CC3K Final Design Document

Haoren Wang
UW
haoren.wang@uwaterloo.ca

Jishan Luo
UW
j89luo@uwaterloo.ca

Ruiyun Chao
UW
r5chao@uwaterloo.ca

1 Introduction

CC3K is a captivating rogue-like dungeon pixel game developed in C++, which brings players into an immersive world filled with challenging adventures. Using ASCII graphics to display the game world, the turn-based gameplay features procedurally generated levels created using random algorithms. The game is designed to provide a challenging and immersive gaming experience with deep strategic gameplay and a high level of difficulty.

Players first need to choose a race they like as a way to start their fantasy journey. In a total of five levels of dungeons, each level contains five chambers of varying sizes filled with different races of enemies with unique racial skills. Players need to defeat as many enemies as possible and loot to get a higher score. Powerful dragon bosses await challenges, and randomly generated potions within the dungeon enhance the player's attributes, although some may reduce them. Merchants selling items to help players better fight monsters are also present in the game.

The CC3k project aimed to create a simplified version of the classic game Rogue using C++ programming concepts and libraries. To achieve these goals, the project was broken down into smaller components, such as player movement and attack, floor generation, and enemy AI and attack. Object-oriented programming principles were employed to create classes for game entities like the player character, enemies, and cells.

This report will discuss the challenges faced during the development process, the key features of the game, and the lessons learned throughout the project. Overall, the CC3k project was a great opportunity to apply programming concepts to game development and learn more about both fields, resulting in a unique and engaging experience for players in this varied dark dungeon.

2 Overview

Game: The game class is used as a controller class for clients to call commands to read the map, start the game, refresh the enemies, refresh the screen, and other basic functions.

Character: character is treated as an abstract class for all characters, which contains all monsters and players. In this class, any point of the map may generate character class, and it also records the parameters of any character such as blood, attack power, defense power, and basic action commands such as attack and move.

Player: The player class is a class that defines and records the player's actions by calling functions to move, attack, and drink potions or whenever the player makes an action. It is also used for keystroke detection.

Floor: The floor class is a class that represents the different chambers in each level. different chambers are distinguished by their different floorNo. The class also records the exact dimensions of each room, as well as information about each pixel in the chamber, and the walls outside the chamber that block the player and monsters. By calling the functions of the floor class, it is also possible to generate monsters, generate players to random locations and implement random monster movement judgments.

