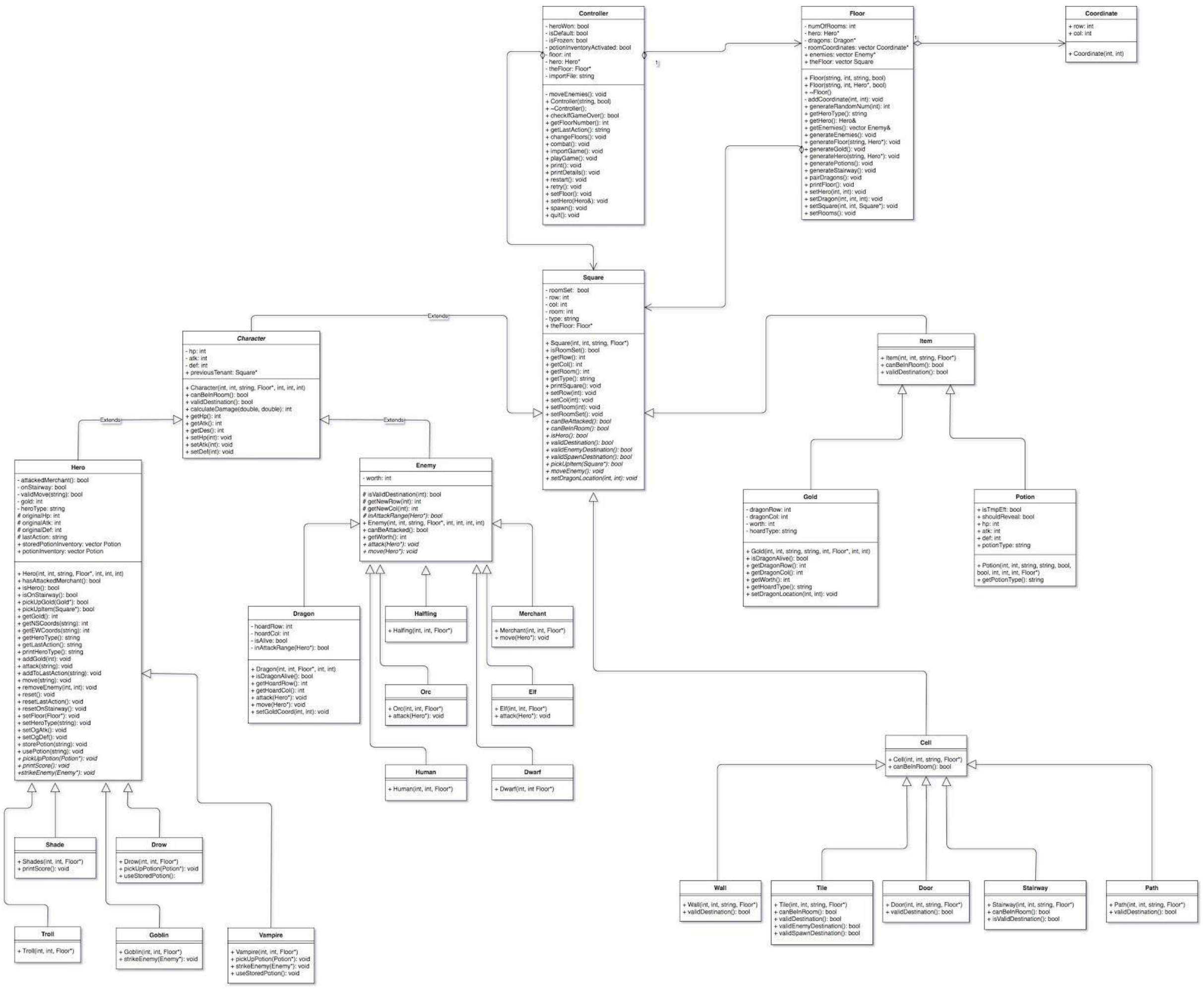
Introduction

CC3K was the ideal challenge for one big nerd and two dedicated programmers, blending a clear connection to object oriented principals and homage to classic RPG's, the group was excited about the project from the start. With a big focus on subclasses, method overloads, and the board state, we feel we've created a strong display of four months' worth of C++ knowledge.

