Final Report: CC3k Game Design and Development

Project Team

- Shenq Kang Tan (sk4tan)
- Kristofer Gaudel (kgaudel)
- Xin Huey Wong (xhwong)

Overview

Our team utilized 2 design patterns to build our project, which are the MVC pattern and the Singleton. Additionally, we adhered to fundamental OOP concepts, namely Abstraction, Inheritance, Encapsulation and Polymorphism to ensure a well-organized and maintainable codebase. In terms of data storage, we made strategic use of different data structures such as vectors and maps. Vectors were used to store sequential collection of objects, while maps allowed us to store key-value pairs which was essential for efficiently managing entities with unique identifiers (e.g. to track Unit's coordinates on floor).

Updated UML

Please see the last page of this document.

Project Design

1. Model-View-Controller (MVC) Pattern:
Model: Responsible for managing the game state which includes the player stats, floor layout, enemy actions and items.

View: Displays the game's graphics and user interface. It uses the data from the Model and displays it as an updated game board for every turn a player takes. The player is able to view the random enemy movements, their player character's movements and player stats.

Controller: Takes in user input as commands which dictates player action for every turn. The commands enable the player character to move, attack and use any potions they encounter.

2. Singleton Pattern:
The Singleton pattern was employed in our project to manage the Board class, serving as the "Model" in our application's Model-View-Controller (MVC) structure. This decision was driven by the following reasons:

- Uniqueness: In each instance of the game, only one game board is created and used. The Singleton pattern ensures this uniqueness, preventing the instantiation of multiple boards in a single game.

- Global Access: The Singleton pattern allows for global access to the Board instance throughout the application. This is crucial because various elements of the game, such as the Player class or the Floor state, need to refer to or interact with the game board.

- Centralized Control: The Board class is responsible for keeping track of several critical game elements, including the Player class, the floor state, the default layout, and whether we are using a custom layout file. By implementing the Board as a Singleton, we have a centralized control point that manages these various elements consistently and effectively.

Using the Singleton pattern for the Board class thus greatly simplifies the structure of our game, ensuring consistent access to game elements and the enforcement of game rules.

Object-Oriented Concepts used:

1. Abstraction
    a. We created abstract base classes for characters and enemies. This served as a blueprint for various concrete implementations of characters and enemies in the game. We included both pure virtual and virtual functions that defined the common interface and behavior all derived classes will use. We are able to represent shared characteristics and methods without using too much code.
    b. The MVC pattern abstracts the model's data from the view and controller. Having this abstraction allows each component to focus on its specific responsibilities.

2. Inheritance
    We used inheritance in various ways to model different entities and their relationships. This facilitated the sharing of properties and behaviors between classes, and promotes code reusability. We have 2 main base classes:

    a. Unit Hierarchy: All characters and items share common attributes such as their coordinates(row, col), symbol and type; as well as methods like setting their coordinates and getType. So we created a base class for Unit which contains all these shared characteristics and then had the classes Item and character inheriting from base class. Similarly, this was also done for Enemy, Player and Potion classes which inherited from the character class.
    b. Board Hierarchy: This hierarchy was mainly used for generating the game board. We have the derived classes Floor, Display and Command Interpreter which inherits from the base class Board.
3. Polymorphism
   Different player character races, enemy types and potion effects are represented as subclasses of some base class, and this enables uniform treatment of objects through the base class interfaces. Dynamic binding ensures that overridden functions in subclasses are invoked correctly at runtime based on the actual object type. This approach simplifies interactions and behaviors and allows the addition of new types of the same base class without overwriting existing code.
4. Encapsulation
   Encapsulation hides the internal details of classes, ensuring that their data and implementation are hidden and only accessed through defined public interfaces. We declared some member variables as private and have some getter methods in classes Character and Unit. Encapsulation promotes information hiding and abstraction so that changes to internal implementation do not impact the other components that interact with that class.

