CC3K Final Design Document

<u>Introduction:</u>

This document outlines the final state of our project, and details on the implementation of the game CC3K. It will also describe changes that were made to our proposed plan of attack since due date 1.

Note: Within the scope of this document, we will refer to user ailiashe as Anton, user c39gao as Charles, and user s243park as Huni.

Note: Names of classes will be *italicized* and Capitalized (ie *Hero*). Names of fields and methods will be *italicized* (ie *hostile* or *getX()*).

<u>Overview:</u>

We will assume that the reader is familiar with the general gameplay involved with the game CC3K. Our design section will outline the implementation of all our classes, and include details about what changes were made to the planned implementation from before we started the project. We did the project with modularity in mind, and so if we were to add new features to the game after completion, it could be done relatively easily. We also had time to implement a few bonus DLC features, described in the appropriate section below.

<u>Updated UML:</u>

Please see separately submitted UML.

<u>Design:</u>

Some miscellaneous notes about our design choices:

A note about the way we chose to handle invalid commands:

-Actual invalid commands, ie "sladfhsgda", were made to be ignored by the program.

-Commands that are invalid due to not having a defined interaction, ie calling attack on a *Tile* without an enemy, were interpreted as skipping a turn. *Monster*s can also skip a turn if they attempt to run into a *Tile* that is invalid, such as a wall. This decision was made in accordance to how the game series *Pokemon Mystery Dungeon* handles such cases, as CC3K is, in essence, a stripped-down version of that game.

Another note about our implementation choice: We had all *Floor*s generate at once at the start of the program, each being pre-loaded so that we do not have to generate them as the program progresses. To resolve the transfer of *Hero* data when progressing to the next *Floor*(ie current health, gold, held item, etc), we simply copied the fields that needed to be transferred and set the fields of the next *Floor's* player character equal to these values, thereby essentially transferring over the character to the new *Floor*.

Class: *BoardCreator*

This class was made to parse any input files, and create the *GameBoard* accordingly. Using the factory pattern design pattern, the class uses the set method of *Gameboard* to create the new object. If an input file is provided, then the class is responsible for parsing it, generate the monsters, hero, gold, and otherwise create assets for the game accordingly, ensuring that a "legal" game is created. Otherwise, the class will use the default floor plan, and call the function in *GameBoard* that randomly generates *Character* objects on the floor. The text file that contains the default floor plan must be specified when constructing a new instance of this class.

Class: *GameBoard*

This is the class that will contain, directly or indirectly, all the other components that make up the game. It will have a vector of *Floor* objects, an int indicating the total number of floors in the game, and an int indicating the player's current floor. This makes it easy to modify the number of levels, if we so choose to in the future. It will be displayed by a *TextDisplay* object.

The vector of *Floor* objects in *GameBoard* will contain all the *Floor*s preloaded with all monsters, items, and the player's position, generated by the *BoardCreator*.

Class: *Tile*

This class now uses an enumeration to denote the different types of tiles. It no longer has any fields (we had planned for it to have some in our plan of attack), as we have found that they are not necessary to implement movement for our *Character* objects. A grid of these *Tiles* populates a *Floor*.

Class: *Floor*

This class was made to handle generation of all *Character* objects. To accomplish this, we set fields in *Floor* to denote the number of *Monster, Gold* and *Potion* objects on the current floor, set by default to 20, 10 and 10 and stored shared pointers to each of these types of *Characters* as fields inside of this class. We then It also contains a vector of *Info* structures, which simply denote an x and y coordinate of a *Tile* of type room. We also gave *Floor* methods to check for whether a *Character* was allowed to move to a specific tile.

The *Floor* has a collection of methods to deal with *Character* interactions, such as combat, movement, potion usage, gold, dragon, etc. It also contains most of the checks that were required to ensure that a move attempted by the player was a valid one.

Since the *Floor* deals with both generation and asset interaction, it was the most involved class to write the implementation for.

Class: *Score*

This is a new class that we had not planned for previously. It handles the output of messages on the bottom panel, below the display of the gameboard. It contains shared pointers to both a *Hero* and a *GameBoard*, and is able to access the information it needs to print to the screen from those classes via getter methods. It then overloads the output operator to be able to print this information to the screen.

Class: *TextDisplay*

This class is responsible of the output of the map onto the terminal. In contrast to the original design, we opted to have this class contain a shared pointer to the current floor, rather than have each *Floor* have a *TextDisplay* pointer. The field *grid* is the ASCII representation of the *floor*. An instance of *TextDisplay* serves as the first layout for the basic floor plan in *grid*, and we then set the *Hero*, *Gold*, *Potion*, and *Monster* objects by overwriting each *Tile* symbol in the appropriate location whenever a *Character* needed to be generated in a location.

