Chamber Crawler 3000 - Plan Document
CS 246 Final Project - Spring 2020
Kaiming Qiu, Allen Lu

Introduction

ChamberCrawler3000 is a unique and exciting Rogue-like single player game. It was implemented using modern object-orientated design frameworks and ideologies, creating a fun and interactive game. Object-orientated design and ideologies were used to create a versatile program, flexible to future modifications. Git has been the chosen version control system to enable a seamless workflow throughout the progression of this project. With respect to the DD1 design document, the final design of the program has deviated little to none (any small changes will be discussed below). Teamwork plus the hard work of both team members, along with the usage of all available tools allowed for the completion of an incredible game.

Overview

This project has been divided into 3 main Sections to be implemented:

1. <u>The "Pieces" of the Game (Players, Enemies, Potions, Gold)</u>

The Player, Enemy, Potion, Gold classes represent their respectively named game Pieces. Both Player and Enemy are abstract parent classes to multiple concrete child classes that implement specific Player and Enemy Races. Player and Enemy also both inherit from the Character abstract class so that a Character object pointer can point to either class in the Board's Walkable Tiles. Players and Enemies also interact directly with each other through the Visitor Design Pattern for the game combat mechanics. All potions inherit from the Potion abstract parent class, allowing for the Strategy Design Pattern's algorithm interface to be exposed through Potion type pointers. All gold types are implemented through the Gold concrete class, with the exception of the DragonHoard type/class, which inherits from Gold and includes a pointer to a Dragon object and the Walkable Tile that it can be found at.

2. <u>The Game Board and Floor</u>

To represent the Game Board (which serves to store current game state/Piece location and perform Character actions), there is the concrete Board class. Whether a Floor plan file is passed as a command-line argument or the default Floor plan file is used, the Client will read in several Floor plans to generate a populated/unpopulated Floors 3-D Vector of Tiles and Walkable Tiles using the Tile Factory classes, following the Factory Design Pattern. If a floor is unpopulated, the Board class randomly generates the game pieces on the current Floor as the player starts the game or enters a new Floor. The Board's Walkable Tile objects contain pointer fields for the Piece objects (any of Character, Potion, Gold) that can reside at that spot. Regular Tiles do not contain any piece pointers as they are non-walkable tiles. The locations of the Player and Enemies are stored as pointers to the Walkable Tile that they reside at. The movements of Enemies and Players are handled by Board functions that swap the pointer to Character objects from the current Walkable Tile to a destination Walkable Tile. Following action validity verification, the consumption of potions and gold is also handled by Board functions that apply their effects then delete the Potion/Gold pointer from the Walkable Tile.

Chamber Crawler 3000 - Plan Document
CS 246 Final Project - Spring 2020
Kaiming Qiu, Allen Lu

3. The User Interface

The User Interface encompasses both input and output functionalities of the game. The interpretation of user input commands to perform actions, such as consuming potions, attacking enemies and movement (referenced to Section 4 of game specifications), is handled in the Main function and results in the execution of the appropriate Board/Character methods. As mentioned in the previous section, the Main function also handles the input of the Floor plans and passes the Floors 3-D vector to the Board. The output of each game state frame is handled by the Text Display class, which is how the player sees the Floor with all the Pieces and the current Game Status. Text Display produces text output by interpreting the contained pointers in each Walkable Tile or resolves to the stored Tile type.

Design

Design patterns were implemented to solve various design challenges in the following areas:

1. Factory Design Pattern: All floor tiles of both Tile or Walkable Tile types are generated from NonWalkableFactory and WalkableFactory classes through the factory design pattern. These tile objects are generated using the text characters in the given floor input files or the default floor layout file of the game.
2. Visitor Design Pattern: The combat mechanic between the Player and Enemies is implemented through the visitor design pattern with virtual, overridden Attack and getAttacked methods for each Character race. This enables precise attack interaction based on the given Player and Enemy races (e.g. applies special effects specific to various combat scenarios).
3. Strategy Design Pattern: The use of Potions by the player is implemented through the strategy design pattern. The Potion virtual algorithm interface method of applyEffects allows for the specific effects of any given potion to be applied to the player during game play.
4. Model-View-Controller Design Pattern: The overall input and output flow for the game follows the MVC Design pattern. User input is handled and interpreted by the Client in the Main function, which is the controller. These interpreted commands then modify the Game Model (the Board) through the Board methods. Finally, the resulting game State/View is displayed back to the user through the TextDisplay class.
5. Inheritance (usage of base and derived classes): This is a crucial part of object-orientated design and the program uses this concept throughout almost all components. Inheritance enables runtime polymorphism with virtual methods which works to both abstract implementation details from clients and reduces redundant code. It also helps to minimize coupling through the different component interfaces which allows future program development flexibility.
6. RAII: Resource Acquisition is Initialization: With a program that involves multiple class hierarchies, it is essential to keep track of used memory to avoid leaks and potential crashes. The RAII idiom ensures that all heap allocated objects have lifetimes that are bound to resources that are cleaned up automatically. It enables this program to seamlessly keep track of memory allocation.

