CS 246 Assignment 5 - Chamber Crawler 3000

Long You Cai, Yifu Li
ly3cai, y649li

November 30, 2014

Contents

1 Introduction

1.1 Why CC3K?

Out of the three possible game choices for the CS246 final project, CC3K is considered to be the hardest. So why then did the authors individually choose it? Simply for a challenge. Both authors consider themselves to be succeeding in the course and wanted to do more. Exactly how much more, will be detailed in the rest of this document.

1.2 Why the late grouping?

The authors originally planned on each completing their own versions of CC3K and had submitted individual UML and Plan of Attacks by the first deadline. Their coming together and merging of their existing code happened because of over-ambitiousness [1] and the happy circumstance that their prior work was on largely separate (and quite compatible) portions of the program.

Author Cai had finished the input handling, graphics and map parsing while author Li had worked on completing the item, player and enemy race classes.

1.3 Project Overview

The game code is divided up into four main groups of files.

There is the Game class which handles most of the game logic including the main Game loop and player control.

Secondly, there is the Renderable class and its derivatives. These files make up the bulk of the code in the project, with everything from Player and Enemy races, to Potions and other Item types, to the Tiles making up each Floor.

Third is the Display class which handles the rendering of Renderable class objects and the drawing of them to screen. This is also the class which handles tilesets and game messages.

Finally is the Main file. The smallest of the four, due to effective use of object-oriented design, the main() function consists simply of command-line argument handling and the initialization of the Game loop.

2 Game

The Game class is a monolithic class making use of the singleton pattern which encompasses all that one might need while playing the game of CC3K. Its features involve DLC management, Player object tracking, storing and initializing Floors, running the game loop, rendering the UI, and wrapping random number generation. We will discuss a selection of the above, below.

[1] See 6.2-2 for more details.

2.1 Player object tracking

The Game class manages the player by initializing and destroying the its Player pointer member as well as providing an accessor to it in the form of the getPlayer() function.

2.2 Floors

The Floors are either called to be loaded or generated within the Game initialization functions.

2.3 Game loop

The Game class game loop() function is a classic game loop involving 3 steps.

1. Rendering the game state and UI with Game::render() using the Display class
2. Obtaining and acting on user input with Game::getInput().
3. Running enemy AI by calling AbstractEnemy::doTurn() on the current floor's enemy list

By repeating these three steps until an end-of-game condition is reached, we can produce a simulation of the CC3K game.

2.4 rand() function wrapper

The Game class wraps the random number generation functions in the C standard library. Since Game is a singleton class, all functions which require the use of rand() can share a single pseudo-random number generator with a single seed value. This ensures that given an initial seed value, the Game state is fully deterministic.

3 Renderable

While the base Renderable class only contains what is required for an object to be rendered on screen, mainly a sprite and row/column co-ordinates, its derived classes span the majority of the code scope in the entire codebase.

3.1 Character

The Character class has derivatives which consist of the Player and Enemy races. It holds all the virtual Player stat accessors, as well as the movement and movement bounds checking functions.

Character follows the pImpl design pattern, with all the stats of the character being stored in a CharacterImpl class.

3.1.1 Player

The main class of the player avatar, with its derived classes being the individual player races.

The Player and AbstractEnemy-derived enemy classes handle combat through the double dispatch pattern implemented through the visitor pattern. As such, the Player class holds virtual overloaded getHitBy() functions for all the enemy classes.

3.1.2 AbstractPlayerEffect

AbstractPlayerEffect uses the decorator pattern to add effects to Player objects to model the stat changes that Potions and other items may give.

As a derived class of Player, it involves a base Player pointer, as well as stat accessors which can be modified before being piped up the decorator chain.

3.1.3 AbstractEnemy

The main enemy class, with the different enemy types being its derivatives. As with Player, there are overloaded getHitBy() functions for each Player race.

3.2 AbstractItem

This base item class contains the functions necessary to use Potions and GoldPiles, namely pickUp() and getValue().

Through pickUp() function overloads, various AbstractPlayerEffects can be added onto the Player object

4 Display

The display class handles the output of the game. It works by single-buffering all draw() requests to a screenBuffer, before flushing the buffer to the terminal once render() is called.

As this outputs all the lines at once, screen refresh lag is minimized compared to a non-buffered display output model.

5 Main

As stated earlier, main() simply serves to handle command-line arguments and initialize the Game object so that the main game loop can be run.

6 Questions

6.1 Project specification questions

2.1 **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?**

We made an abstract Player class derived from a base Character class with virtual methods and then races were implemented as derived classes of Player. This is similar to how author Cai said would be done in his plan, but the decorator solution mentioned by author Li was scrapped due to it being unnecessarily complex and incompatible with the double dispatch pattern.

2.2a **How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?**

Enemies were not generated by a factory method, but instead using constructors of derived classes of AbstractEnemy, itself also derived from Character. Because of this, enemies and players share the common code in the Character class. The method of generation is about the same.

2.2b **How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.**

The various abilities are implemented through overloads of virtual functions of Character, Player and AbstractEnemy. For the racial interactions, the double dispatch pattern was used.

2.3.1 **What design pattern could you use to model the effect 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?**

The decorator pattern was used to model the effect of temporary potions, as was planned on the plan of attacks. This way, the potions consumed on any particular floor do not need to be explicitly tracked.

2.3.2 **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?**

We have an AbstractItem base class which has virtual and non-virtual functions which both GoldPiles and Potions inherit from, minimizing duplicate code between the two item classes.

6.2 Project guidelines questions

***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?**

Since the authors were in both situations for roughly half the allotted period of time, both questions can be comfortably answered.

When developing software in teams, communication is essential. Having a good code repository (such as Git) and quick methods of contacting each other (including SMS and Google Chat) help, but they are only methods of effectively doing so. When there is good communication between the members of the team, productivity is significantly increased as questions regarding another member's work can be quickly dealt with.

Some lessons learnt about writing large programs include not underestimating a project's scope, setting and following guidelines for style and code in advance, and planning out classes and keeping the plans updated. Much unnecessary anguish was had for lack of the above.

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

Both authors acknowledge that they were overly ambitious in what they believed they could accomplish by themselves in a period of two weeks. Even though both were capable of producing a fairly good effort in the time given, due to tests and assignments from other courses, an excellent program could not have been completed without them working together.

If given the chance to start over, the authors would have worked together from the start to not only save the time lost through the merging of separate (if fairly compatible) code, but also to produce a more coherent final product due to the collabourative brainstorming that working together from the start of a project can allow.