FINAL DOCUMENTATION

CC3K+

CS246

Avi Patel
Mithil Damani
Dhruv Mathur

Overview

The CC3K+ project represents the pinnacle of our gaming core system, comprised of the following integral components:

- Controller
- Game
- Cell
- Entity

The Controller interacts with a Game instance, executing fundamental operations. The Game class, serving as an amalgamation of Cells and Entities, is designed to facilitate the creation of Entities and manage the game's functionalities. Cells essentially represent coordinate points on the Floor, taking on various forms such as walkable tiles, blank tiles, passage tiles, wall tiles, stairs, or Entities. The Cell class aids in identifying adjacent cells and contributes to the functional features within the Game class.

Entities can be classified as either Items or Characters, and they possess a cell to establish their location within the game. Items encompass Potions, Gold, Barrier Suits, and Compasses, while Characters include both Players (Humans, Dwarves, Elves, or Orcs) and Enemies (Dragons, Werewolves, Trolls, and others). Players have the ability to obtain various Potions, which can provide buffs/debuffs or healing/poison effects, whereas Enemies are programmed to navigate the floor randomly and engage the Player when in close proximity.

Our project's user interface has been meticulously developed utilizing the Model-View-Controller (MVC) pattern. Users engage with the Controller class through input, which subsequently manipulates the Model (Game class) and the View, creating an immersive gaming experience.

UML

A UML diagram of our CC3K+ implementation is provided below -

Design

As shown in the UML diagram, we used the Decorator and MVC patterns.

As described above, the MVC pattern was the key to creating a design that separates the user interface, the main model (the game class) and the display. This really helped us organize our earlier design from DD1 uml by creating a controller class to handle user input and call on game controlling functions, such as start game, restart game and quit game. Our game class handled the floor generation, end results, calls on player move and attack, enemy move and attack, potion use, and the view that handles the output. The controller has the game and is called upon by the main file after the game is started. The startGame() function handles the entire harness and uses other functions within controller and floor to run the game accordingly. This allowed us to create a modular and object oriented design to create efficient and reusable code that is organized and easy to follow.

We also used the decorator pattern to give status effects to players, which come from potions. Using the decorator pattern allowed us to dynamically add these status effects to the player. It is unknown how many potions the player will use. By using the decorator pattern, we can add attack, defense and health modifiers with ease, whenever the player uses a potion. Let us contrast this with what would occur if we implemented it without the decorator pattern. We could have fields such as “has Attack, Defense, Health Potion” and “seen Attack, Defense, Health Potion”, but this even disregards the negative potions. Thus, this method would create many fields and could become confusing. Also, if the player came across multiple potions of the same type, more fields would need to be added and the whole project could become very prone to errors By using the decorator pattern, we eliminate this chance and simplify the implementation process.

We also tried to minimize coupling while maximizing cohesion by using these design patterns and inheritance within our project. For most classes we followed the inheritance structure such that every class only inherits and uses code from its parent, thus minimizing coupling. The Game class brings everything together, it owns all the Entities and the Cells. We maximized our cohesion through the way we designed our classes. All classes inherit the functions they share

amongst each other from a parent class, to allow for the methods within each class to be specific to its purpose and fields. Moreover, this helped us minimize repetition of code.

Resilience to Change

In order to enable the seamless integration of new races for both player characters and enemies without necessitating alterations to the existing code base, we have meticulously architected our program with abstract superclasses for Player and Enemy entities. These superclasses encompass subclasses such as Human, Dwarf, Elf, Troll, and Vampire. Each enemy subclass possesses the capability to override the specialAbility function within the superclass, thereby defining their distinct special abilities.

Moreover, by employing Entity as the overarching superclass for Items and incorporating the virtual useItem method within the Entity class, we are able to generate a diverse array of Item types through the generateEntity function. The specific functionalities of these Items can be further elaborated in their respective subclasses. To accommodate the addition of new items that require protection, we have a needsProtection field that determines whether or not an item requires to be protected by a ‘guardian’ enemy barrierSuit and DragonHoard items.

Any future protected items can be effortlessly incorporated as subclasses of the protectedItem class. This strategic design choice ensures that our program remains adaptable and extensible, allowing for the straightforward addition of new character races and items without disrupting the existing framework. As a result, our project can continually evolve and expand to offer players a rich and varied gaming experience, showcasing a wide assortment of unique characters and items to discover and interact with.

Answers to Questions

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?

To make it so that each race could be easily generated, we decided to implement each race as a subclass of an abstract Player superclass. This is because each race of player essentially has the same fields, such as Health Points(HP), Attack, Defence, etc, albeit varying in terms of base stats. Similarly, since each Enemy race has certain same fields, such as maxHealth, attack and defence, we implement each race as a subclass of an abstract Enemy superclass. However, each Enemy race does have a special attribute/effect which we implement through the specialAbility function that we have in the Enemy class. Further, in the Player class, we have virtual incrementPlayerGold and attack functions that enable us to provide different abilities based on the race of a player. This includes the elven ability to make all negative potions have a positive effect, as well as the Dwarven ability to double all coins. Through this type of a system, adding additional classes would entail creating additional subclasses of Player, with its own overridden special attribute/effect function.