Resilience to Change

In the development of our game, we have made strategic design decisions that have increased its resilience to change, providing us with the capacity to incorporate additional features or modify existing ones with minimal disruption to the overall system. This was achieved through the application of the following strategies:

- Model-View-Controller (MVC) architecture: We chose to structure our game using the MVC pattern, resulting in a clean separation between game logic, user interface, and data management. This segregation of duties facilitates modifications as they can be isolated to specific components, reducing the risk of unintended side-effects on the rest of the system. For instance, should we wish to augment the game's visual aspect by implementing a different color scheme or symbol set for game units, we can confine these changes to the view component. Similarly, the controller class allows us to modify the user input interface without altering the underlying game

logic, which resides in the Board, Floor, and Character classes. Conversely, we can alter the gameplay itself without impacting the end-user interface, ensuring a consistent user experience throughout changes.

- Polymorphism: We have employed polymorphism in the design of our Characters, Players, and Enemies. With this approach, we can easily introduce new types of Players or Enemies into the game by defining a new child class and implementing its unique behavior. This design choice maintains the clarity and integrity of our code while facilitating the addition of new features.

- Singleton design pattern: Our game's Board implements the Singleton design pattern, ensuring a single instance of the board throughout the application. This approach simplifies the management of game state and streamlines testing procedures, as there is a single point of reference. In addition, the Singleton pattern supports the future scaling of our project, providing a solid and consistent foundation on which we can introduce new features or enhancements.

- Sequential item generation in the Floor class: Our Floor class has methods designed to generate game elements, such as chambers, players, stairs, and so on, in a sequential manner. This strategy allows us to introduce additional steps or modify existing processes as needed.

  Moreover, the methods we use to spawn enemies, potions, and treasures can be easily adapted to different probability distributions or quantities of specific units. For instance, if we wanted to make a certain enemy type more prevalent in the game, we can modify the relevant spawning method to increase the likelihood of this enemy type being generated.

  Additionally, the Floor class maintains a record of Chamber class instances, providing us with the flexibility to introduce new rules or behaviors based on the Chamber construct. For example, we could introduce a rule where certain enemies can only spawn in specific chambers, or permit enemies to traverse between chambers.

These design choices have not only enhanced the resilience of our project to changes but also streamlined our development process. By anticipating potential modifications and

designing our system to accommodate them, we have ensured that our game can continue to evolve and improve in response to new requirements or enhancements. This adaptability is a crucial attribute in software development, where changes are often a constant, and has positioned our game for successful future development.

Cohesion and Coupling Analysis

One of the fundamental principles we adhered to during the development of our project was to ensure high cohesion and low coupling among our modules. This approach greatly simplified our coding process, made our code easier to understand, and facilitated the testing and maintenance of our system.

- **Cohesion:** We meticulously designed each class to have a specific, well-defined role, ensuring a high degree of cohesion within each module. For example, the Player class is solely responsible for encapsulating the actions and properties of the player character. It is not involved in managing the game's overarching logic or rendering, which are the responsibilities of the Board and Controller classes respectively. Similarly, the Floor class concentrates solely on managing the layout and objects within a particular floor level, and does not concern itself with player-specific actions or global game states. This highly cohesive design approach makes each module more comprehensible and self-contained, simplifying the testing process and making it easier to understand the functionality of each module in isolation.

- **Coupling:** Our project design also exemplifies low coupling, which is crucial for the modifiability and understandability of the system. By implementing the Singleton pattern for the Board class, we ensured that other classes could interact with the game board without having knowledge of the specific Board instance. This reduces the dependencies between classes, enhancing modifiability.

  Further, our use of polymorphism for the Player and Enemy classes allows us to work with various player races and enemy types through a common interface. This design choice greatly reduces coupling, allowing us to add new player or enemy types without impacting existing classes.

  In conclusion, our design exhibits both high cohesion within modules and low coupling between modules. These properties are indicative of a well-structured object-oriented system. The design approach has been instrumental in facilitating

maintenance, understanding, and extendibility of our game, making it easier to incorporate new features and modify existing ones with minimal disruption to the rest of the codebase.

Answers to Questions

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

Answer: We initially planned to use the Factory Method pattern to generate different player races. However, we simplified our process to rely on Polymorphism with each race having its own derived class from the Player class. This simplified approach makes adding additional races straightforward, as we only need to create new derived classes for each new race.

