, then the inventory page will show things like "1. an unknown potion", "2. an unknown potion", etc. In addition, there should be another method that let PC drink one of potions in inventory (e.g. *use(int index)* → use the potion at index in inventory).

  In Game, it should handle several new commands that user could "open the inventory page", "close the inventory page" (or auto closed if wrong command in the inventory page), and "use ith potion in inventory", and possibly, if the game needs to support both "pickup potion into inventory" and "use potion immediately", then new command that does the "pickup" thing should also be handled.

  In Dungeon, its game status might be changed to something like *GameState::INVENTORY* indicating that the current game status is showing the inventory page of PC. Moreover, those new commands added in Game should also be added here. For open/close inventory, Dungeon could interact with PC directly (since Dungeon has composition relation with Player), for possibly added "pickup" command, it should call the current Floor's "pickup" command to do the actual pickup, if "pickup" and "use immediately when pickup" both appeared.
- Expand the inventory and combat systems to make use of new types of treasures (e.g. weapons and armor). The modification would happen for Character, Player, Item and its subclasses. In Character, to be suitable for get the true attack and defense value, its *getAtk()* would return atk + *getBoostAtk()* + *getWeaponAtk()*; similar for *getDef()*. There would be 4 NVI methods, beyond existing *getBoostAtk()* and *getBoostDef()*, there would be *getWeaponAtk()* and *getArmorDef()*, all of them return 0 by default, and Player overrides them and returns real values. In Player, the inventory field created above would be changed into vector<Item *> type, that holds all kinds of Items. Assuming that PC could pick up and store multiple kinds of new treasures (other than gold), by using same command *use(int index)* as stated above, PC could equip an Item (check *Item::isEquipable()* before equip, throw INVALID USE if not equipable) on its body, this is achieved by adding Item pointer fields indicating that which Item has equipped (e.g. there would be one Weapon pointer representing that Player equips that Weapon), by different specifications, PC may have some equipment slots that could contain multiple Treasures at same time, this could be achieved by let that equipment slot being vector<Item *> type. Player could also have two fields, equipmentAtk and equipmentDef, that represent the total amount of benefits gained from equipped Items respectively, and they would have public setters. In class Item, there

would be a new virtual method *isEquipable()* that returns false by default, and it overrides in new types of treasures classes, when they are picked up by PC, they would be added into PC's inventory, and if they are used, they would occupy PC's one equipment slot, and replace old one if no empty slot.

- ❖ Make enemies pursue PC. It could be implemented in two ways. First approach, the pursuit effects all enemies on current Player's floor. In this case, we would simply calculate the row and column differences of each Enemy and PC, then the different directions from Enemy to Player are determined, then we could modify the enemyMove() method in Cell taking a Direction parameter, and each Enemy trying to pursue in that direction, or that direction's neighbor direction, until find an available neighbor. Second approach, the pursuit effects only enemies in current PC's chamber. Similar to the implementation above, but only let those Enemies that are in that Chamber pursue after Player. This would require going through the Chamber that PC is in and getting those Enemies row by row in a left-most fashion for enemyMove, Enemies in another Chamber would remain randomly moving.
- ❖ Merchant buy & sell: several classes required modifications. In Dungeon and Floor, new methods *getGoods(Direction d) buy(Direction d, int index)* and *sell(Direction d, int index)* are added. In Cell, in addition to above methods, *getGoods(), buy(Player &p, int index)* and *sell(Player &p, int index)* are added for other cells to call this Cell's Merchant (if here), they throw INVALID BUY / SELL / MERCHANT exceptions indicating wrong inputs otherwise. In Character, virtual methods *getGoods(), buy(Player &p, int index)* and *sell(Player &p, int index)* are added. They throw same exceptions by default, and only Merchant overrides these methods. In Merchant, it would hold a list of Items as goods, override methods *getGoods(), buy(Player &p, int index)* and *sell(Player &p, int index)* for doing actual implementations. These methods would also throw exception if index is out of range, or PC does not have enough gold to buy with.
- ❖ Other possible changes: Class System (Class System is a little complicated comparing to other extra features. A simple way would add them as one of the fields of class Player, and in class System, their abilities (methods) would be added to existing Player methods such as attack and defense. Some of Classes such as attack Enemy far from PC's neighbor, would require extra work on Floor, Cell, and Character.); More PC races (This feature is implemented, please see *VI EXTRA CREDIT FIGURE* for details.); more enemies (Similar to More PC races feature, new enemies are added under Enemy class, and depending on their special abilities, some methods may override from Enemy and Character.); more items (As stated in the sub-feature of inventory feature above, more items would require implementing more usedBy methods, and if special, such as Weapons and Armor, would require extra virtual methods added in Item); Graphics User Interface using X11 (This feature is implemented, please see *VI EXTRA CREDIT FIGURE* for details.)
- ❖ Due to our code designed with **low coupling**, it's resilient to changes to the existing codebase. Most of our modules communicate between each other via simply calling their methods with either no parameter or a simple parameter. Majority of methods across our modules are merely wrappers for the actual methods that executes a certain task. If a change to how those methods are executed were to be made, only a single class would need to be modified. For instance, if someone were to change the command line inputs with different keys (e.g. 'v' instead of 'a' for attacking), only Game class needs to be changed in order to accommodate modifications. If someone were to change the sequence of spawning of entities (e.g. spawning enemies before items), only Floor class needs to be changed in order to accommodate modifications. Our uses of various design patterns also make the existing codebase resilient to changes. For example, if someone were to allow enemies to be able to pass through doors and passages, one can simply modify Door class and passage class to override the inherited methods. Additionally, if someone were to modify how potions interact with different races (e.g. orc turns positive potions to negative potions), one would just modify the methods in subclasses of potions that are already overloaded with Orc parameter. If someone were to modify the messages printed for a certain action, our code could easily modify the change as printing actions is implemented via observer pattern as