Chamber Crawler 3000 - Plan Document
CS 246 Final Project - Spring 2020
Kaiming Qiu, Allen Lu

Resilience to Change

The design patterns and techniques discussed previously help resolve design and functionality problems (like memory management) and give the program crucial flexibility to respond to any changes in specification. Such flexibility of a program allows for modifications, additions, or even deletions of certain features. The reason for investing significant time during the initial planning phase was to ensure that all features could be added or removed quickly and efficiently around the implemented design patterns, which are significant structural components in the project.

Any program must look to achieve maximum cohesion and minimum coupling. This means that each module should work together towards a common goal but must not be reliant on each other to the point that one small change to one, will lead to a crippling change in the other. Inheritance and class hierarchies are built to minimize coupling. It enables abstract coupling, where the client is only aware of the abstract base class and its interfaces, not the actual implementation of its subclasses. Supplemental features can then be implemented in subclasses that can override existing functionalities without affecting existing features. With such a large program, this enables resilience to a wide range of potential changes.

Many of the project's features were implemented with such resiliency to change in mind.

1. <u>Flexibility to Custom Input Floor Formats</u>

Input floor files are processed through the readFloorPlan function and filter for invalid input such as whitespaces between floors and invalid floor generations (e.g. multiple player locations on the same floor, Dragons without DragonHoards and vice versa). readFloorPlan can also handle both pre generated and non-generated floors, as well as a combination of both from the same input file. Together, these implementation features can handle a large variety of floor input formats and perform validity checking. In addition, any changes to defined behaviour of reading in floor plans can easily be accommodated through modifications to the control flow in the readFloorPlan function.

2. <u>Flexibility to New Tile Types</u>

The implementation of the Tile base class serves as an inheritance interface for potential new Tile Types. This is already the case with the WalkableTile type as it inherits from the parent Tile class and adds augmented functionalities. Since the 3-D vector Floors is a 3-D vector to shared_ptrs of Tiles, any subclasses of Tiles will be easily incorporated into the existing Board.

3. <u>Flexibility to New Character Types and Races</u>

The implementation of the Character/Player/Enemy virtual abstract parent classes allows the game to couple abstractly to all Characters/Players/Enemies. Specific Players and Enemies races are currently implemented through concrete child classes that inherit from Player/Enemy, which also allows for the creation of new player/enemy races by simply overriding the given virtual interfaces. Also, the Character abstract parent class allows for the introduction of new potential character types, such as NPC Sidekicks/Followers and Neutral characters like Healers/Merchants. Since each WalkableTile has an Occupant field for a shared_ptr to a Character, all existing and future Character classes are able to traverse through the board, including the doorways and hallways.

4. <u>Flexibility to New Player/Enemy Combat Mechanics</u>

The visitor design pattern allows for specific race to race combat mechanics to be implemented (such as the existing attack hit/miss probabilities). New or modified mechanics can easily be implemented in these virtual, overridden methods within each of the Player/Enemy child classes. Each concrete class holds the implementations of the Player and Enemy race functions while the abstract base classes (Player and Enemy themselves) abstract away such details and simply offer the public interface to the Client. As a result of this low coupling, any modification can be made in the concrete classes without affecting the public interface, thus maintaining high cohesion.

