# Final Design Document (CC3K)

## Initialization

*The Board*

(random/specified) CC3k constructs the board by creating a Board object which contains Tiles and Chamber objects. These get initialized and then each Tile is assigned to its corresponding Chamber.

*The Player and Stairs*

(random) CC3k constructs the Player by picking a Chamber, then the Chamber will randomly assign the player to one of its Tiles. The Stairs will be constructed and when picking its Chamber, it will select a random Chamber that is not occupied by Player.

(specified) CC3K will construct the Player and Stairs object and add it to the specified Tile.

*The Enemies, Potions, Gold*

(random) CC3k constructs *n* enemies by randomly calling random Chambers *n* times to add a random enemy. The same is for Potions and Gold. The Chambers will keep a list of enemies for moving them later on. When a Dragon Hoard is generated, the Chamber will automatically generate a Dragon on one of the empty tiles surrounding the Dragon Hoard.

(specified) CC3k constructs each board object according to its symbol and adds it to the specified Tile. When adding an Enemy to a Tile, CC3k will also update the Tile's Chamber's list of enemies.

## Movement

Whenever the Player makes a move. CC3k calls all its Chambers to move the enemies on their list of enemies and then the Chamber will call all the Enemies to move.

The Enemy will search its surrounding Tiles, if the Player is within one of those Tiles, it attacks the Player. Otherwise, if it doesn't find a Player it moves to an empty surrounding Tile.

## Attacking and Defending

When a Player or Enemy attacks, it calls the defender's defend function with its own reference, allowing the defender access to the Attacker's stats to calculate the damage dealt by the attack. This is implemented using the **Visitor Pattern**.

## Using Potions

When a Player uses a Potion, it gets the Potion object from Tile and extracts the potions stats. This is implemented using the **Visitor Pattern.**

## Display

CC3k uses a text display on the console. The display observes each Tiles on the Board and the surrounding Tiles of a Tile are its observers. Whenever a Tile changes, it notifies the text display and its surrounding Tiles. This is implemented using the **Observer Pattern**.

We decided to use the Observer Pattern in case we had time to create a real graphical display. But due to time limitations we were only able to create a text display.

## Use of Inheritance

When designing our program, we noticed that there were many relationships between classes. To maximize code-reuse, we introduced various levels of inheritance. For all objects on the board, they share core features that are present in all objects to reduce code redundancy.

The board objects all had a character representing the respective object and a pointer to the current tile that the board object is on. We made this the BoardObject class and made it abstract.

Furthermore, we used single inheritance for Potion, Gold and Stair objects, where the superclass was the BoardObject class. Although it was possible to introduce multilevel inheritance to encompass the various different types of potions, we found it rather unnecessary to have 6 different classes to represent the types of potions possible and instead just use one potion class that kept track of three stats to represent all 6.

We also noticed that between the player and enemies, there were many similar key variables that we would want to keep track of. These include their health points, attack and defence stats. We decided to used multilevel inheritance here. The superclass was the BoardObject class. The subclass was the character class, which had variables for health points, attack and defence as these were present in both Enemies and Players. The subclass of the character class was the Player class (with subclasses for Human, Orc, Dwarf and Elf characters) and the Enemy Class (with subclasses for Phoenix, Vampire, Werewolf, Goblin, Troll, Dragon and Merchant enemies). Through the use of inheritance in various parts of our program, it gave us the ability to add new types of BoardObjects with ease. If we wanted to add a new Enemy that could only move vertically, we could implement that just by overriding the move method for the class. Through our project, we realized how important it is to structure our objects and determine the relationships between them, as it’s an extremely powerful tool that can make programming a better experience.

## Reflection on original UML and design

Originally we had a decorator pattern for the temporary player. But, we removed that object because it required too much work when a simple temp field in the player class is sufficient.

The overall design concept and object tasks stayed the same. But the member functions had some changes to improve security. For example, we made the random number generator private since the other class does not need access to the random number generator.

A problem we ran into was using shared pointers for all allocation onto heap. But, we are also using the observer pattern. This caused the objects observing each other to not delete themselves. So, we had to revert back to raw pointers.

After doing the project, we realized the importance of planning and designing a proper UML. A huge portion of our time was used to code functions, figuring out it was missing something, then restructuring and recoding the class again. If we had planned out the project fully, it would have made the process much easier.

This project has also taught us the Object Orientated Programming benefits. We just needed each other's header files to use that class and its member functions. Also, when an object needed to be redesigned internally, we just needed to change the .cc file.

## Final Responses

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

We realized the importance of a version control system where we cannot accidentally modify a file and can easily retrieve old overwritten files. Collaboration between us would also not have been possible had it not been for git. Furthermore, we learned how important uml files are and how one should design their project first before diving into the actual programming. The uml file allowed us to split up the work with ease. It also allowed us to reduce coupling and increase cohesion. Not only that, it allowed us to use design patterns to carry out various aspects of our program like displaying text.

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

One lesson we learned is to use the student server for compiling. We ran into memory leak issues where our program wasn't at fault. We wasted hours trying to figure it out, but it turns out that Macs leak memory regardless of the program it runs. Furthermore, we should have followed the recommended order of components to complete for the project. We did it the exact opposite way. We completed all the basic components first, then the core components last. When we were testing the completed project it was a nightmare to debug and fix as we had never tested the program before. The bugs were in every other line of code and segmentation faults were everywhere as well.