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

Our system is set to handle the generation of different enemies through simple RNG. As stated through the specifications, a total of 20 enemies have an equally likely chance of spawning on any valid floor tile on a specific floor. Since all of our various types of enemies are sub-classed off of a parent Enemy class, through randomization and abiding by the likelihoods of generating specified types of enemies, we would create Enemy pointers and then assign them to a specific type of Enemy. This slightly differs from how we generate our Player, since that is only a one time occurrence at the start of a Game, whereas Enemies should be generated every time a new Floor is generated (and that too with a randomized race, in contrast to the Player character for which the game prompts the user to select their specific race).

How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires?
To implement special abilities for different enemies, we plan on creating a pure virtual specialAbility() function within the abstract Enemy superclass. That way, we would be able to override said function to account for a specific special ability which would be distinct for each and every race. This specialAbility() function would then be called via RNG or upon meeting a set condition without our moveEnemies() logic or a simple playTurn() logic.

What design pattern could you use to model the effects of temporary potions so that you do not need to explicitly track which potions the player character has consumed on any particular floor?
The design pattern we plan to use to model the effects of temporary potions are Decorators. We have implemented a Player decorator in our Potion class which would wrap a given Player with a class that would represent a Potion effect. We use the useItem method in our Potion class to wrap our player with a potion whenever the player wants to use it. If a given potion effect was a temporary effect, it would then simply be popped off at the end of said floor.

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 a treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?
Items can be generated in such a way, so that code is reused by using inheritance. We have an Item base class that has derived classes, Gold(Treasure), Potions and barrierSuit. Gold further has derived classes that are distinguished by the type of hoard. In our generateFloor function in Game class, since an Item is an entity, we create Entity pointers using generateEntity for all items. The Item superclass would then have a virtual useItem() method, which would then simply be overridden by its subclasses and be implemented as they see fit. By adopting inheritance we reduce the amount of code that needs to be used for items. For instance, for treasure it would simply increment players' gold; for potions, it would add a temporary/permanent effect to a player; or for the Barrier Suit it would set a particular attribute for the Player to true (setHasBarrierSuit). The way we would reuse code to protect both dragon

hoards and the barrier suit was similar to our initial plan of attack. Further, I would go about also having a specific ProtectedItem abstract class which would encompass the BarrierSuit and DragonHoard items, as both require a dragon guarding them. That way, if I ever wanted to introduce even more of these ProtectedItems, I wouldn't have to duplicate any more logic and instead, just create a new subclass of ProtectedItem.

Extra Credit Features

The features we added for extra credit are as follows

Special Attack Ability Procedures
We wanted to make the game more interesting and make certain player and enemy races more unique. This would act as an incentive to play other races as well.
We gave a chance for the abilities to activate during the attack procedure. The abilities are as follows:

Vampire : Health Steal (Active Ability)
Troll : Health Regen (Passive Ability)
Phoenix : Desperate Rage (Active Ability)
Goblin : Gold Theft (Active Ability)

Dwarf : Doubles Gold
Orc : Halves Gold

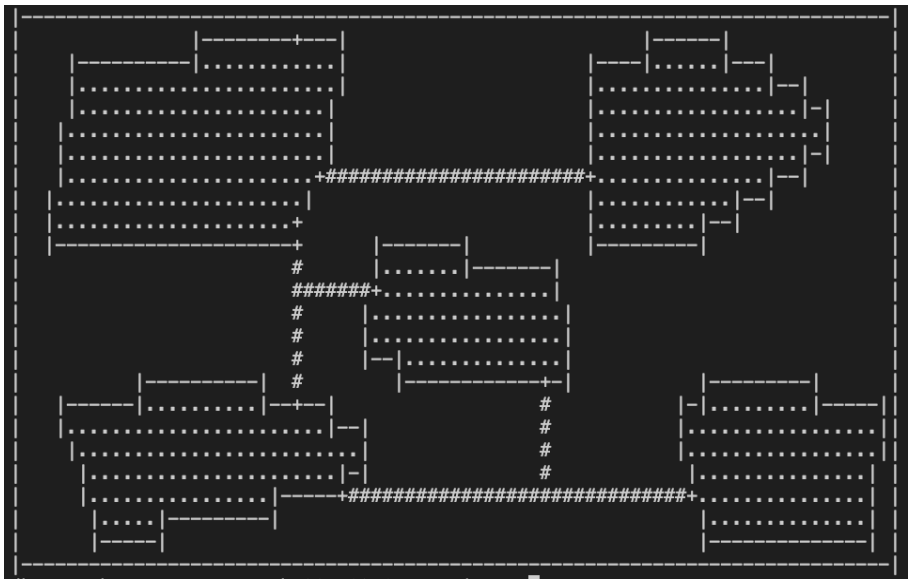
We simply decided to make the SpecialAbility method in Enemy and Player Pure Virtual in order to have each race do something special. We also gave these abilities the opportunity to be randomly activated either when in active contact with the Player or not.