5. Flexibility to New Potion Types and Effects

The strategy design pattern allows for a single virtual interface to apply different potion effects based on the potion available. Similarly to the Flexibility of New Player/Enemy Combat Mechanics, this allows for low coupling/high cohesion between the actual implementation in the concrete Potion subclasses and the public interface. All existing, new, or modified potions simply inherit from the parent class and follow the given interfaces. Since each WalkableTile has an Potion field for a shared_ptr to a Potion, all existing and future Potion subclasses will integrate seamlessly into the game.

6. Flexibility to Different Input/Output Formats

The MVC design pattern offers significant flexibility to the Input/Output functionalities of the program. Having the View implemented in a TextDisplay class allows for different output classes to inherit from TextDisplay with a virtual drawFloor function that can be overridden to interface with new display formats, such as a graphical user interface format. Having the Model Board implemented with all functions required to play the game exposed in the public interface allows for flexibility with new input sources being implemented in a Decorator design pattern format, such as WASD controls. Whatever the modifications may be, they are isolated from the other MVC program components as they focus solely on their given MVC role (printing out an appropriate interface, reading input, etc).

7. Flexibility to Varying Game Length and Difficulty

With the settings menu and the Board constructor, new or modified settings can be easily introduced through new Board constructor parameters and Board fields that toggle features implemented within Board functions. Current features include setting the RNG seed, toggling enemyTracking, and setting difficulty levels/radius sizes for enemyTracking. Additional features such as allowing Enemies to enter hallways and adjusting Enemy movement/attack probabilities based on difficulty settings were also considered but were unable to be implemented due to time constraints.

Answers to Questions

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 races?

The system has an abstract Player base class that groups together the different race types that a player can choose. These race types are implemented as subclasses to the Player base class which enables any Player pointer in the program to point to a specific race instance representing the player. This provides both ease of generation and makes it easier for future extensions since this design uses the idea of abstract coupling. Any client is only coupled to an abstraction of the Player object without knowing the specific implementations of said abstraction. Therefore, since clients do not know (and will never need to know) the implementation of each subclass (race), adding new races that also inherit from the Player class will integrate seamlessly and will not affect the rest of the program. Moreover, the design's virtual methods that handle attacking, getting attacked, and using potions are overridden in the individual race classes to implement any specific special abilities of the race, which again reduces coupling and minimizes any necessary changes to the entire program when a new race or special ability is added.

Chamber Crawler 3000 - Plan Document
CS 246 Final Project - Spring 2020
Kaiming Qiu, Allen Lu

2. <u>How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?</u>

The generation of enemy races is generated randomly via RNG (a random number modded to the total probability denominator) with probabilities given in the specifications, while the Player race is selected by the user, as outlined in the Project Specifications. However, the location generation of the Enemies and the Player is identical, first with one of the floor chambers being randomly selected and then one unoccupied Walkable Tile within the chamber being randomly selected as the spawn location. This is identical since the requirements of the spawn generation of the Player and the Enemy are identical. The requirement for the Exit Stairs to be in a different room than the Player does not affect Player generation as Player generation occurs prior to Exit Stairs generation.

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

The attack mechanism of every Enemy character is implemented as a visitor pattern to the Player character. The Player being attacked accepts its attacking Enemy and invokes Enemy::attack(Player&) method to begin effect processing. The attack method is virtual and overloaded in each Enemy subclass to handle each of the Player subclass types so that the special Enemy abilities can be applied specifically to whichever Player race they are attacking. The virtual aspect of the attack method also checks at run-time for the type of Enemy that is attacking. The Player's attacking mechanism also utilizes the visitor design pattern, having an identical structure/flow and purpose to the attack/get attacked methods in the Enemy classes. The special ability effects of the Player races are similarly implemented within their appropriate virtual, overridden methods. For example, the Player::usePotion(...) virtual method is used to consume a potion and only the Drow character has an amplified potions effect ability, so this method is overridden in the Drow subclass to provide the appropriate multiplier effect.

