<u>Design Document</u>

Overview

This project, ChamberCrawler3000, was built using object-oriented methodologies in the C++ programming language. The goal of this role-playing game is to navigate through a dungeon, while slaying enemies and collecting treasure along the way until the end if reached.

Map

The game consists of five floors. Each floor is comprised of a 79 x 25 grid which is entirely populated by cells. Its layout consists of the same predefined configuration of five chambers with connecting passages. As a result, each floor is separated into floor sections known as chambers, passages and caves (which represents no man's land). The floor is implemented using **inheritance**. Subclasses Chamber, Passage and Cave inherit from a FloorSection superclass. Furthermore, each cell is categorized as either a wall, stair, tile, doorway or passage piece as a result of the **decorator pattern**. This allowed custom attributes and methods to be added to each cell type without affecting the behaviour of other cells. Specifically, the implementation of the decorator pattern on the cells allows the player to move through passages and doors, while enemy movement is contained within a chamber.

The Board class is implemented as a **singleton pattern** in order to restrict the instantiation of this class to one object throughout the system.

Setup

At the start of the game, the location of the player, staircase, enemies and items are randomly spawned across the floor, based on the specific probability distributions defined in the project specification sheet. This randomization is achieved through a series of helper functions that rely on random number generation in order to reduce the duplication of code. For example, one helper function randomly selects a chamber, while another function randomly selects a cell within the chosen chamber. These two helper functions are shared by enemies and items.

Characters

There are two types of characters in the game, players and enemies. The Character superclass has a set of base fields and methods that abstracts the similarities between the two types of characters.

The player is implemented using the **singleton pattern** to once again restrict the instantiation of this class to one object throughout the game. At the start of the game, the player has the option of choosing between the following player races: shade, troll, goblin, vampire and drow. Each race has its own special abilities. Player is implemented as an abstract subclass of Character, with each player race implemented using **inheritance**. With this implementation, it would be extremely easy to add additional player races simply by creating a new subclass that inherits from the Player class. Furthermore, inheritance allows each race's abilities to be customized simply by overloading the base methods.

Similarly, the Enemy class is implemented as an abstract subclass of Character, with each enemy type implemented using **inheritance**. Each enemy is either a human, a dwarf, an elf, an orc, a merchant, a dragon or halfling. Like with players, each enemy type's custom abilities can be customized by overloading the Enemy base methods. Enemies are generated using the **factory method**. This allows us to create enemies without specifying the specific class or directly calling the constructor of each enemy type. Specifically in this program, a spawning method was written that employs random number generation to select an enemy type, call its constructor and return a pointer to it. Only in this function will the constructors of enemies be called. Thus whenever an enemy is generated, one does not need to explicitly pick an enemy type constructor. Instead, the factory method is called which ensures random enemy generation, while reusing as much code as possible.

Enemy movement is randomized within the confines of the chamber they are spawned in. In order to achieve this, random selection helper functions described above were written.

Combat

Combat in this game is implemented using the **visitor pattern**. Since some player and enemy special abilities require the type of both the enemy and the player to be known at runtime, this pattern will allow for such interactions. For example, orcs are able to do 50% more damage to goblins, so the orc class's attack method will be overloaded. By doing so, the **double dispatch mechanism** calls different functions depending on the runtype types of the enemy and the player involved in the call.

Items

The Item class is a subclass of the Entity class. The Entity class essentially represents all objects in the game that take up a tile on the board (player, enemies, gold, potions).

Piles of gold of different value depending on its type can be found across the floor. Gold can be collected when players walk over it. As well, piles of gold are dropped upon the death of an enemy.

Potions are the only type of useable item in this game which have either a positive or a negative effect on the player. Certain potions such as Restore Health add HP to the player without exceeding the maximum prescribed by the race, while potions like Poison Health subtract HP from the player. However, the effects of certain potions are not permanent. With Boost Atk, Boost Def, Wound Atk and Wound Def, the effects of these potions are only limited to the floor they are used on. These temporary potions are modelled using the **decorator pattern** so the effects of potions do not need to be explicitly tracked. Whenever the player consumes a temporary potion, an enhancement decorator is added on top of the player to overload the getters of the Atk and Def fields. This decorator layer ultimately allows multiple potion effects to take place without explicitly having to track the potions consumed. Finally, when the staircase is reached and the player advances to the next floor of the dungeon, the decorator layers can be discarded, removing the effects of the temporary functions.

Display

The display of CC3K consists of two classes, Map and Info. The Map class is responsible for depicting the state of floor to the player, including chambers, walls, stairways, passages, doorways, tiles and caves. As well, the location of the player, enemies, potions and gold is also shown on this map.

The Map is implemented using the **observer pattern**. The Map class maintains a list of its dependents, also known as the observers, which notify them automatically of any state changes. For example, every time the player moves to a different tile on the board, a pile of gold is collected, a potion is consumed or an enemy is slain, the respective notify methods are called and the Map is updated to reflect these changes.

Similarly, the Info class also relies on the **observer pattern**. The Info class is responsible for displaying player information such as HP, Atk, and Def values. With this design pattern, whenever the player is attacked, the Info class is notified. The player's updated HP is then displayed to the player. Similarly, when the player reaches a new floor, a method is called to notify the Info class of this change.

