CC3k Project Summary

Christopher Jiang

Jessie Zhang

Andrew Lu

Overall Structure:

In this project, my group and I worked together to create the game known as Chamber Crawler 3000, a simplified rogue-like game. The overall structure of this game is divided up into four sections: player, enemies, floor, and entity, which is highlighted by the file structure of the submission folder.

The player portion of the game includes subclasses for each individual race that the user can choose to take on (*Shade, Drow,Goblin, Troll, and Vampire*). Furthermore, there is a playerFactory class used to segment the logic for creating each race. This is known as the Factory Pattern, which will be discussed later on in the report.

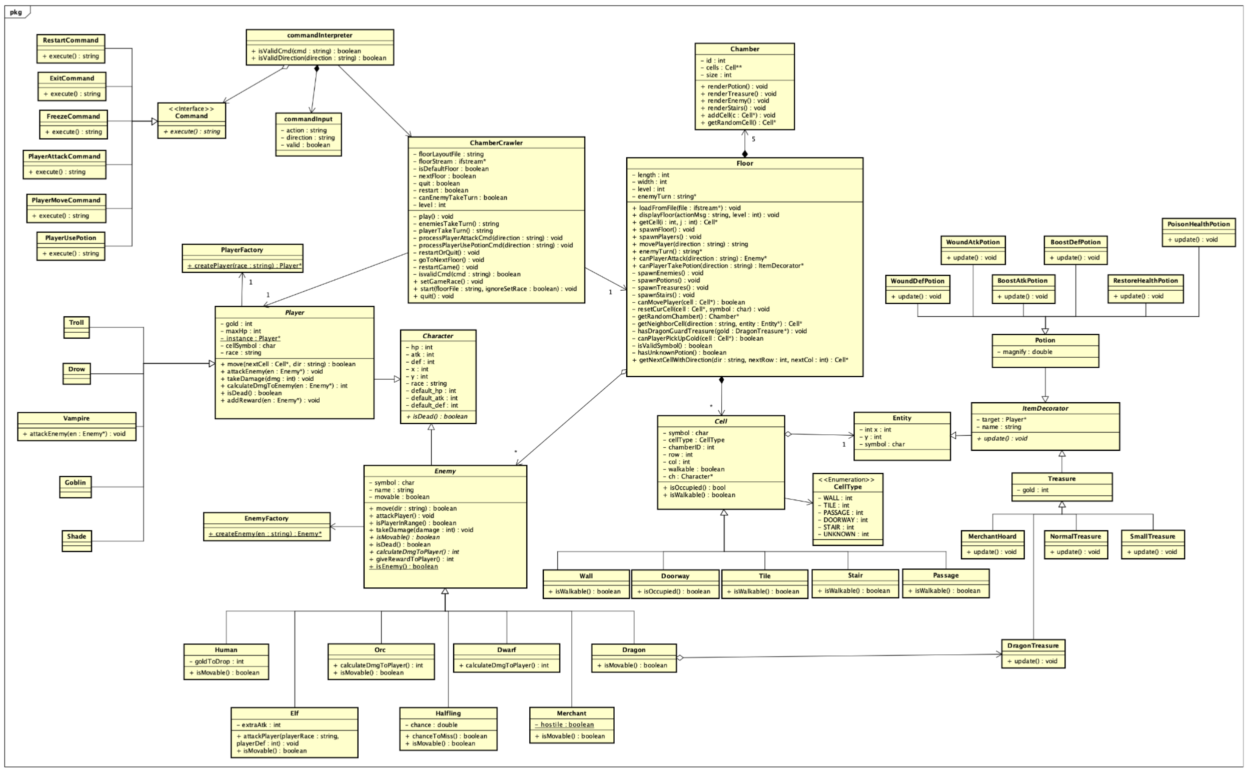
Similarly, the enemy section of this project was also implemented using a similar idea. It includes classes and header files for the race of every enemy: i.e. (*dragon, dwarf, elf, halfling, human, merchant, and orc*) Again, the factory pattern is also utilised here with an enemyFactory class used to segment the generation of enemies.

The floor section has the structure as follows. A floor has five chambers, highlighted by an array of Chamber pointers. Each chamber keeps tracks of a list of cells, implemented via a vector that tracks cell pointers. A cell has five subclasses (*doorway, passage, stair, tile and wall*) which are tracked using an Enumeration.

The entity portion has primarily two sections: an itemDecorator class with two subclasses: potions and treasure. There are six concrete decorators extending from the potions subclass, one for each potion (RH, PH, BA, WA, BD, WD). Similarly, there are four subclasses extending from the treasure decorator, one for each type of treasure (Small Hoard, Normal Hoard, Merchant Hoard, DragonHoard).

The starter class called *ChamberCrawler* is an instance of the game which contains a pointer to floor, a string to the contents of the floor file, as well as a filestream used to read files from the file system. The program is compiled by a unique Makefile, which will be discussed later.

