Omar Elserwi
John Clark
Maanav Rajesh

CC3K
Design

Introduction

In our implementation of CC3K, we learned much about the importance of incorporating good design patterns into object oriented software. A project as large as CC3K forces you to implement readable, well-thought out code. Most of our time was spent on planning how our program's architecture would look and how we can optimize it to reduce code duplication and increase code clarity and simplicity.

Overview

We stayed true to the MVC design and used MVC type solutions for most problems in our program. For example, the aggregation and ownership of objects follow MVC strictly. We have encapsulated the code in such a way that certain layers of the application operate fully within the Model, Controller, or View.

The second priority after a deep investigation of the MVC paradigm was using inheritance and other C++ specific features to our advantage. The standard library provided us with many tools, smart pointers, hashmaps, and various data structures. Using deeply layered inheritance and creating a new derived class for any new type of object allowed us to keep our logic simple and work effectively. As a result of our efforts we spent an extremely minimal amount of time bug fixing and working on previously existing code as we moved along. New features used pre-existing systems.

In conclusion, our project is structured with a high priority on readability and reducing code duplication. We have over 50 files, but it has been organized in a manner such that each file is concise and directed, as a result of inheritance, STL, MVC, and interfaces.

Design

As previously stated, we designed the project from start to finish knowing that we had a very large task in front of us. We believed that through a thorough demonstration of object-oriented techniques, MVC design, and interfaces we could design a scalable project efficiently.

There is an Entity hierarchy from which all enemies, playable characters, items, and treasures derive. There is a Tile hierarchy which derives all cells. The Model, View, and Controller implement the logic for the game.

The MVC and Interface design patterns are used thoroughly. The Model, View, Controller implement the MVC design pattern. The Controller handles user input, the Model transforms the input so that it can change the state of the game. The new state of the game is pushed to the View which renders the game state to the user.

Resilience to Change

Our project was defined with change in mind. Of course, initially we created the architecture so that we could have a simple character moving around. Through the implementation of our software design, new characters can be added with 10 lines or less as a new class. New potion effects, special effects on kill, death, interact, collection, attacking, defend are all easily implementable by simply adding an enum value and overriding Entity functions.

New cell types are easily implementable by adding new Tile subclasses. All classes which are not interfaces except from 'Model' and 'Controller' use inherited functions from interfaces heavily.

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?

The character system can be designed with an extremely high level of code reusability. All CC3K characters share many functionalities; attacking, defending, stats, movement, and they all occupy physical space. By implementing a 'Character' interface, we can create generic behaviours that could be optionally overridden in specific subclasses for these shared functionalities. Adding additional races would be as simple as overriding the unique functionalities for that specific race, such as a new movement pattern or combat mechanic.

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

Specifically, our object-oriented architecture allows us to generate enemies in the same way that we generate the player. After all, in CC3K, the only distinction that needs to be made in a technical sense is that enemies' movement and attack patterns are determined by the computer and the playable character's movement and attack patterns are determined by the user. An 'Entity' interface derives a 'Character' interface which derives all playable and non-playable races.

When it comes to placing the generated 'Character' entities on the board, our game logic keeps track of which character is the user and which are the enemies in a semantic way. Player inputs are applied on a specific character, and the others operate without inputs. In essence, player generation and enemy generation are indistinguishable, except that the user inputs determine the behaviour of the player character.

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

As discussed, we have implemented characters in a very general and abstract way. Technically, we could implement any race as a playable character or enemy without distinction. It follows that player abilities and enemy abilities would be implemented in the same way. As for the method that we will use to implement abilities in general, the 'Character' and 'Entity' interfaces provide generic behaviour that is expected of a race with no special abilities, and a derived race would need to override that default behaviour in the case of a special ability.

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.

Initially, it was decided that the Decorator pattern would be the better choice of the two in order to model the effects of potions. Since it is good practice to have mutators and accessors for member variables as required, we use the accessors of a character's stats always and the accessor is decorated by a potion effect. The 'Character' class is an interface and can represent the abstract subject. The race subclasses represent concrete subjects. An abstract decorator class derives each potion effect as a concrete decorator which decorates a 'Character'. Retrospectively, since we are only decorating a very small part of a character, namely the 'defense' and 'attack' accessors, a Strategy design pattern does not fit, since we do not need to change anything that is happening algorithmically. A decorator adding a simple integer to the accessor suffices.

We have also initially decided that we are not implementing HP potions as decorators. Since these effects persist and only represent a one-time permanent change to the health value of the character, we determined that using a decorator to model HP potions would increase the complexity and size of our logic, which is the opposite of what we would want to accomplish by introducing a design pattern.

After actually starting our implementation, we decided to use neither of these patterns, and ended up resorting to the use of setters and getters to manipulate and retrieve the stats of the player, depending on the potion he picks up. This ended up simplifying our potion implementation, as we found it unnecessary to implement a decorator/strategy pattern, in order to achieve something so simple, that could be easily controlled by the controller class.

How could you generate items so that the generation of Treasure of 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?

Treasure and Potions share similarities semantically, and also in their interactions with the game. In accordance, we have a general ‘Entity’ interface which derives ‘Character’ and ‘Item’ interfaces. The ‘Item’ interface derives Treasure and Potions. This provides much code clarity, since we found that Treasure and Potion as physical objects in the dungeon share functionality with characters. Generation of Treasure and Potion therefore occurs in a similar way that characters do and altogether reduce code duplication.

Extra Credit Features

Our main enhancement is that we successfully completed the entire project, without leaks, and without explicitly managing our own memory. In other words, we handled all memory management via STL containers and smart pointers. This enhancement was not that challenging for us, as we planned on implementing it before we ever started coding, and we all had a good understanding of STL containers and smart pointers. In fact, we found that by implementing this challenge, we made the project easier for us overall, as managing memory was rarely a concern when troubleshooting.

Final Questions

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

This project taught us that communication is a very important factor when it comes to developing software in teams. In order to be able to work effectively together, all team members must understand what is going on with the project, and should agree on the implementation details before starting to actually code it. This ensures that every member has a clear understanding of the tasks required of him, and also ensures that their implementation fits with the overall desired style. Additionally, it ensures that no two members of the team are working on the same task, which would end up being a waste of time for the team overall. This project also taught us that no matter how big the problem is, you can always divide it up into smaller parts, and split it up, which would simplify the overall implementation and also allow additional features to be added with ease.

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

One thing we would do differently if we had the chance to start over would be to manage our time better. Our group completed a lot of work early, way ahead of schedule at first, then our progress slowed down dramatically as most of us had other courses to worry about, and we thought that we could easily finish the project after we catch up on our other courses. This caused us to lose our lead, and be greatly behind our ideal schedule. In the end, we still had enough time to fully complete our project, but we weren't able to implement a GUI and ncursers as planned initially.