Interface

The player character is controlled through the command line through a series of predefined inputs. For example, players can specify the direction they want to move in, the direction they want to attack in or the race they wish to be at the start of the game. In order to achieve this functionality, the program reads from the **standard input stream**.

UML Diagram

The UML diagram summarizes the architecture behind the program including all classes, fields, methods and the relationships between classes. Please see uml.pdf in the zip folder for the diagram.

Reflection

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?

Thanks to our limited programming careers, we have grown accustomed to working alone, mostly on side projects in our own spare time. As a result, this project has been a very eye opening experience, and we have learned about the importance of planning and communication when developing software in teams.

First of all, we completely underestimated the importance of planning, especially with respect to the design and architecture of the program. We hastily put together the UML diagram so that we could begin coding right away. We didn't put nearly as much thought into the implementation as we should have. Through this, we learned that a significant portion of time and energy should be spent on the planning stage, especially when writing large programs,

Secondly, we realized how important communication is in software development. There are numerous misconceptions surrounding this industry - programming is often portrayed and ridiculed as a solitary task. However, this is completely untrue. Communication is key to the process and makes all the difference. It allows teams to complete projects efficiently and productively. Beyond splitting tasks and responsibilities, we found it extremely helpful to update one another on not only our progress, but the challenges we have encountered as well. By thinking through problems and discussing potential solutions together, we were able to minimize the amount of time and effort wasted.

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

If we could start over, we would have definitely tackled the project differently. First of all, like we mentioned in the previous question, we would have spent a lot more time on the design of the program and thinking about the implementation before we started actually coding. Secondly, we would have split up tasks differently because we encountered a lot of version control issues such as merge conflicts. By having each of us focus on a different section of the codebase, we would be able to avoid making modifications to the same files at the same time. This was a result of poor planning and lack of communication. Finally, we would have taken a much more iterative approach. Instead of following the priorities listed in the specification sheet (get the board to work first, then implement general players, enemies and items, then introduce special races and types), we tried to tackle everything at once. As you can imagine, this turned into a massive debugging nightmare. If we could start the project over again, we would have definitely written the generalized base classes first and tested them individually before trying to get fancy. In the end, we ran out of time. Lesson learned.

<u>Questions from Due Date 1</u>

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

Each character race could be easily generated using inheritance. We would have a base class called Character with a set of base fields and methods that captures the similarities between all. Then, we would create a Player subclass which inherits from Character, adding player functionality through new fields and methods or overloading previous fields and methods. Then, we would create Player subclasses which model each character race in a similar manner. With this implementation, it would be extremely easy to add additional races simply by creating a new subclass that inherits from the Character superclass which have all the base functionality already defined. Then, we would simply have to overloaded the base methods to customize the new class.

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

Different enemies would be generated using the factory method. A generate enemies method will be written that will use random number generation to pick an enemy, create it and return a pointer to it. Only in this function will the constructors of enemies be called. Thus whenever an enemy is generated, you do not need to pick an enemy constructor. Instead, you just call the factory method which ensures random enemy generation. This would be different from how the player character will be generated; since there should only be one instance of the player at any time in the game, the player is implemented as a singleton. So a get player function will be used to generate or get the pointer to the player.

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 custom abilities of the enemy characters will be made through a mix of overloading base abilities and the visitor pattern. All enemies will have a set of base methods for actions such as attacking and being slain. When a race specific ability changes how these base methods work, a new function which will overload the base function will be written. Since some special abilities require the type of both the enemy and the player at runtime, the visitor pattern will be implemented to allow such interactions (for example, orcs being able to do 50% more damage to specifically goblins). The same technique will be used for the player who will also have a set of base actions that will be overloaded by unique abilities since there are no differences between the player and the enemy that would prevent this implementation from working.

What design pattern could you use to model the effects of temporary potions so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

The effects of temporary potions could be modelled using the decorator pattern so the effects of potions do not need to be explicitly tracked. Whenever the player consumes a temporary potion, a enhancement decorator is added on top of the player. This enhancement decorator will overload the getters of the attack and defence field of the player and alter the results of the getters depending on the specific potion(s) consumed. The layer of the decorator will allow multiple potion effects to take place without explicitly tracking the potions consumed, and when the floor is passed, the decorator layers can be discarded which will remove the effects of the temporary functions.

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 generation of treasure does not duplicate code?

Treasure and potions are both items which share several similarities. These similarities can be generalized in order to reduce code duplication. First, the random cell selection procedure for both are the same. Thus, we can write a function that randomly selects a chamber and then randomly selects a floor cell within the chosen chamber. This random selection function would be shared and used by both items to randomize location. In addition, the type of the potion or treasure generated must be random. Since the base logic for this is the same, we can also generalize this type selection function. In this way, through generalizing the two functions - one for randomly generating the item location and one for randomly generating the item type - we can reuse as much code as possible.

<u>Deviations from Due Date 1 Plan</u>

The architecture behind the program is more or less the same as the one outlined in our UML in terms of the classes and the relationships between the classes. However, we have made several modifications to the fields and the methods of certain classes. For example, once we started coding, we realized that we missed a couple getters and setters so we had to add those. As well, we realized that some of our classes didn't store all the information we needed to keep track of such as enemy count, floor level and race type, so we had to add those.