UML



Design Implementation:

Singleton Pattern:

The singleton pattern was implemented in the Player class. This pattern is often used when limiting the number of instances of a class is necessary. This design pattern is the perfect solution to implementing the *Player* class, as only one instance of the *Player* class is necessary throughout any point of the program. An alternate way to implement the *Player* instance is via a global static variable available to every class and file. However, the singleton pattern is the better option. It allows for encapsulation and hides the implementation detail of the instance creation and management. Singletons also offer better-controlled access. Unlike a global variable, which can be accidentally modified anywhere in the program, a singleton provides a central access point that will allow me to control when and how the instance is accessed and changed. Lastly, we can conduct lazy initialization with a singleton, meaning it will only be initialised when it is first called. In contrast, a global variable is initialised at the program's start regardless of immediate use.

The Singleton pattern's defining trait is the static instance in its class definition. This can be seen in the static variable called "instance" in the player.h file. This static variable allows global access and lazy initialization throughout the program. Another notable feature of the singleton pattern is a private constructor and destructor. However, due to implementing the factory pattern alongside the singleton pattern, the constructor was declared to be protected and the destructor was declared to be public. This allows us to maintain the Singleton Pattern's control but allows us to keep the Open Closed principle intact. (Open for extension, closed for modification)

Decorator Pattern:

The Decorator Pattern allows behaviours for objects to be added dynamically without affecting behaviours of other instances of the same class. This can be used to model the gold and potions classes as they add behaviour to the Singleton *Player* class. The Decorator contains three major components:

- the concrete component (in this case, the Player instance)
- the decorator interface (in this case, the Decorator.cc)
- concrete decorators (in this case, each of the potion subclasses)

By using the decorator class, we can add and modify the behaviour of our *Player* Instance at run time, increasing our code's flexibility. It also allows us to conserve two fundamental design principles: the single responsibility principle and the open-closed principle. The decorator pattern allows us to divide the responsibility of updating HP, ATK, and DEF into smaller and more focused classes making the code base easier to maintain, debug, and understand. Similarly, the decorator class allows us to make code open for extension and closed for modification. In other words, it is possible to introduce new potions and gold piles without modifying pre-existing code.

Command Pattern:

The Command Pattern is a design pattern that allows the user to encapsulate a request or an action that an object performs into an object. This pattern allows us to parametrize different commands and requests. This pattern was used in *ChamberCrawler.cc*, specifically in the *playerTakeTurn* function, where user input is processed, and a command is run based on the input. Using the command pattern, the client code, *ChamberCrawler.cc* does not need to know the specific details behind each command it uses. This makes the code easier to maintain and understand. The command pattern has some defining characteristics. It must contain an interface to declare the execute function and concrete command classes for each necessary command. The command pattern must also be able to parse the necessary input to determine which command needs to be executed. Another class *CommandInterpreter* is used to parse the user input into another *CommandInput* object and to execute the concrete command. Our solution only partially implemented the command pattern due to limited time fully.

Factory Pattern:

The Factory Pattern is a creational design pattern that offers an interface for constructing different classes of objects. There are three subclasses of factory patterns, simple factory, factory method, and abstract factory. In our code, we made use of the simple factory. This pattern is a concept that encapsulates object instantiation. Some defining trait of the factory class is the create object function, passed a char as the switch case for the enumeration used to create different objects. The simple factory pattern was used in two places in our specific implementation, *Player* and *Enemy*. Since there are 5 to 6 different types of *Players* and *Enemies* in this given project, segmenting and separating the logic behind creating the instance(s) is crucial to keeping code tidy and maintainable. The factory pattern also encapsulates the object creation process by hiding the logic from the client. This encapsulation can therefore centralise object creation, promoting the high cohesion, low coupling design concept.

MAKEFILE

The MAKEFILE for this assignment is not the one offered by the course. The unique MAKEFILE searches through all the .cc files under the SRC_DIRS list and generates .o files accordingly, which will then be used for compilation. The searching is accomplished using wildcard matching as taught in CS136L. Another difference that this MAKEFILE offers is that it leaves all the dependency files (.d) and object files(.o)in a separate folder named build. All in all, the choice of making a unique MAKEFILE was to make the build process seamless so in the future we don’t need to update the MAKEFILE everytime new source code files are added.

Constants.h:

One of this assignment's primary concerns and challenges is the absurd number of constants each module requires. It would be tedious and untidy to include constants in every header file. Therefore, to resolve this issue, I created a header file named constants.h. Anytime a specific implementation needs to use constants, I include constants.h in the header file for the implementation. The use of constants.h reduces the size of each file substantially and makes the code more readable, testable and maintainable. Anytime I need to change constants to test a specific functionality of the implemented code, I can immediately locate what needs to be changed.