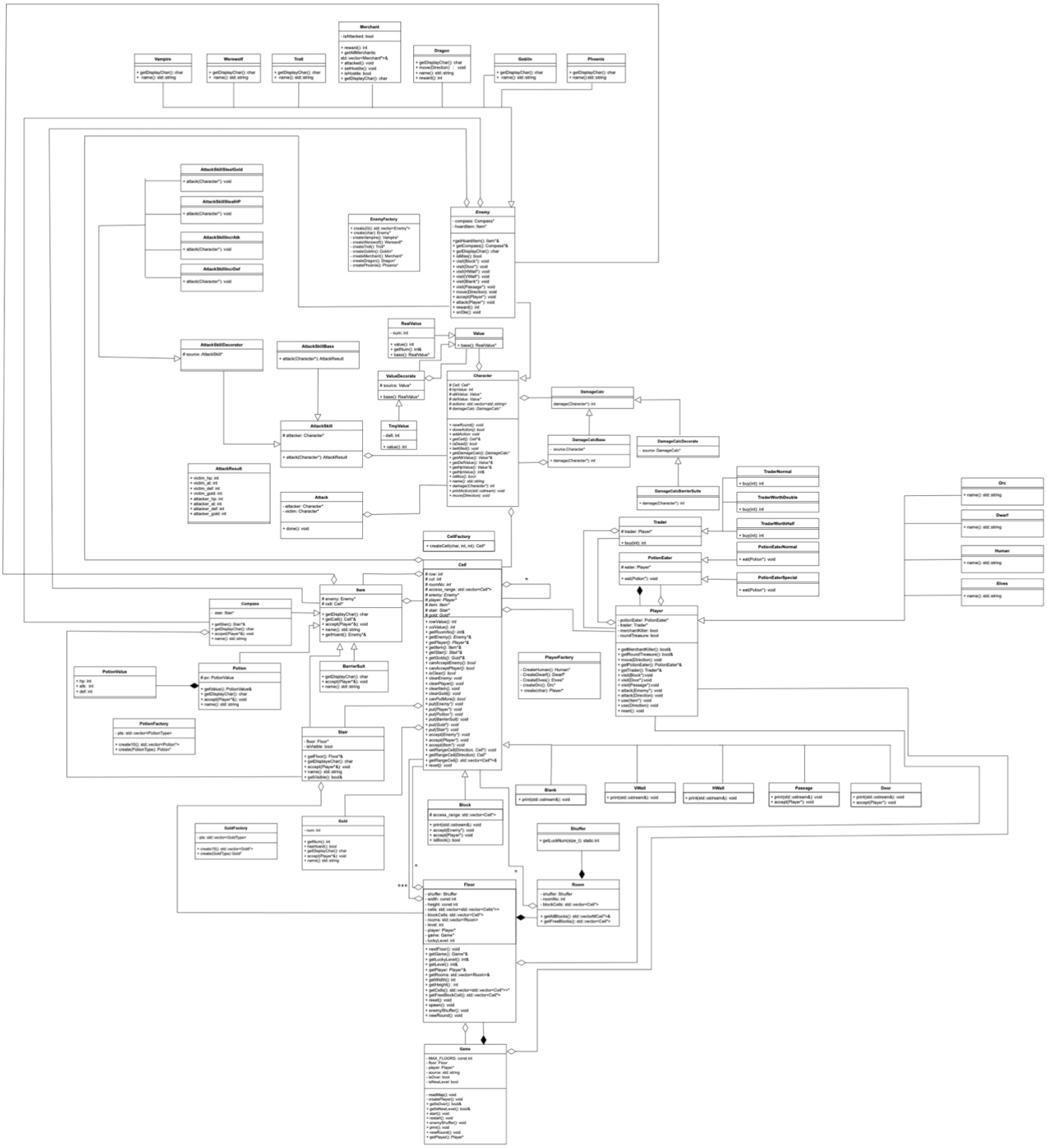
EnemyFactory: EnemyFactory is an abstract class, through the use of factory design pattern, so that the client through the command of this class can directly create a different race of enemies, when we need to add a new race of enemies only need to add the generation function of the race in this class can be achieved.

Enemy: Enemy class is an abstract class that inherits from the character class to interact with obstacles by using the visitor design pattern, and there are different races of enemy classes that inherit from this class. There are many types of concrete enemy classes, including Vampire class, Werewolf class, Troll class, etc. These classes record the names of the enemies and their corresponding symbols in the display.

Cell: Cell class represents any pixel point in the display, which can be treated as a blank, a wall, a channel, or any character. Cell class also records the horizontal and vertical coordinates of the current pixel point, and the range that can be explored by that pixel point. Also, by using the factory design pattern, it is generated by the cellFactory class so that it does not need to be created repeatedly when generating the map.

Item: Item class is an abstract class for all items, including potions, gold and major items. It records their names, positions, which applies a design pattern to generate effects when the player character visit the item. It uses getHoard() method to generate compositional relationship with specific enemies including dragon and merchant.

3 Updated UML



4 Design

We employ the factory design pattern, visitor design pattern, observer design pattern, and decorator pattern to optimize our code and simplify the introduction of new features or functionality.

- The ***factory design pattern*** is used for creating cells, enemies, and players. As a creational design pattern, its advantages include high flexibility, allowing for the creation of objects without specifying their concrete classes. For instance, the EnemyFactory class is called to generate enemies of various races, making it easy for the client to create new enemies. To add new enemy races, we only need to extend the class with the appropriate generation function. Similarly, the CellFactory class is invoked to create cells, eliminating repetitive code in the game map generation process.

- The ***visitor design pattern*** is employed for interactions between enemies and players, and between players and items. As a behavioral design pattern, the visitor pattern requires two dynamic dispatches to facilitate interaction between distinct class hierarchies. For example, the Visitor method in the Enemy class allows enemies to provide feedback during combat with the player or when encountering obstacles. This pattern separates the Enemy class functionality from the various actions that can be performed on enemy objects, enabling easy extension of enemy capabilities without altering their internal structure.