Class: *Character*

This class was given x and y coordinates to aid with movement, but otherwise remains unchanged. The following is an entry from our plan of attack:

A *Character* is an entity that may occupy a tile. They will either be a *Creature* or an *Item*, and will have a char symbol to represent themselves on the *TextDisplay*. This will be an abstract base class.

Class: *Creature*

This class was given fields, *currentAtk* and *currentDef* to denote the values used for combat damage calculations with respect to potions used. This made it very easy to apply multiple potions to a *Creature* without having to rely on a design pattern, as we would simply change the values of these new fields, and then reset them to *attack* and *defense* respectively upon entering a new floor. This method also had the added benefit of adding a potential functionality to apply *Potion*s to enemies, if we so chose to include such a feature later on. Beyond this, it remains unchanged from our original design. We have included the paragraph from the plan of attack below:

A *Creature* can be either the player or a non-player character. Each one will have stats for combat, as well as a *speed*, indicating the movement speed per turn of that *Creature*. They will also have *attack()* and *getAttacked()* methods, employing the visitor pattern to handle the different combinations of combat scenarios. A *Creature* may also be holding an item, dropping it on death.

Class: *Hero*

Hero was given a string *action* field that would be given to *Score* in order to print the current turn's action to the screen. This field would be updated based on any actions that take place, including combat, deaths, nearby potions, etc. We have included some details from our plan of attack below:

A *Hero* is the player character. It will be extended by multiple subclasses, where each subclass is a different race. We note that *Hero* itself is an abstract class. For more details, see UML.

Class: *Monster*

This class was given a method that indicates the coordinates it attempts to move to, or if the player is in range, to attack the player. If this coordinate is valid, we may move the *Monster*, but if it attempts to move into a wall or into a tile with another occupying *Monster*, we have the *Monster* skip its turn as it is unable to move. Beyond this, it is unchanged from our original plan, outlined below:

A *Monster* is a non-player character. It may be hostile to the player by default, or become hostile after an event occurs(such as the player attacking it, as is the case with *Merchant*). Anything that is hostile to the player by default is a subclass of *Enemy*, which extends *Monster*. See UML for details on the *Monster*s that are available in-game.

Class: *Item*

An *Item* is an object in the game that can be picked up by a *Creature,* and is one of *Potion* or *Gold.* In the event a non-player character dies while holding an *Item*, the *Item* will be dropped at the location of death.

Class: *Potion*

In lieu of using a hybrid of a factory pattern and observer pattern as was previously planned, we have decided to simply give each subclass of *Potion* a static field to indicate whether it has been discovered or not. This way, since each type of potion may only be discovered once, it was very easy to simply set the field to indicate discovery, and have it work for all *Potions* of that type.

Class: *Gold*

Gold can be one of two concrete classes: *Regular* and *Hoard*. A *Hoard* is associated with a *Dragon*, and has a boolean indicating if it is able to be picked by a *Creature*. All *Gold* objects have a value.

Class: *Dragon*

Dragon was given a shared pointer to a hoard, just as hoard has a shared pointer to a dragon. This was done because we would like a *Dragon* to attack the player when he/she is within one tile of the *Hoard* as well as the *Dragon* itself. However, we found that this would cause a memory leak, as the program was unable to free pointers that pointed to each other. To get around this, we called a function to immediately kill all the monsters in existence at the end of the game (ie when the player dies or when they reach the final staircase). This dereferenced the *Hoard* pointer to the *Dragon*, while not affecting the state of the board otherwise, as each *Monster* simply died rather than being killed by the player.

<u>Resilience to Change:</u>

Addition of new types of *Creatures*, both *Hero* and *Enemy* types, is very easy. To add these, we may simply add new subclasses and implement any virtual methods to account for special abilities that the new types of *Creature* may have. With small modifications to the generation function, we may also create entirely new types of *Monster* objects - perhaps one that only travels along wall *Tiles*, or one that is able to travel between rooms. However, neither of these would require us to make any modifications to our *Creature* or *Monster* classes.

Addition of new types of potions is also very easy; since the *Potion* class contains methods to determine the effects of a specific potion, we simply add a subclass for a new type of potion and simply specify the values we want to modify. Because of our implementation, adding “greater” or “lesser”potions that have heightened effects is also trivial, as is any implementation of a “hybrid” potion, one that modifies more than one stat of the player. It is also possible to add a potion that does not modify the player’s stats at all, but rather affects an enemy or even the tiles on the floor, though this would require a little more code. Regardless, even an addition such as this would simply require us to add a new subclass to *Potion*, nothing more.

