CS246 Spring 2024 CC3K Project Report

Acknowledgement:

This project was made possible by our group effort of getting cooped up for 5 hrs straight for a week. Thank you to CS246 Instructors and ISAs for their timely Piazza response and overall course experience. It has been a very enjoyable bonding (and stressful) experience.

Special thanks to EIT 2053, MC Lab, Tim Hortons Iced Coffee and Monsters for existing.

Introduction

We work in a team of three where each of us takes on a portion of the program and in the process of designing the dungeon, we learned a lot about efficient OOP design as well as solving techniques for various design patterns. We also gained valuable experience in developing a holistic program that encapsulates both programming expertise as well as communications and collaboration.

Overview (describe the overall structure of your project)

Our project is divided into a few main Superclass[1] that shoulder the project: *AsciiArt*, *Character* (which includes *Player* and *Enemies* Superclass), *Item* (which includes *Potion* and *Treasure* Superclass), and various other game-play classes that rendered the game possible. In addition, in light of designing the dungeon, we have a *Dungeon* class that will be constructed using the concrete class *Blank* to lay out the cornerstone map of the game. *Dungeon* is responsible for rendering the game and output the current status of the game, while *ItemFactory* and *EnemyFactory* are calling the item constructor and enemy constructors respectively to produce game essentials. The overall game control is coded under the *PlayGame* class that is responsible for calling each of the classes' methods to spawn staircase, potions, enemies, etc randomly[2] when necessary, and it also handles various operations like quitting and restarting.

Design

Our program overall follows a decorator pattern, where player, enemies, potions, etc. are "decorated" on top of the concrete class *Blank*. By implementing the *AsciiArt* class, we were able to package the virtual charAt() method to all *Decorator* and concrete class *Blank*, enabling them to easily access each other's coordinations and making interactions more polished.

Character acts as the super-class of *Players* and *Enemies* since they share lots of common commonalities such as "htp", "atk", "def", "race", etc. so implementations accessors, mutators will not duplicated as well as accessing some of the shared fields like "announcement" will be more cohesive. Even though *Enemies* and *Players* are subclasses of *Character*, they are still responsible for their own separated functionalities to ensure high cohesion in code implementations and compiling. It also eliminates code duplication by mapping Player's race for easy char -> string or directions (char -> string) when needed to set an announcement in action. In terms of enemies' functionalities, specific features can easily be implemented by overriding the pure virtual method, atkOrMove(), in the *Enemy* base class.

[1] Classes will be Italicized for formatting purposes.
[2] Randomness is controlled by *CheckCoord*

The *Item* base class is responsible for defining potions and treasure (gold) elements in the game. It uses polymorphism and virtual methods to enhance code reusability, while ensuring that each item operates accordingly. Its subclasses: *Potion* and *Treasure* have numerous subclasses that construct various types of potions and gold, including *BA, Dragon_Hoard,* etc. The *Item* class contains a field that indicates the effect of a potion or the value of a treasure, as well as virtual methods responsible for applying and undoing effects. Since all potions are displayed as 'P' and all treasures are displayed as 'G' regardless of their types, the *Item* class provides a vector that allows players to access the item at a specific location. This enables the correct effects to be applied to the player.

Items and Enemies utilize an *ItemFactory* class and an *EnemyFactory* class respectively to create instances in a more consistent manner. Furthermore, the use of enumeration in these factory classes enhances code readability.

Our program employed low coupling by facilitating communications between modules through parameters. For example, a *Player* * is passed to the *applyEffect* method in *Item,* allowing the method to operate on the player. Similarly, a *Dragon_Hoard* * is passed to a *Dragon*, so that the *Dragon* is able to modify its hoard's state when slain. Another example would be how we implemented combat between PC and enemies: Enemies and PC inform each other to take damage while having little to no knowledge how all other features are implemented. In addition, we provided two factory classes for enemy and item creation, which encapsulate the creation steps, and allow clients to create instances of these classes through the interface. This reduces the dependencies between modules, especially *PlayGame, Enemies,* and *Items,* thus making our program more adaptable to change. This way, each of our modules has low dependencies on another, thus they are more maintainable while being able to effectively interact with other elements in the game.