- The ***observer design pattern***, another behavioral design pattern, is implemented to monitor the eight cells surrounding the player character, gathering information about nearby enemies and potions. Each time the player character updates, it attaches its observers, subscribing to its adjacent cells. When nearby cells update, the player character receives a notification.

- Lastly, we use the ***decorator design pattern*** to generate different potion effects, specific attacking abilities, and player abilities. For instance, we define classes for abilities and derive specific attack ability classes for distinct enemies. The decorator class then selects the appropriate ability to use based on the interacting characters.

In conclusion, the factory, visitor, observer, and decorator design patterns contribute to a more efficient and manageable codebase. These patterns facilitate easier modification and expansion of game features, leading to a flexible and scalable game architecture.

5 Resilience to Change

In order to make better use of our code and to make the commands clear and concise on the client side, we designed the game class following the principles of encapsulation and cohesion. By applying the Command design pattern, we created a central controller for the game, allowing the client to construct the game directly through a set of high-level APIs. In the game class, we provide the following APIs through encapsulation: 1.getIsOver(), 2.getIsNewLevel(), 3.start(), 4.restart(), 5.enemyAutoAct(), 6.print(), 7.newRound(), 8.getPlayer(). Each API serves a specific purpose, ensuring high cohesion within the class and making it easier for clients to understand and use the provided functionalities.

In terms of scalability, our design ensures that the memory burden remains constant, even when the number of enemies increases. When the map is initialized, the class of each pixel is generated in advance, and as the number of enemies grows, no additional memory burden is incurred. This approach guarantees that the game can handle large maps and numerous enemies without affecting performance.

For the user interface and user experience (UI/UX), we have designed an intuitive and engaging interface that guides players through the game. The interface provides clear instructions for each command, ensuring that players can easily understand how to perform the corresponding actions. For instance, at the beginning of the game, players receive a welcome message along with instructions on selecting their race. Similarly, when a player reaches the fifth level or dies, they are presented with an appropriate scene, informing them of the outcome.

Error handling and recovery play a crucial role in maintaining a smooth gaming experience. We use conditional statements to validate player commands and ignore any invalid input, ensuring that the game continues to run seamlessly. In situations where a player encounters a wall and attempts to move in that direction, the player is blocked, while other enemies continue to move, allowing for an immediate update.

To facilitate the creation of enemies and players, we utilized the Factory design pattern by implementing the enemyFactory and PlayerFactory classes. These factories not only simplify the instantiation process but also promote low coupling and high cohesion by isolating the creation logic from the rest of the code. If we need to add a new enemy race or player class, we can easily do so by extending the respective factory without affecting existing classes or their relationships.

The player and enemy classes both demonstrate the principles of encapsulation and inheritance. The player class serves as the base class, with each race implemented as a derived class. Similarly, we created the cell class as the base class for blanks and walls in the game map, achieving polymorphism and promoting code reusability.

Encapsulation is further demonstrated in our use of getter methods to access class properties. For example, getName() returns the name of the class, providing read-only access to the class fields and ensuring that they cannot be inadvertently modified. This approach enhances the safety and robustness of our code in practice.

In the potion management, we again applied the Factory design pattern by implementing the potionFactory class. This factory is responsible for creating potions

and placing them randomly on the map, promoting low coupling and high cohesion within the system.

To handle interactions between players and enemies, we adopted the Visitor design pattern through the accept() method. This design choice allows us to decouple the attacking logic from the specific classes, making it easier to modify or extend the behavior without affecting the existing class structure.

Finally, we created the shuffler class to generate random numbers within specified ranges or probabilities, further promoting encapsulation and code reusability.

In summary, our overall design follows the principles of encapsulation, cohesion, and coupling, and incorporates several design patterns to improve code maintainability, flexibility, and extensibility. The map is generated by the floor class, which contains five room classes, each composed of multiple cell classes. By adhering to these principles and utilizing design patterns, we have created a robust and adaptable game system.

6 Answers to Questions

Question 1: How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

By applying the factory method design pattern to generate races, adding additional classes becomes easy. This approach provides an interface for object creation, allowing subclasses to determine which object to create without specifying their concrete classes. This simplifies the process of generating new races and adding new classes. To achieve this, we create a base class Character and inherit from it with enemy and player classes, both of which require health, attack power, and other fields. We then use different Race classes to inherit from the enemy and player classes and use enemyFactory class to generate different races of enemies. This means that we don't have to construct new classes each time we want a different race. Instead, we can simply write a new class and inherit from the enemy class, without any additional changes to the previous code.