Overview

For our implementation of CC3k, we decided to structure our code primarily around the map, and the existence of objects as a part of each floor. As such each class of available object is a subclass of a square, the generic, position holding, parent class of which each floor is composed. To keep in objected oriented style, we derive each object a subclass of the next, with races as a subclass of heroes, which inherit from the character subclass, which belongs to squares, and so on. This structure simplifies the existence of objects on the board and the generation of objects, and as the subclasses can deal with each other more carefully through overloaded or overridden functions, this approach seemed to hold few limitations. As we progressed, we noted that some occasional confusion was encountered in the writing of the code, as elements could be mistaken for others. Overall, the general design worked smoothly.

Updated UML



Design

There were few problems that we encountered in this project that could not be solved by the versatility of virtual methods. One of the greatest concerns that we had in the initial planning phase was the implementation of special abilities, especially those that acted differently based on the target of an attack. The impulse solution came quickly and simply, store a parameter within the target such that the method can type check and react accordingly. We, however, were reluctant to implement this from the get go, as it violated object oriented design and raised some questions as to the implementation of certain edge cases (not to mention that it seemed likely to lose us some marks). After group discussion and instructional support, we realized that subclass definitions of virtual methods could be overloaded to deal with their own specific edge cases, and thus a branching tree of specialized attack functions could be written without interfering with the basic 'hp -= atk - def' attacks that held true for much of combat.

We also spent time designing the method by which we created a board on any given turn and decided to store information permanently; consequently, printing the board would simply be a matter of looping through the existing vectors of squares. The additional benefit of having a stored state for a board was that affecting the state of the game could only mean accessing information at a given position, and possibly at the positions immediately next to it, and then updating any data that may be stored there accordingly. This implementation was strongly inspired by the observer pattern, though it did not follow the pattern in the strictest sense.

Otherwise, design issues were solved without significant challenge. The team implemented many boolean fields and methods for state changing and checking, created a controller object to store key information and run the game, as well as a playGame() method to improve the legibility of main(). In general, we tried to design our program in such a way that it avoided bad design such as type-checking, code duplication, and excessive coupling. At each stage, manageability was kept in mind and the program was designed to be modular and maintainable.

Resilience to Change

Because of the groups' design and implementation of the heroes and enemies, adding more characters would require derived classes for each additional character and the possibility of certain methods. If the default virtual method provided observes to be inadequate due to character attributes, additionally methods would be needed. Regarding certain integer attributes, the implementation used constants and variables to store these values and therefore these values can be changed with ease. Another change that would be possible is to give the enemies intelligence. At the moment, enemies are randomly moved in a chamber based on probabilities. So, in order to make the enemies pursue the hero would require a calculation to move the enemy along the hypotenuse between itself and the hero. That said, the probabilities given to move the enemy would be discarded to remove the random movement.

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 races?

In our system, we have a derived class, Character, that has two derived classes: Hero and Enemy. The Hero derived class has five derived classes: Shade, Drow, Vampire, Troll, and Goblin. The Enemy derived class has seven derived classes: Human, Dwarf, Elf, Orc, Merchant, Dragon, and Halfling. This design yields a simple implementation of each race generation. Since each race is a derived class of the derived class, Character, these classes inherit the methods from Character. Adding additional races would require implementations of (pure) virtual methods that are in the other race classes. That said, this helps with abstraction so that we do not have to have to check the type of race when using a method. For example, in our strikes method, we alter the Hero/Enemy health points, however, the implementation of these pure virtual methods may be different depending on the combatants.

