# CS 246 Project - Final Design

CommandCrawler3000 (CC3K)

Aryan Sureka(asureka), Cecilia Qiu(c32qiu), Lisa Huynh(127huynh)

University of Waterloo

December 15, 2021

# Overview

Our version of CC3K contains the fundamental class Board, which is responsible for managing the states of the floors throughout the game, as well as the game pieces. We decided to make a parent class called Piece, for which all game pieces inherit fields like location on the board and piece type (‘P’ for Potion, ‘G’ for Treasure, or Character). Then, subclasses Race, Enemy, and Dragon are inherited from the Character subclass, where Character has methods for default attack (a basic attack method that disregards special combat abilities of the attacker and the defender), getting character stats (HP, Attack, and Defence), and altering character HP. Since potions and gold do not have such stats, they both inherit directly from Piece. The Potion subclass only contains its piece type (‘P’) and exists for the purpose of being a game piece on the board. We chose to implement a strategy pattern, in which the Board class and the Strategy interface have a Has-a relationship. There are six subclasses of Strategy, one for each type of potion, each having a different implementation of usePotion(Race &PC). On the other hand, the Treasure class has subclasses: SmallPile, NormalPile, MerchantHoard, and DragonHoard, of which contains its gold value.

We separated the Dragon class from the Enemy class because dragons behave differently from all enemies. Instead of moving randomly after every turn, a dragon stays stationary and attacks only when the player is around itself or its dragon hoard. Hence, the Dragon class has exclusive methods to check for players within it and its hoard’s radius, as well as altering its dragon hoard’s location. To further categorize the Character subclass, the Race subclass contains exclusive methods that alter Attack and Defence, which are only used on the player, and an attack method that takes into account the special combat abilities of the player, enemy, and dragon. We made a subclass for each race in the game: Shade, Drow, Vampire, Troll, and Goblin, which all inherit from the Race class. In addition, the Enemy subclass has exclusive methods to deal with changing enemy hostility, no duplicate movements, and generating random gold value to be added to the player once the enemy is slain. We have a subclass for each enemy in the game: Human, Dwarf, Elf, Orc, Halfling, and Merchant.

# Design

There were multiple design challenges in this project, one of which was the structure design to implement the different potions without needing to keep track of the effects of each potion spawned. We decided to implement the Strategy pattern for potion effects. To do so, we created an interface called Strategy, which has a virtual method of usePotion(Race &PC). Then, we made six subclasses of Strategy, one for each potion. In these subclasses, we overrode usePotion(Race &PC) and implemented the method with the desired potion effect. In our object Board, we have a shared pointer of Strategy to hold the strategy pointer, and a vector of shared pointers of strategies, each pointing to a specific potion strategy. At runtime, whenever the player correctly uses a potion, we call a function to randomly generate an index of the vector of potion strategies and set our strategy pointer to the resulting potion strategy. We then call the usePotion(Race &PC) method of our strategy pointer, passing in the reference to our player. Doing so allows us to change potion strategies at runtime by simply alternating implementations of usePotion(Race &PC).

Another design challenge that we faced was handling the attack between the player and an enemy. Initially, we planned to implement the attack methods using the Visitor design pattern, where attack() would replace accept() and attackedBy() would replace visit(). By using the visitor pattern, we would have been able to call the attack method with a smart pointer of any of the Race subclass or Enemy subclass and the program will invoke the appropriate method for that parameter. We would be able to implement specific combat abilities without having to check which race or enemy the player is and what race and enemy the opponent is. However, we ran into a problem when we were passing in smart pointers of the parent class Enemy into a function that takes in a smart pointer of a subclass of Enemy. The same problem occurred for function calls passing a smart pointer of a parent class Race into an attack method that took in a smart pointer of a subclass of Race. This occurred because we were keeping track of our game pieces via vectors of smart pointers of parent classes Enemy and Race. Though this allowed us to group child-classes together and better manage them, we were unable to implement methods that took in specific subclasses. Given the time constraint, we prioritized completing the program and decided to work on implementing the Visitor pattern after we have the base game. We instead defined attack methods that took in parent classes Race and Enemy and implemented the combat

abilities of each subclass through check conditions. As much as it hurt us to put the idea aside, we had to prioritize completion.

In our project structure, we used all four object-oriented programming principles to ensure we had high cohesion and support low coupling. Firstly, we used encapsulation by separating Potion, Treasure, and Character into different subclasses. This allowed us to keep variables like HP, Attack, and Defense of characters private and accessible only through the given methods. This also encouraged high cohesion, such that Character is only responsible for actions involving characters, and Treasure is only responsible for actions involving gold. Our program also has low coupling because the classes are not strongly connected to each other. If we were to change a field in Potion, it would not affect Character in any way. We used abstraction in the attack methods of our Race and Enemy classes. In these attack methods, we call the defaultAttack(Character &C) method from the parent class Character. Although Race and Enemy do not know how it is implemented, they are able to use it to perform their attacks. To continue, we used inheritance by having Potion, Treasure, and Character classes inherit from the Piece class. This means that they reuse the position fields in Piece while having their own unique fields and methods. Lastly, we used polymorphism in our Strategy pattern, where usePotion(Race &PC) has multiple implementations depending on the potion strategy used. By following the principles of object-oriented programming, we have created a project that has high cohesion and supports low coupling.