Question 2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

We maintain the use of the factory method design pattern to generate enemies, benefiting from its high level of flexibility. The enemy constructors are kept private, and even with friend relationships declared in concrete classes, the encapsulation is maintained, preventing clients from accessing the enemy constructor. By encapsulating enemy creation, we can replace or modify subclasses without altering the public interface, as in the Non-virtual Interface Pattern. This reduces object independence and minimizes code duplication. In our implementation, the enemy class inherits from the character class, and different types of enemies inherit from the enemy class. Therefore,

creating different kinds of enemy classes is all that is required to generate various enemies, similar to generating player characters, we created a PlayerFactory class, based on the race player select, we call methods in PlayerFactory class to create corresponding race of player.

Question 3: How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

To implement special abilities for different enemies, we can use the visitor design pattern, which is a behavior design pattern that applies double polymorphism as its coremechanism. This pattern enables two class hierarchies to interact with each other, allowing us to manage different enemy types and their specific abilities.
First, we create a base class called Character. From this base class, we derive an Enemy class. We then create subclasses for different enemy races that inherit from the Enemy class. In the base Character class, we have an attack() method.

We introduce an abstract class called Attack and include it in the derived abstract class Enemy. Furthermore, there is a concrete class named AttackConcrete that inherits from the Attack abstract class. This concrete class contains an accept() method. Each enemy race subclass overrides this accept method according to their specific characteristics.
In the Enemy class, we apply the method accept(), which determines the concrete type of the enemy and, consequently, their special abilities. For the Attack class, we implement a visit() method that uses function overloading to handle interactions with different concrete types from the enemy class hierarchy.

To implement the attack mechanism based on the visitor pattern, we create a visitor interface or abstract class that defines an accept() method for each character class. Each visitor class then implements the accept() method for a specific character class and performs the corresponding attack. By using this approach, we can achieve polymorphism in the attack behaviors of various enemy types.

Depending on the enemy race, we implement different attack methods. For instance, when dealing with a vampire, we call the accept() method of the Vampire class, which in turn activates the appropriate visitEnemy method from the AttackConcrete class.
In summary, by utilizing the visitor design pattern, we can effectively manage interactions between different enemy types and their special abilities. This pattern allows us to create a flexible and extensible system that can handle a variety of enemy races, making it easier to add new enemies and abilities in the future, thus enhancing the overall gaming experience.

Question 4: What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

Initially, our intention was to employ the decorator design pattern to implement the

temporary effects of potions on a player's attributes. However, during the development process, we discovered that removing decorators to reset the player's attributes at each level was highly inefficient. Since players who are resurrected at each level require resetting their attack and defense attributes, we opted for an alternative approach.
Instead of using the decorator pattern, we directly access the player's pointer within the Floor class and reset the player's attributes. This method effectively clears all temporary potion effects and eliminates the need to create additional decorator classes or construct temporary potion classes. Consequently, our code becomes more concise, readable, and easier to maintain.
By foregoing the decorator design pattern in favor of directly resetting attributes, we can achieve the desired functionality without introducing unnecessary complexity. This approach streamlines the game's codebase, making it simpler to manage and understand while still delivering the intended gameplay experience.

Question 5: How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?

We can minimize code duplication for different item types by creating an ItemFactory class to generate various items and utilizing inheritance to define common properties and behaviors. First, we construct a base class Item containing common properties such as name, value, and position. Then, we create derived classes for specific item categories, such as Treasure, Potion, and MajorItem.

For each item category, we further create subclasses to handle unique properties and behaviors. To generate items like potions and treasures, we define a method within the base Item class that allows us to write generic code operating on any item type. Exclusive properties are managed by the derived classes.

To reuse code protecting both dragon hoards and the Barrier Suit, we create an abstract class ProtectedObject, derived from the Treasure class. Inside this class, we define a pure virtual method protect() to specify each object's protection mechanism. We then create two subclasses, DragonHoard and BarrierSuit, implementing the protect() method separately. This approach enables us to reuse a common interface or base class for both object types while allowing customization of each object's protection mechanism.