Moreover, our program employed high cohesion with the use of polymorphism. We defined several base classes that have focused responsibilities and similar functionalities. For instance, the *Character* class (which includes *Player* and *Enemies*) is responsible for keeping track of a character's stats and state, such as moving and attacking. *Item* (which includes *Potion* and *Treasure*) is in charge of managing the effects of items on players according to their unique properties. By defining subclasses that serve a particular purpose and encapsulating their functionalities in a base class, our modules are reusable and consistent.

Another interesting design problem that we were able to solve with no memory leak is deleting the current dungeon while maintaining the player stats inside the game. By using the decorator pattern, as each of the presented *Decorator* is *linked* with each other via linked-list, when the player needs to "level up" we simply (simple in implementation but took a few

iterations to come up with) “un-linked” the next[3] pointer inside *Staircase* Decorator (i.e the *Decorator* right before the player) and set it to “null-ptr”. This way, when we delete the current level, as it approaches the *Staircase* and eventually destroys it, it will not be able to reach the *Player* Decorator. The caveat here is being able to make sure nothing in our implementations is pointing to these deleted *Decorator*(s), which takes a few “segmentation faults” to learn that the maps of enemies also need to be clear. We also come to appreciate the beauty of the decorator pattern as it is neatly deleted and frees all the memory by recursively destroying the next Decorator and in the process, meticulously prevents memory leaks.

Resilience to Change

Since our program is divided into small and independent modules, we are able to modify portions of our coding without affecting others. Polymorphism also enables us to easily introduce new types that can make use of existing functions from the base class, without the need to make significant changes. We are able to extend new functionalities of existing types by building on top of the base class.

1. PC: PC uses a “raceAbility” map so adding new races to a Player will be handled without hard coding or using magic numbers in the constructor, however, it will require creating a new subclass which is expected for the high-cohesion aspect of code maintainability. Furthermore, shared implementations of methods like taking potions, attack, or moving will require little to no changes depending on the “race” ability since they are functions implemented in their base class - *Player*.
2. Enemy: To recreate an existing enemy or create a new enemy with extra features, it suffices to add an extra derived class from the base class – *Enemy*. The *Enemy* class is abstract, where the pure virtual method is the atkOrMove() method, which could be overwritten according to the specific feature needed. New fields can also be added when necessary[4].
3. Items: To create new potions or treasures we can define new subclasses that inherit from *Potion* or *Treausre,* which inherit from *Item.* The base class is abstract, with fields and methods that outline the general structure and behaviour of all items. To define specific features for a particular item, we can override virtual methods and add additional fields.
4. *CheckCoord*: the only class that might not be resilient to change would be the *CheckCoord* class due to the way it approaches random generation of potions and enemies.

[3] Field in Decorator, see Decorator.h for more details
[4] See how *Merchant* and *Dragon* are implemented

5. Dungeon: Though not implemented, we decided to add a chamber class that would be able to generate a random map after the dungeon is constructed. This would elevate the game play experience and overall enhance the quality of our project.
6. *EnemyFactory* and *ItemFactory*: We used two factory classes to centralize the creation of enemies and items. This enables our program to easily adapt to the extension of new types and the modification of existing types. To extend new types, we can add new named constants in *enum* and define the creation logic in the factory classes. To modify existing types, we only need to work within the factory classes, allowing for changes without affecting the client code.

Answers to Questions

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

   The "Player" class is a derived-class of "Character" Super Class. The "Character" Super Class already contains most of the shared methods like accessors, and mutators that are needed for the implementation of each player. The "Player" class is also the Base Class of the derived-classes that generate different races of a player like Shade, Drow, etc. Therefore, to add an additional race to the game, we can easily make a subclass with a constructor that calls the player constructor, in that way, we can freely generate a new race with little changes to the implementation.

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

   To handle the generation of different enemies in the game, we implemented a factory pattern through an EnemyFactory class. This class includes a method to create enemy instances based on the type specified. By passing parameters such as the enemy type, coordinates, and any additional attributes to the factory method, the appropriate enemy subclass is instantiated. Each enemy subclass, like Human, Orc, or Dragon, overrides common methods such as attack and specialMove to provide unique behaviours.