# Resilience to Change .

We used the strategy design pattern to implement the different effects of the potions. With this design, we are able to easily accommodate additions of new potions. To add a new type of potion, simply create a new subclass inherited from Strategy and define the implementation of usePotion(Race &PC) with the intended effect of the player. Then, make a shared pointer with the type as the new potion class and append it to the vector of potion strategies. Assuming that we want an equal chance of spawning for all potions including the new potion added, the call to randomly choose a potion strategy uses the size of the vector of potion strategies, thus we do not have to change anything else. By using the strategy design pattern, we are able to easily add new potions to our game and change effects of existing ones.

If we need to change the default HP, Attack, or Defense of a Race or Enemy, we can simply make the change in the constructor, and it will be carried out through the program. If we want to add a new subclass of Treasure, we can simply make a new subclass inherited from Treasure, and define the constructor with the intended gold value. Then, include the .h file for

this new class in Board.cc, and you can alter the spawn rate of the treasures by changing the probabilities. The game will automatically spawn the new class appropriately and the player will be able to pick up the gold in the game.

Should we want to add a new race or enemy, we can simply create a class of the new race inherited from Race or a class of the new enemy inherited from Enemy. Then, define the constructor with the intended type, HP, Attack, and Defense. Since we move enemies based on the 2D vector of characters, we would only need to add the condition for if the character at that index is the type of our new enemy subclass, to our call to attack(Race &PC) and moveEnemy(int x, int y) methods. Then, if there are no special combat abilities, attacks made on this new enemy subclass or race subclass will be the default attack method. If we want to add a special combat ability, then we can simply add that condition into the attack(Race &PC) or attack(Enemy &E) methods implemented in the Race and Enemy classes. If we were to add a Race, we would need to add a condition to let the user choose that Race, then call the constructor for it. That is, all we need to do if we were to add a new race or enemy is add to one file and create a subclass.

If we want to change the floor plan of the game, we can simply redefine the vector of vectors of pairs of pairs (we apologize that you had to read that) that holds all the rectangles composing each chamber. Then, the methods that produce a random position on the board to spawn a game piece will function normally with that updated variable. This enables us to experiment with different floor plans, note that we would have to provide the program a text file with the layout of the board. We can also easily change the radius of movement of the enemies. To do so, we can increase the 2D loop to search for unoccupied spaces around each enemy to search within a radius that is greater than one. In addition, we can also change the number of floors in the game, which in turn changes the condition of winning. This is possible by simply changing the maximum number of floors the player can climb.

# Changes to Project Design

There were a few changes between our initial UML diagram and our final UML diagram. We initially thought that we could combine the Strategy pattern into the Potion subclasses, hence the multiple subclasses of Potion in the first UML. However, we realized that that did not follow the true purpose of the Strategy pattern. If we were to put potion strategies into potion subclasses, that would defeat the purpose of the Strategy class. Since that would entail not changing implementation at runtime, and we would unknowingly keep track of each potion effects in the game. Hence, we decided to create an interface Strategy that has a virtual method and instead create subclasses of that interface for each potion strategy. Aside from this, we removed all private fields from the class definitions and added many more public methods.

# Answers to Questions

Q3: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.
Answer: In our initial plan, we wanted to implement the various abilities for the enemy characters the same way that we implement the various abilities for the player races, through the Visitor pattern. However, due to complications explained in the Design section above, we implemented the abilities of enemies in the attack method that attacks the player. In this method, we check which enemy type we are, and what race the player is before deciding how to attack. This is the same for the player, we implemented the abilities of players in the attack method that attacks the enemies. In this method, we check which player race we are and what type of enemy we are attacking, before deciding how to attack.

# Extra Credit Features

We have implemented our game with no memory leaks, without explicitly managing our own memory, via STL containers and smart pointers for memory management. This was our goal from the start, so we planned the game with this principle in mind. For example, we used vectors of shared pointers to keep track of all of our game pieces. We also used vectors of pairs to find unoccupied areas around a character. We had to be careful with cyclic behaviour between the shared pointers but after we got the hang of using smart pointers, managing memory was much easier throughout the game.

# Final Questions

Q1: 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?
Answer: Working on CC3K as a team of three showed us the magic and setbacks of teamwork in a programming setting. While working on a big project, it is crucial that we plan well. We had difficulty gauging how much planning is enough before DD1 since we were all used to going with the flow. As we proceeded with our plan after DD1, we realized that we had to make

changes to our class structures and method implementations. These complications set us back a few days, which could have been avoided had we planned it out better. In addition, we also learnt that communication is crucial in a teamwork environment, especially when it comes to working on the same file at the same time. Although the LiveShare extension on VS Code enabled us to work together on the same file, we had difficulty debugging/implementing our own methods because there were multiple changes happening in the same file. This made it hard to compile the game because we had to ask the others if they were finished on their ends. However, this project was the perfect learning experience to get a feel for what it is like to develop software with a team.

Q2: What would you have done differently if you had the chance to start over?

Answer: If we had the chance to start over, we would dedicate more time to working on this project earlier on as well as make a more thorough plan. By doing this, we would have a much better headstart and the coding components will be smoother, which allows us to spend more time on code efficiency. If we had more time, we also would have liked to experiment with the Visitor design pattern for attacking between player, enemies, and dragon.

# Conclusion

All in all, this project was a valuable learning experience. We were able to implement object-oriented programming principles and it encouraged us to conduct self-guided research for better practices. We learned how to work in groups and how to program software in a collaborative environment, better preparing us for the workforce. Although this was a relatively small group and a relatively small project, it was an excellent experience and we enjoyed every bit of it.