Lastly, we implement an ItemFactory class that takes parameters to determine the item type to generate and leverages the inheritance structure to create instances of the appropriate item subclasses. The factory generates various items based on random functions, enabling the generation of different item types using only the ItemFactory class and reducing code duplication.

Using this design, the process of generating items becomes streamlined and efficient. When creating new item types or expanding the current item hierarchy, the changes are isolated to specific classes, adhering to the Open/Closed principle. The

ItemFactory class simplifies the item generation process by taking care of creating the appropriate item instances based on the given parameters.

Additionally, the separation of concerns in the class hierarchy ensures that each class is responsible for its properties and behaviors. This design facilitates code maintainability and extensibility, making it easier to add new features or modify existing ones.

7 Final Questions

Question 1: 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?

The CC3k project provided valuable lessons in both team-based and solo software development. Effective communication, collaboration, and coordination were critical in team settings. Clear communication channels, shared understanding of goals and timelines, and openness to feedback and suggestions ensured that each team member's skills and strengths were utilized effectively. Mastery of tools like GitHub facilitated timely code synchronization, enabling efficient modification and coordination of each component. A designated leader played a crucial role in making informed decisions to prevent slow development progress. Rigorous code checking and testing at each step ensured that errors were identified and resolved promptly.

When writing large programs alone, it is important to establish the program's theme framework and create UML diagrams and daily flowcharts for efficient code completion. Modular code development enhances reusability and maintainability. Consistent naming conventions for classes and variables prevent reference errors, while using tools like Git to track code history increases version stability. Clear comments facilitate better understanding during subsequent development, and breaking down tasks allows for more effective tracking of progress and updates.

Overall, the CC3k project was a valuable learning experience that underscored the importance of communication, planning, and organization in both team-based and solo software development. These lessons will be beneficial in future software development projects.

Question 2: What would you have done differently if you had the chance to start over?

Reflecting on the CC3k project, there are several aspects that could have been improved to create a more streamlined development process and a better final product. One area of improvement is planning ahead, which involves taking more time to design the program's architecture before writing code to manage complexity and ensure maintainability and extensibility.

Another area to improve is testing, where early and frequent testing, as well as incorporating automated testing tools, would help catch bugs more efficiently.

Furthermore, seeking feedback from team members, mentors, or users is essential for identifying areas of improvement and enhancing the program's overall quality.

Writing clear and concise code by following coding conventions and using comments and documentation is important for readability and maintainability. Additionally, using version control software, such as Git, facilitates managing code changes and collaboration.

If given the chance to restart the project, determining a final leader would be crucial to avoid delays in development progress and ensure a unified solution. This would prevent components from malfunctioning when working together. Moreover, improving schedule planning would help maintain consistent development progress and avoid delays in dependent classes.

Lastly, addressing document questions requires summarizing each team member's perspective to comprehensively answer the question, even though combining these viewpoints can be challenging. By reflecting on these areas and incorporating these lessons into future projects, a more streamlined development process and a better final product can be achieved.

8 Conclusion

In conclusion, the CC3k project provided us with a valuable opportunity to develop technical and interpersonal skills in the context of team-based software development. Through this project, we learned the importance of effective communication, collaboration, and coordination in achieving project goals.

We also gained an appreciation for breaking down complex problems into smaller, manageable components, planning and designing the program's architecture, and testing the code early and often. Furthermore, we recognized the importance of writing clear and concise comments, documenting code, and organizing code into modules or libraries to promote maintainability and readability.

The CC3k project also challenged us to think critically and creatively in solving problems, as well as to adapt and pivot when necessary to meet project requirements. We were able to leverage each other's strengths and skills in order to produce a better end product, and we learned valuable lessons in teamwork and collaboration.

While there were areas for improvement in the CC3k project, such as better planning, more comprehensive testing, and incorporating feedback from users, we believe that this project provided valuable lessons that can be applied to future projects. We will strive to implement these lessons in our future endeavors, and to continue to grow and develop as software developers and as team members.

Overall, the CC3k project was a challenging and rewarding experience that provided us with opportunities to develop technical skills, work collaboratively with others, and gain practical experience in software development. We are grateful for the chance to work on this project, and we are excited to apply the skills and knowledge we gained to future projects.