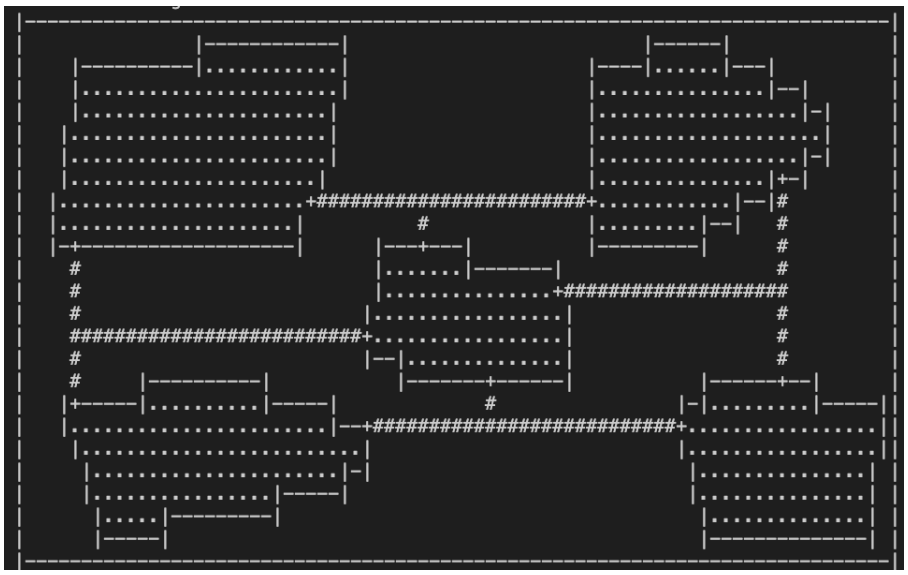
Random Floor Generation
It seemed like a fun idea at first, generating our own floors sounded like a really interesting enhancement, and we personally took interest in the concepts involved in generating these floors. (It was very tedious.)
The fundamental idea behind random generation revolved around constructing chambers and dynamically connecting them through passages. Our original intention was to position five chambers randomly throughout the floor; however, this approach presented several challenges:

- The floor often appeared lopsided, as the majority of chambers were concentrated on one side, resulting in an unbalanced layout.
- The close proximity of the chambers further complicated their interconnection, due to the tight spacing.

To overcome these obstacles, we devised a new strategy: dividing the floor into five approximately equal sections, with each section containing a randomly generated chamber. To create each chamber, we generated 6 to 10 random points within the respective sections, subsequently connecting them to form the chamber's basic outline. We then filled these shapes with floor tiles and enclosed them with wall tiles, successfully generating our random chambers. The final task was to interconnect the chambers using passages. We developed a variety of passage templates as seeds, each with an equal probability of being selected. This approach enabled us to establish seamless connections between the chambers, resulting in a more visually appealing and functional floor design.

Provided below are two possibilities of randomized generation-





Final Questions

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?

Throughout this project, we gained a wealth of knowledge about programming and teamwork. As we were not accustomed to working on such large projects, we had to approach the work with great care and attention to detail. Even small errors became challenging to detect and fix when dealing with multiple files, and some mistakes were critical and required immediate attention. We learned that taking the time to write code carefully and systematically was more effective than rushing through it, as this helped us avoid careless errors and save time in the long run. In addition, we recognized that the planning phase was critical to the success of the project, as it allowed us to see the bigger picture and implement classes more efficiently. We also learned about the importance of maintaining low coupling and high cohesion in our code. To ensure that everyone was working towards the same objectives and avoiding misunderstandings, we engaged in clear communication about tasks, deadlines, and goals. Regular check-ins were essential to ensure that everyone was on the same page, and we also learned to respect each other's work styles and approaches to problem-solving, which helped us work together more effectively.

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

If given another opportunity, we would have approached the development process differently. One change would be to write code at a slower pace. It would also be helpful to begin the project earlier and plan more effectively. Further, we began thinking of the enhancements even before we had the code for the basic game down. Since we spent a considerable amount of time thinking about enhancements, we deprived ourselves of valuable time that we could've given us more time to debug the code. While we often correctly planned the implementation of certain methods, we still made mistakes, such as typing errors or nonsensical lines of code, even though we had the correct logic

in mind. These errors caused us considerable trouble, as it took twice as long to debug them as it did to implement the method. For instance, when we were writing the destructor body for Game, instead of assigning this->FloorPlanSrc, we just used FloorPlanSrc = FloorPlanSrc which ended up giving us an empty string.