   As to randomly generating the enemy. We will also use the rand() function provided in the standard library to generate each enemy based on the probability given[5], which is also how we implement it to generate the PC. What is different is that the PC cannot be

[5] Controlled by *CheckCoord*

spawned in the same chamber as the stair, as the enemies have no such restriction.

3. How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

   To implement the various abilities for enemy characters, we can leverage polymorphism by defining virtual functions in the Enemy base class and overriding them in the derived classes (the subclasses such as Human, Merchant, etc...). This way, each enemy can have specific implementations for their unique abilities. A similar approach also applies to the PC as it is also a super class of classes such as Shade, Vampire, etc... This way, as all the specifically defined game characters share some of the same information (such as HP and Atk), they have their own methods implementing their special abilities.

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?

   To model the effects of temporary potions, we defined virtual methods of *applyEffect* and *undoEffect*, and overridden them in each subclass (which includes BA, WA, PH, etc.). This allows each potion to achieve its distinct effect when the methods are called. The *Item* class provides a *getItem* method that allows callers to access the item at a specific position. This ensures that the correct effect is applied to the player. Furthermore, the player has a “bag” of temporary potions, represented by a vector. Once the effect of a temporary potion is applied, it gets added to the “bag”. This is managed by the *Potion* classes. When the player reaches a new level, *undoEffect* is called on all potions in the player’s “bag”, in order to eliminate the effects of temporary potions. We decided not to use the Decorator Pattern to retrieve player’s initial Atk and Def due to its complexity and risk of causing memory errors. Using a vector to store the consumed potions is more straightforward, easy to understand and maintain.

5. How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

   To avoid code duplication, we could use polymorphism with Item as the base class, and Potion and Treasure as the subclasses. In the base class, we could define virtual or concrete functions that are shared by both subclasses, as well as fields including the amount, x, and y coordinates. The more specific functions are defined or overwritten in the subclasses. Furthermore, we could use an *ItemFactory* and enumeration to manage

the generation of items, and this would enable the generation of Treasure and Potions to reuse as much code as possible.

Extra Credit Features

We implemented two of the suggested bonuses in the pdf by implementing an inventory for the potions. This is an interesting OOP-design problem since we considered the bonus after implementing all of the functionalities, so we realized that the decorator of the Potion added in inventory would be deleted when the player levels up, which would cause issues in accessing the freed memory block of *Potion* item. We then solve this problem by constructing a new Potion whenever it is picked up and not linking it into the Dungeon AsciiArt. This way it will exist as its own entity and will not be affected when the level is clear. The memory leak problem is also addressed in the player destructor as it will destroy the inventory when everything is either restarted or exiting the game. We also implemented a Merchant shop that can be accessed whenever and take a gold coin to purchase. These bonus features really tackle our design thinking and able to really deepen our understanding of OOP design to solve intertwining problems that are not discussed in class.

We didn't have the time as we wished to implement a random generation map, but the included chamber{.h, .cc}, if implemented correctly with *CheckCoord* class, will randomly pick a generated chamber and spawn the necessary items/characters inside it.

Final Questions

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 learned more about code management tools and techniques including git push and pull, and merging conflicts. We recognized the importance of active communication to address any issue or propose any suggestions, especially when different components need to interact with each other in the program. We resolved conflicts in different ways which in turn led us to learn how to branch, live-sharing code, and resolve painstaking conflicts. We realized the importance of code documentation and maintainability

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

We would outline the overall structure and design of our project before delving into the actual code. This includes considering how various design patterns and inheritance can enhance the structure, clarity, and reusability of our code. By employing these techniques, we can ensure that elements interact logically and seamlessly. Additionally, we would dedicate more time to investigating special cases that may arise during the game, and develop solutions to effectively address them. By integrating these design patterns and thoroughly considering special cases, we could enhance our program's organization, effectiveness, and ability to handle unexpected scenarios.

Conclusion

In conclusion, our CS246 Spring 2024 CC3K Project has been an invaluable learning experience in object-oriented programming, design patterns, and team collaboration. Through the development of our dungeon game, we not only improved our technical skills but also grew in terms of our ability to communicate and team work. This project has highlighted the importance of planning, active communication, and thorough testing, all of which are crucial for successful software development. Moving forward, we feel better equipped to tackle complex programming challenges and collaborate on future projects with confidence and efficiency.