4. <u>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/dis- advantages of the two patterns.</u>

The Strategy pattern is more effective than the Decorator pattern in implementing the Potion effect mechanic for several reasons. The Strategy pattern allows Potions to directly modify the Player's stats without the need for additional Decorator objects in memory. Since the order of the potions consumed does not matter, there is no need to preserve the order of application with the Decorator pattern and the effects can simply be applied directly to the player stats through the Strategy pattern. In addition, the potions consumed on each floor do not matter as once the level is completed, every field except the health of the player character will be reset to their class default. Resetting is done faster by assigning new values than sorting and deleting a potentially long chain of Decorators. Overall, this makes implementing Potion interactions with the Player simpler, allowing for few changes if new Potion types/interactions are to be added since they would simply be new Potion sub-classes. In addition, with the Decorator pattern, each time the current state of the Player is needed, the entire pipeline of Decorators would have to be processed to retrieve the information, leading to runtime inefficiency, especially if the Player consumes many potions. One disadvantage to the Strategy method is that it directly changes the state of the Player each time a potion is used. With a Decorator pattern, the Player is left unchanged and the effects of each potion are stacked on top of each other. This may offer more flexibility to adding new items in the game that have different sequential or order-specific effects on the Player, in addition to the existing Potion effects.

Chamber Crawler 3000 - Plan Document
CS 246 Final Project - Spring 2020
Kaiming Qiu, Allen Lu

5. 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?

The generation of Treasure and Potions was planned to reuse as much code as possible by the creation of a template generateItem function which would take a std::vector of Item Factory objects (for all Treasure Potions subclasses). The generation probabilities would correspond to the number of each Factory object in the vector and random generation would simply be a randomly generated number modded to the size of the vector, then calling the spawn() method of the Factory object at the given vector index. The generated object pointer would then be placed in a randomly generated chamber tile. However, this implementation was unable to be completed for the final deadline (the feature was still implemented but in a relatively code inefficient manner).

Extra Credit Features

1. Use of Smart Pointers & RAII

All memory management used in this project strictly used smart pointers (shared pointers) and RAII principles to accomplish the outlined specifications. As a result, the program does not leak any memory in any situation.

2. Intelligent Player Tracking

Enemies have been implemented with Player tracking within a given radius around the enemies. The feature of enemyTracking can be toggled in the settings menu and the enemy tracking radius can also be customized into three difficulty levels in the settings menu by the player. This feature allows for adjustable difficulty levels to suit the skill levels and play styles of different players, especially for those who like a challenge.

3. RNG Seeding

The option to seed the RNG in the game has also been implemented in the settings menu, which allows for reproducible results in game.

Final Questions

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

This collaborative project has provided us insight on several components that contribute to the success of team software development.

1. Importance of detailed planning

The importance of detailed planning prior to implementation cannot be understated as having all functions outlined in pseudocode allowed for quick and painless code implementation. Pre-planning/pseudocoding also allowed us to identify the feasibility of different implementations and decide with the ideal implementation ahead of time, rather than figuring it out as we went.

2. Importance of clear communication and documentation

Related to the first point, the existence of a UML outline document allowed for the team to flexibily implement different components of the program without having to worry about variable or method name/signature inconsistencies throughout. This significantly reduced the debugging time required as most bugs were logic level bugs. Having clear communication between team members (who is assigned to which components to implement and the progress that each team member has made) was also extremely helpful in identifying areas where the workload could be redistributed more evenly.

3. Importance of code reviews

Performing code reviews of all team members' code was also very helpful in keeping all team members on the same page on project progress so that different component interfaces could successfully connect. Code reviews also helped to identify deviations from the pre-planned implementation structure and sparked implementation revision discussions as the code was produced. Finally, reviews also provided extra sets of eyes that caught syntax and logic errors early on in the process.

4. How to use Git and Github

Working collaboratively remotely allowed us to get experiential learning with Git SCM and Github. From the experience gained from this project, we are now able to fluently and independently navigate and contribute to projects on Github (or know how to search for functions that we don't know about).

5. Exposure to different implementation methods/styles

Working collaboratively also exposed us to different implementation methods and styles that are idiomatic to individuals other than ourselves. This helped us gain new perspectives on unfamiliar implementation methods, as well as practicing adjusting to such differences in a working environment.

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

If presented with the opportunity to start this project over, there are a few things that we would have done differently. Obviously, we would aim to have the project completed earlier than we actually did so that we would have time to implement additional extra features to add value to our game, such as WASD input and a GUI output. While the circumstances were unavoidable, we were unable to devote our full bandwidth towards completing this project due to other final examinations competing for our attention. We would have especially focused on implementing the generatePiece template function to reduce repetitive code in the generation of Treasure, Potions, and Enemies, as we were very close to successfully implementing it. Otherwise, we are overall very satisfied with the outcome of our project.