TextDisplay being an observer. Finally, as our code is designed with **high cohesion** in mind, our classes are designed in a way that methods within them would work together to accomplish a single task. For instance, anything related to game mechanics related to floor is handled in Floor class. Therefore, if someone would want to modify such mechanics, methods that need to be changed to accommodate the modifications would be grouped in the floor class. Moreover, as anything related to printing out information is handled in TextDisplay class, if a change was to be made on how messages are printed, methods that need to be changed would be grouped in the TextDisplay class.

*IV. DIFFERENCE BETWEEN ORIGINAL AND FINAL VERSIONS*

We have replaced the initial thoughts on using Factory Design Pattern with using helper functions on Character and Item generations (please see V.QUESTIONS IN PLAN OF ATTACK for details). All the classes and most of their relationships remained unchanged with some optimizations on helper functions. One stand-out change was to make isHostile a static field in Merchant class thus making it easy to make sure the first attack on any of the merchants on any floor would trigger all of them on all floor turning hostile. On top of this, we have challenged some enhancement features (please *see VI. EXTRA CREDIT FIGURES for details*).

*V. QUESTIONS IN PLAN OF ATTACK*

*(Note: answers to Questions 3 & 4 have not changed in final version)*

**Question 1 How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?**

We thought about Factory Design Pattern. (Original)

In the final version, we used helper functions to get the PCType as race and made appropriate calls on respective constructors for the race. It was more convenient and code efficient. Dungeon::setPlayer(PCType race) initializes PC for the game which is shared by 5 floors.

**Question 2 How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?**

We planned to either randomly generate enemies or read customized maps with information of the enemy locations, then we planned to use Factory Design Pattern. (Original)

We kept the random generation and reading input for customized maps (as command line argument when running the executable). And enemy generation is done on each floor whereas player generation is for the entire dungeon (consisting all five floors). However, we used the same approach for enemy generations as for PC generations where we used helper function to find what type of enemy (or race of player) we are going to generate.

**Question 3 How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?**

We could use double dispatch Visitor Design Pattern to achieve enemies' special abilities. For combat-related abilities, we could implement them in each enemy class's attack/defense methods. For game-round related abilities, such as troll's health regeneration, we could add a virtual method (for example, "healthRegenerate()") in class Enemy (by default do nothing) and let class Troll override it (do actual health regeneration). In this way, at the start of each round, class Game calls class Dungeon to regenerate health for all enemies on current level, but only those trolls are actually regenerated.

**Question 4 What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?**

We will use private fields in Player class to track temporary boost effects and reset them when Player Character (PC) moves to a new floor, those fields would be modified by potions using double dispatch Visitor Design Pattern. PC calls "use(Item &)" and item calls "usedBy(Player &)". Both Player class and Potion class will override their use/usedBy methods in their super classes respectively. In Potion class, there will be multiple overloaded "usedBy" methods for different subclasses of Player class, for example, "usedBy(Elves &)".

**Question 5 How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hordes and the Barrier Suit?**

The second part of the question was not changed in the final version. For the first part of the question, we planned to use Factory Design Pattern to generation different Item. (Original)

In the final design we generate the Item the same way as generation of enemies (we used helper functions). Except for a slight difference for Normal and Small Horde of gold which can be generation on top of PC. Before we generate items, we used double dispatch Visitor Pattern to resolve this issue where we have three overloaded versions of Item::isGeneratble for those two types of items and other items.