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

Previously mentioned above, our Character class has two derived classes. The Enemy derived class has seven derived classes: Human, Dwarf, Elf, Orcs, Merchant, Dragon, and Halfling. At the moment, two virtual methods that need to be overridden are our move and strikes methods. The move method will move the enemy accordingly i.e. a dragon will not move whereas other enemies randomly move within their chamber. The strikes method will strike the hero according to specifications i.e. a dwarf causes a vampire to lose 5 HP. In run time, generating enemies and generating the player differ the most significantly, as the PC is generated according to player specifications while the enemies follow a preset but randomized behaviour. Furthermore, accessing enemies and the hero occurs from different pointers, as the hero is stored in its own pointer separate from the enemy array. Aside from this, as both are derived from the same superclass, generation is rather similar.

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

Generation of special abilities was in general, implemented as planned. That most of the abilities did simply get created as overwritten attack methods, where the special effect of the given class altered the original attack method with whatever effect they had, such as a vampire's health gain. Furthermore, as many of these abilities were conditional to the race of the combatant, many times the methods were again overwritten in the case that they were passed a pointer of the given alternate race subtype, such as overwriting a vampire's attack specifically for dwarves. In general, this process was the same for players and enemies, with the obvious exception being the slight difference in the methods for both subtypes of character. The only cases in which implementation of special abilities was not dependent on the attack method, such as the shade's score boost. In these cases, we simply have the default behaviour stored in a method that is overwritten in the race for which the behaviour needs to adapt. Therefore, so long as no special ability behaves wildly different from regular behaviour, new races and abilities are very easy to implement.

The Decorator and Strategy 8 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.

The Strategy 8 pattern would be a wise choice considering we want to define a class, Potions, that will have one or two behaviours - in our case positive or negative effects that are like other behaviours in a list. These behaviours could be permanent effects and temporary effects which could be used dynamically. For example, when the Drow uses a potion, the effects are multiplied by a factor of 1.5. For bonus implementation, we could have this happen to other Heros. Implementing the effects of potions using the Strategy pattern would help with encapsulation and decoupling and reduce a long list of conditionals needed for the different combatants being considered. Further, changes to one class would not need to be implemented in other classes thus saving time. On the other hand, there are five derived classes which would require an increased number of objects/classes. If we decide on the Decorator pattern, modifying an object dynamically would be very easy and manageable. Furthermore, since we have five derived Potion classes using the Decorator pattern would be more flexible than inheritance. The Decorator is also a good choice since we need the capabilities of inheritance with the subclasses, but at the same time might need to add functionality at run-time. Two important factors we also must take into consideration is the fact that the Decorator pattern simplifies our code because the implementations are done using many simple classes so the structure can be copied and pasted. One last important factor is that to include new functionality we won't have to rewrite old code but can extend with new code. Carefully considering the pros and cons of each pattern, we believe that implementing the Potions class with the Decorator pattern will be wiser. This is mostly due to the fact we want to build off the basic functionality of the potions and build new features.

That said, we did not end up implementing either of the design patterns to give functionality to potions in our build of CC3K. Instead, our potions inherently hold the modifier to each stat within themselves and are called upon to change those given stats upon use. These modifiers are provided to the potion during generation. This means that creating new potions of mixed effects, or stronger/weaker effects, would be as simple as modifying the way potions were generated at the beginning of each floor. Our implementation does admittedly have some drawbacks. Firstly, we did limit our potions to only affecting basic stats, and therefore creating something such as double gold potion would requiring either redefining the potions class of including an additional item subtype. Furthermore, as the stat resets are invoked automatically at the beginning of each new floor, modifying the duration of a potion would require some additional implementation, though methods for resetting attack and defense stats to racial bases are readily available. Overall, we feel justified in our implementation of potions as the simplified the code at our present level and allowed for relatively simple extensions with or framework for more complex extensions.

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?

Our implementation for CC3K allows for code reuse in many of the random generation aspects. This is because all objects are derived as subclasses of the basic 'square' object; consequently, when a floor is being generated, our code allows for the highest parental type to be guaranteed to be a square, and therefore any subclass element will be able to be stored within the floor vector. Therefore, the only code that needs to be written per object is that which is unique to the generation of that object, be it the probability of generating an object of the type, or the section of generation that deal with the sorts of parameters passed to the generation function.