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

Answer: We originally intended for a Factory Method pattern to handle the enemy generation, but ultimately decided to rely solely on Polymorphism. The enemy spawning process starts with the derived classes for each enemy. Each enemy has a derived class from the parent Enemy class. This parent class handles the enemy's main activities (e.g., moving, dropping gold). Each floor of the game has its own set of enemies, so we decided to leave it up to the Floor class to generate enemies. Upon loading the game and generating the floors, a spawnEnemies method is called in the Floor's constructor. This method will use pseudorandom numbers to generate the enemies for the floor using the respective enemy classes.

This design decision reduced our implementation time significantly and improved our testing of the enemy generation. We wanted our game to have its core features isolated, building on our idea for using the MVC pattern. Isolating the game's character spawning allowed us to implement game details without changing our entire codebase with each update.

Our game's enemy generation is similar to our player generation, except for the randomness factor. To generate an enemy, we call the constructor of its respective class and pass in the necessary character statistics. Because of our design, the enemies' traits are implemented within the class. Each enemy's state could be monitored easily thanks to our simple design.

3) Question: 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: The various abilities of enemy characters were implemented as overridden methods in each enemy class (similar to the implementation of player character abilities). Most of the enemies' abilities are handled in the overwritten attack method. Such an example is the attack ability of the halfling. In its class, the attack method generates a random number between 1 and 2, giving a 50% chance of negating the attack. Abilities outside of attacks on players (e.g., neutral merchants) are handled in the fields of the class. Using inheritance and polymorphism allows for code reuse and a clear structure of enemy behaviors.

4) Question: 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 discussing the advantages/disadvantages of the two patterns.

Answer: Throughout our development of CC3K, we pondered the pros and cons of each pattern to handle potion generation. In the end, we favored a simpler design focused on polymorphism. We found this approach to be beneficial because we could isolate the behavior of each potion with their class, and leave its state in the game on the game board. This design follows our idea for the MVC, separating the two features with their respective game components.

Pros:

- Potion effects are isolated to the respective classes, making it easy to specify the behavior of each potion without changing other aspects of the codebase.
- The behavior of using the potion (e.g., removing it from the display after use, randomly generating potions for each floor) is isolated to the display and game logic respectively. We could isolate issues specifically to the component of the MVC, simplifying testing and debugging.

Cons:

- Extra code was required to link all of these parts. We had to utilize lots of steps to connect the potion, the board, the unit the potion lies on, and the player when they use the potion.

5) Question: 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 the generation of treasure do not duplicate code?

Answer: We prevented code generation by only overriding the functions of game aspects that require unique abilities. For the potions, the only virtual function we have is for the display, where we have a function that returns the abbreviation for the potion (used for output to the console). In regards to the treasure, the code is short as we handle the amount of gold in the field of the class. We have made efforts to reduce duplication through a system of following an MVC architecture, where each function has only one purpose within each component.

Final Questions

1) Question: 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:
Through this project, we learned valuable lessons in version control, agile methodology, and group communication. We used personal code branches for autonomy and a main branch for integration, helping us spot bugs and track progress. Regular stand-up meetings kept us aligned and on track, and constant communication on Discord ensured clarity of ideas and code coherence.

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

Answer:
We would improve our task management to prevent duplicate work by using a visual tool like a Kanban board. This would help us maintain a clear project overview and prevent task overlap, thus promoting efficiency and coordination. Also, we'd consider creating an early game prototype with basic features, which would offer insights and a better understanding of game mechanics. This approach would balance our upfront planning with iterative development, enhancing our learning and efficiency.

Conclusion

We had a great time working on CC3K. This project solidified our understanding of C++, object-oriented programming, and industry design patterns. It's quite rare to combine academic and work environments, so it was refreshing to apply what we had learned from our work terms in an academic environment. This project proved to be very useful for our coming work terms and personal projects in software development, teaching us how to

discuss our ideas, utilize version control, and implement our code efficiently. Lastly, it was nice to meet fellow students and make new friends, this was arguably the most interesting part of the project. To conclude, it was fun to apply our knowledge, create a working project, and make new friends along the way.

Updated UML for CCK3 (DD2)

Shenq Kang Tan, Kristofer Gaudel, Xin Huey Wong