In our implementation of *GameBoard*, we have also made it so that we may intake any preloaded map, with *Character* objects placed wherever we wish, and the game will load that map and be playable as normal. It is also possible to load a map with an entirely different terrain, given that the provided text file is a “valid” map, as we have generated the *Floor* by parsing the given text file and creating *Tile* objects to fill the grid as appropriate.

Even adding entirely new assets to the game would not be overly difficult; for instance, adding a weapon and armor system would simply require us to extend the *Item* class, and build the different types of weapons and armor as we desire. Though this would certainly require quite a bit of work to add to the game, it would not require us to modify any existing code, but rather only necessitate the addition of classes that would themselves implement these new functionalities.

<u>Answers to Questions:</u>

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

For overview, see UML.

We will create a *Character* class, with a subclass *Creature*. Class *Hero* will extend *Creature*, which will be extended by multiple different subclasses for the different types of player characters available(shade, goblin, etc). This makes adding new races very easy, as we simply add a new class *RaceName* and make it extend the *Hero* class.

The same applies for enemies; our class *Monster* extends *Creature*, and is extended by class *Enemy*. Any non-player characters that are supposed to be hostile to the player by default will extend *Enemy*, and so adding a new kind of enemy would just be adding a subclass of *Enemy*.

And finally, any non-player characters that are not hostile to the character by default will inherit directly from *Monster*, such as the *Merchant* class.

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

Within the *Floor* class, we have generation functions for all our assets - *Hero, Monster, Gold,* and *Potion*. The *Floor* class itself has a vector of vectors to *Info* (a coordinate, intx and int y) structures, called *rooms.* The first index, *rooms*[0], contains all room tiles on the floor that are available to have a *Character* placed on it. Each of the other indices contains the room tiles corresponding to a specific room; for instance rooms[1] contains all room tiles that are in room 1.

The vector in index 0 is responsible for generating the player character and enemies, and generation works in the same way for both. We simply call the function *shuffle* on *rooms[0],* which randomizes the ordering of the coordinates of the tiles. We then take the *Info* structure at the back of the vector, extract its x and y values, and set the coordinates of either the player or the enemy to those values. In the case that we are generating a *Monster*, we then throw a shared pointer to this enemy into a vector *monsterHolder*, which stores the monsters that are currently alive on the current floor. Afterwards, we call *pop_back()* on *rooms[0]*, thereby removing the coordinate from being used again in order to ensure that we do not generate another entity in the same location as the one we just generated.

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

We use the same technique for both *Hero* and *Monster*. With each combat interaction, we have two functions that run - *attack(&other)* and *getAttacked(&other)*, where *other* is either a *Player* or an *Enemy*, depending on which class the method happens to be in.. The *attack()* function will be responsible for dealing with whatever abilities correspond to the entity doing the attacking; for example, on every attack a vampire will gain 5hp unless *other* is a dwarf. Likewise, we employ the same technique for *getAttacked()*; for instance a halfling will roll a random number, 0 or 1, to determine if a hit was successful. This way, we can account for many different combinations of *Character* objects hitting each other, without worrying about creating separate functions to deal with each individual combination of attacker and defender.

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 weighing the advantages/disadvantages of the two patterns.

Actually, we found that neither pattern was necessary to implement the use of potions. Rather, a *Creature* would have fields *currentAtk* and *currentDef* in addition to the standard *atk* and *def* fields. With a *Hero*, each applied potion would affect one of these fields, or the *hp* field if it was an RH or PH potion. Then in combat, we would use *currentAtk* and *currentDef* to calculate damage, and upon entering a new floor, we reset the values of *currentAtk* and *currentDef* to *atk*

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?

Potion and *Gold* both extend *Item*. As explained above, we generate both of these by shuffling the indices of *Room* tiles. The only difference between the two is that *Potion* objects should be distributed evenly across rooms, while *Gold* should be distributed evenly across tiles. Thus, we simply use different vectors containing *Room* tiles to generate each type of *Item*, but the underlying code is essentially the same - we shuffle the indices, and run the constructor for each *Item* and place it onto the corresponding *Room* tile.

Extra Credit Features:

-Intelligent tracking: Enemies have been given a tracking system. Each enemy will now move in such a way as to minimize the distance to the player on their turn, given that it is in the same room as the player.

Each *Creature* was given a *roomID* field, denoting the current room of that *Creature*. On each turn, if the *roomID* of a *Monster* matches that of the player, that monster will follow the player instead of moving randomly.