In addition, your plan of attack must include a breakdown of the project, indicating what you plan to do first, what will come next, and so on. Include estimated completion dates, and which partner will be responsible for which parts of the project. You should try to stick to your plan, but you will not be graded by the degree to which you stick to it. Your initial plan should be realistic, and you will be expected to explain why you had to deviate from your plan (if you did).

Deviations from the original plan stemmed primarily from group availability. The plan was generated under the assumption that all members of the group would have the same time to contribute and begin at the same time. This, however, proved not to hold in our scenario. Nevertheless, work on the project began early and in an organized manner. Patrick setup a GitHub repository for the project as planned. From this point on, items were tackled in order of necessity. Board generation was taken on by Nicholas, while

Patrick handled item and character generation. Combat became Nicholas's next large project, as Patrick picked smaller interconnected object methods. Vuk, constrained by time and other commitments, contributed primarily in planning and documentation.

Extra Credit Features

Smart pointers can be seen throughout our code mainly in the form of shared pointers. We chose to implement smart pointers because they make memory management a breeze. Since shared pointers do not require the direct use of the keyword delete, overwriting indices of the floor becomes a breeze. The implementation turned out to be 100% effective as we had no memory leaks and no instances of the words new or delete.

Our second bonus feature was a character ability to keep a potion inventory. At the beginning of the game, the user is prompted to choose whether they would like to implement this feature. If the user chooses yes then they now have the following two commands:

1. p <direction>

This is the command "pick up potion" in the direction the user has given. With similar criteria as the command 'u', 'p' stores the potion in the user's inventory which is not reset after each floor.

2. g

This is the command "grab potion from inventory". It allows the user to use the most recently "picked-up" potion. Like the usePotion method, the user does not know the effects of the potion until "grabbing" it. If the user has no potions in their inventory, an informative message is displayed.

This bonus feature is implemented in the hero classes through the useStoredPotion() and storePotion(string direction) methods. These methods are called when the command g and p are read in respectively. The useStoredPotion() is virtual as vampire and drow experience feel the effects of potions differently.

Final Questions

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

Probably the most marketable benefit of completing this project was all the firsthand experience that we received programming with Git repositories. Even with the prevalence of subversion in the industry, basic working knowledge of Git repositories is a great first step in the subject, and the understanding that we gained from code sharing a mutual editing was invaluable to our future experiences in the field.

Furthermore, we also learned a great deal about the more abstract concepts that we'd been taught, such as cohesion, coupling, abstraction, code readability and so on. You encounter many tiny but vital differences when interacting with and programming around others' code. Small changes in naming, spacing and logic conventions can be serious, if temporary setbacks in processing and adding onto the code, but at the same time, recognizing these differences can be very important to your ability as a programmer. Not only does it help with programming language affluence, but many times reading others' code grants insight into your own programming practices and how they can be improved. I think we all had definitive moments interacting with some portion of someone else's code that resonated with us and changed, if only slightly, how we programmed from that point onwards. Overall I think this experience was highly beneficial for us, and almost definitely better prepared us for real work in the industry.

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

Although we implemented two bonus features and get a working executable, if we had started earlier we would have been able to: one, implement more bonus features and two, complete more in-depth testing. In fact, it was the vision of one of the team members to ask his friends to play the game. If this had been the case, these friends would have been an excellent case group for white box testing. Another aspect we would have enjoyed completing more of were bonus features. Without a doubt, all team members learned a lot completing this project, however, the bonus features posed excellent challenges that we would have liked to tackle.

Conclusion

Completing this project was a bittersweet experience. With many hours invested, seeing it finally reach its “final form” is undeniably gratifying. The functionality leaves us with a great sense of accomplishment and pride, and yet the opportunity for expansion doubtlessly leaves a sense of longing for more time as well. Suffice to day, CC3K has left a definite impression on us all.