## *VI. EXTRA CREDIT FIGURES*

- Ncurses support for WASD controls. This change relates to Input and Output, so the game logic does not need to be modified, only Game class and TextDisplay class need to be modified, we implemented in a way that minimize the changes in TextDisplay. Since everything displayed using Ncurses using <ncurses.h> library should be the same as that using command line input, in TextDisplay we just added a method to print the welcome page for Ncurses control (with different instructions on how to play the game: printWelcomeCursor()). It is called in TextDisplay by output operator when it was detected that the game status is GameState::START_CURSOR. In Game class, we added a new method playCursor(), invoked by Game::play(). In playCursor(), Game sets up and initializes the Ncurses components, reads command from Ncurses input method, and executes the same way as for command line play mode. When screen needs to be updated, Game calls TextDisplay output operator and directs the output to a stringstream variable, then keeps reading the variable and prints it to Ncurses screen. This way, there is no need to add a new View class to manipulates those data, since the messages being printed are almost identical for both command line play mode and Ncurses play mode. We just direct the output in class Game, like a "pipe".
- More PC races. A new race class Mingyu is added, its special ability is that when PC picks up gold, HP is restored with quantity that equals to the amount of gold (for example, picking up a Normal gold restores 1 HP, 2 HP for SmallHorde, etc.). It was challenging since that there would be double dispatch Visitor Design Pattern between all concrete Player races and all Items (*use* & *usedBy*). Adding new PC race means that all items are required to implement it. In addition, this race requires extra works on *Treasure::usedBy(Mingyu &m)* and *DragonHoard::usedBy(Mingyu &m)*.
- Graphical User Interface using X11. This is built with help of Xwindow class. Similar to implementing Ncurses, only Game class and class Xwindow are modified, game data logic and manipulation remain the same. In Game class, three new methods are introduced: first, new method *playGraphics()* has been added for specifying the graphics play routine; second, a new helper method is *nextChar()*, similar to Ncurses *getch()*, it keeps busy looping (with *std::sleep()* a reasonable time) until find a XPending event and returns the character if the event is keyPress and is a single character key; third, a method *repaint(Xwindow &w)* that manipulates the output, similar to Ncurses implementation, first it reads the message from TextDisplay passed in by Game, and formats it to fit the needs of Graphical display with the difference that Ncurses takes all messages using GUI output, we only collect messages we needed to show (five rows at the bottom of game board in GameState::INGAME), and lets Xwindow to draw other images. Here it still makes use of TextDisplay, for listening notified actions. Thus, there is no need to attach Xwindow as an Observer to all Subjects and detach when a round of game is completed (*Game::playGraphics()* goes out of scope). In Xwindow class, it stores all icons of Floor elements, including all types of Cells, all types of Enemies, Items, and PCs as well as welcome page, won and lost pages. It has a pointer to Dungeon with aggregation relationship to collect data for current Floor's foreground and background (for example, symbol etc.). It has methods *drawDungeon()* for drawing the game board when it is with GameState::INGAME (note that 2D vectors foreground and background gets updated here); *drawWelcome()* for showing the welcome image; *drawWon() & drawLost()* for showing won and lost screen; *drawString()* to display game data under game board. Note that this class lives on *Game::playGraphics()*'s stack and it will be created

at the function call and destroyed after the function call has ended.

## *VII. FINAL QUESTIONS*

**Question 1 What lessons did this project teach you about developing software in teams?**

We learned that in order to achieve a good outcome, all team members should have the desire to collaborate and always strive for the best. The key components of a successful team are frequent and effective communication, solid technical skills, and strong teamwork. Among all three of them, communication was the most important factor in our experience. In this project, we adapted the agile methodology. By following the principle of this approach, we constantly analyzed, tested and discussed about our program. When we ran into obstacles, we examined the options and evaluated the pros and cons of each of them. Through these small discussions, we learned a lot about Object-Oriented Programming from each other. In addition, we sought for external help. We clearly understood that with the help from experts, our program can only get better. After we collected all the feedbacks from different sources, we decided together what changes should be integrated into our program. We repeated this process throughout the developing stage. Overall, these practices ensured that we were always on track to what we wanted to achieve, and thus, led to a better result.

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

If we had the chance to start over we would spend more time on making Xwindow class an Observer. We currently have Xwindow stand-alone with a has-a relationship with Dungeon (to get information on what to display).
Game::playGraphics() stack-allocates Xwindow as a local variable (which will be deallocated when it goes out of scope) and it helps to bridge communication between TextDisplay and Xwindow so Xwindow will know what to repaint.
Additionally, we would have considered implementing more bonus features such as PC levels, larger attack range (instead of 1 cell radius), potion inventory system and random map generation, since it is fun ☺