The biggest problem with this was figuring out how to update the *roomID* of the player. We achieved this by regenerating the *rooms* vector after generating all our assets for the *Floor*, and then running a function to update the player's *roomID* after each turn if their new coordinates happened to match a coordinate in a *room* vector.

Another problem was dealing with non-rectangular rooms - if an enemy tried to move to the *Tile* closest to the player, but that *Tile* happened to be a wall, then it would simply not move as it would have attempted to move into a wall. To fix this, we made each *Monster* move as close as it could to the player after each turn, but in the case that this move was invalid, to try to move closer in just the x direction, and then also try to move closer in just the y direction.

This DLC is activated using the -t command.

-Achievement system: We have implemented an achievement system, printing out a message to the screen when the player accomplishes any one of the goals specified in our list. What was probably the most difficult thing about this was figuring out how to add this functionality in such a way that it was able to get all the information that was required, but yet not have it check for this information too frequently, as that would cause performance to suffer. Even more so, since most of the achievements were unique, we had to come up with ways to check the conditions of each individual achievement separately.

We completed this feature using three vectors, where each index of a vector represents an achievement. One of these vectors contains a boolean to indicate whether an achievement has been obtained, and the other two contain strings to store the name and description of the achievement. We then perform checks at various points in the game in order to discern if the player has been able to get a particular achievement. We will list the possible achievements below:

-Overdose: Consume 3 potions.
-Hoarder: Collect 10 gold pieces.
-Bully: Kill your first halfling.
-Reckless: Kill a dwarf as a vampire.
-Dragon Slayer: Kill your first dragon.
-Racist: Win a game having attacked exactly one type of enemy.
-Clearing House: Kill all enemies in a floor.

-Pacifist: Do not engage in combat in a floor.
-Immortal: Take no damage in a floor.
-Speedrun: Win the game in under 200 moves.
This DLC is activated using the -a command.

-Challenge mode: Every enemy gains a permanent buff to each of their attack, defense and hp stats. In addition, we may spawn more enemies per floor.

This was implemented very easily as the constructor for each floor was able to take in values to indicate the number of *Monster, Gold* and *Potion* objects. To implement the boosting of stats, we simply added a field in the *Monster* class that multiplied each of the desired stats in the constructor. This DLC also has the added bonus of being able to play the game on easy mode, as both the multiplier and the number of generated assets are fully modifiable.

The DLC is activated using the -c command for challenge mode, and -e for easy mode. If multiple arguments are provided, the game takes the first one.

<u>Semi-bonus feature:</u>

We are calling this a semi-bonus feature, as it was included in the base game to resolve an ambiguity that occurs when a *Monster* steps on a *Tile* containing a *Potion*. As this interaction was not specified in the assignment, we have decided to deal with this case in the following way:

-Inventory system: Both enemies and the player can now pick up a potion by walking over it. If an enemy picks up a potion, it will drop it on the ground upon death. If the player picks up a potion, he/she may use it at any time.

This was solved by adding a shared pointer field to each *Creature*, and filling it with a *Potion* if that *Creature* walked onto a tile already containing a potion. Upon death, we obtain the coordinates of a *Monster* and set the coordinates of the *Potion* equal to these values, and emplace the *Potion* back into the *potionHolder*.
If a player attempts to use a *Potion*, we call a function that takes an x and y coordinate, set to -1, -1 by default. If we use these default coordinates, the player uses the currently held potion, if it exists. Otherwise, we will use the potion at the specified coordinates.

An oversight that we did not handle in this bonus feature was what would happen when a *Character* attempted to pick up a potion while already holding one; in this case, we decided to simply overwrite the old potion for simplicity. Given more time, however, we would have liked to have the *Creature* simply pass over the second *Potion*. It is also worth noting that if we so desired, we could have used this same technique to have enemies pick up gold piles as well, and drop any gold upon death - which would not have the same “overwrite” issue, as we could simply add the value of the second gold pile to the *Creature*’s total gold.

<u>Final Questions:</u>

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?

Git is hard. Even after we had become relatively comfortable with it, we still had to ensure coordination and push/pull files as we wrote them so that other members could test their own code. Compartmentalization was important as well; we needed to minimize the need to communicate all of our implementation details to each other in order to save time - though communication in itself also played a significant role in the project.

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

We were trying to force ourselves to use design patterns in cases where they did not really yield any tangible benefit, just for the sake of using a design pattern. This ended up wasting quite a bit of our time, which could have been used towards bonus features for the project. In hindsight, it would have been better to only use design patterns in instances where they would simplify the code, or make it more efficient, or make it easier to modify in the future, and not just throw them in for the sake of using them.