Resilience to Change

The MAKEFILE for this assignment is not the one offered by the course. The unique MAKEFILE searches through all the .cc files under the SRC_DIRS list and generates .o files accordingly, which will then be used for compilation. The searching is accomplished using wildcard matching as taught in CS136L. Another difference that this MAKEFILE offers is that it leaves all the dependency files (.d) and object files(.o)in a separate folder named build. All in all, the choice of making a unique MAKEFILE was to improve and compile a code base with better readability. When designing and planning the program and its implementation, I kept the design principle open and closed principle in mind. Therefore, if and when additional specifications and requirements are included, minor modifications will be needed to the existing code and outside of adding the necessary modules. One excellent example is the item decorator. By implementing a decorator, we leverage the idea of using composition over inheritance. The Decorator class has an instance of *Player*, which is a composition relationship. With this instance, we can dynamically modify the objects without creating more classes with the modified behaviour. Furthermore, these Decorators can be stacked on top of one another, allowing us to simulate the behaviour of using more than one potion per floor. In the future, if more potion types need to be implemented, we can quickly implement these by extending the new potions class onto the itemDecorator class, which is then available to decorate and modify the player class. More specifically, the decorator class is used to model the game's potions and treasure hoards components.

Another example of resilience to change is the simple factory pattern. The simple factory pattern allows us to segment the logic behind creating the object into a separate class. We abstract the logic behind the object creation from the client code. This decoupling reduces the dependencies between different modules. The factory pattern was used on the Player class and the enemy class. Therefore, if we need to add new races to *Players* and *Enemies*, i.e. (fairy or basilisk), we do not need to change the client code, such as *floor.cc, Chamber.cc* and *ChamberCrawler.cc*. We only need to change the *playerFactory.cc* or *enemyFactory.cc* to introduce new stories into the game.

Answer to Questions:

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?

As mentioned previously, we used a factory pattern and a singleton pattern to isolate the logic behind object creation to make adding new races an easy process.

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

Enemy generation utilises the same logic as Player with a simple factory pattern. Upon retrospect, this may need different solutions as the currently implemented solutions require *dynamic_cast* to do type checking which is error-prone and generates bad code smell. However, due to the time limitation and lack of assistance, it was the best I could come up with.

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

The enemy characters also use a Factory pattern with an Enemy base class. The subclasses can override methods and properties to provide unique behaviours or abilities against certain races. A function was defined on the base class for every special ability of enemies, and we took advantage of inheritance to implement these special abilities.

The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

The strategy pattern allows you to change the implementation of something used at runtime. The decorator pattern allows you to augment (or add to) existing functionality with additional functionality at runtime. In this case, our group decided to use the decorator pattern in order to model potion and gold generation as we are augmenting and decorating to our existing player class and not changing any specific implementation of the Player class.

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?

We have chosen to employ the Decorator Pattern to minimise code duplication and adhere to the design principle of Separation of Concern. To achieve this, we created an abstract class called ItemDecorator, which is a subclass of entity. In this class, we keep track of the common themes between Treasure and Potion (the target in which these items modify and name of these items). By doing so, each concrete subclass does not need to maintain this information. Any similarities for potions (the magnification for Drow) and any similarities for Gold are implemented respectively in treasure.cc and potion.cc. Only the update function is implemented individually due to each item's unique properties.
To avoid duplication, I used the advantages of inheritance to generalise the similarities between potions and treasures to enhance code reusability. Another possibility is to adopt Abstract Factory Pattern to create concrete factories such as PotionFactory and TreasureFactory to create various Potions and Treasure. However, due to time constraints, we did not implement this approach.

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?

I learned that large projects often contain large codebases, which are often difficult to maintain and debug. Estimating the specific time needed for a particular portion of the project is also often challenging. Therefore, my teammates and I should have adopted the agile scrum approach, a framework for effective collaborations on complex projects. It involves frequent check-ins and inspections of the quality of work. I tried my best to implement this work process. However, their busy schedules made it hard to follow the intended plan.

Furthermore, to ensure each aspect of the program works properly, we need to develop a rigorous regression testing suite. Unfortunately, again due to the busy schedules of my teammates, I was unable to do so and started properly testing the result much too late, causing unnecessary and undue stress toward the end of the project. Finally, it is essential to read over all of the project specifications. I only realised sections 6 and 7 a few days before the due date, and coupled with barely any communication with my teammates, led to unnecessary and undue stress later on in the project. Lastly, I learned that it is essential to find group members with the same passion and dedication for the subject and project as you. I knew it was important to have motivated teammates who did not sit around and wait for someone else to complete the assignment. We can only put forward our best work and create the best possible product when working with people with similar interests.

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

Overall, I am satisfied with the implementation. If given more time, I would like to read and understand the specifications thoroughly, as it is the foundation of successful development. A clear understanding of the requirement can also help the group plan and implement the code efficiently.

I also want to start testing my code early instead of after the development work. I should adopt a test-driven development (TDD) approach to help me to reduce the bug. Given more time, I should focus on refactoring my code and seek feedback from my group members.