CC3K Design Document

Overview

CC3K is a terminal-based Rogue-like game with a player character who can interact with the various entities in the game such as using potions, attacking enemies, and picking up gold and a select few special items. Each of the various character types, enemy types, and items have their own subclass of their type. For example, humans, orcs, elves, and dwarves all inherit from the Player Class, and vampires, trolls, goblins, etc. inherit from the Enemy class and so on. Each functionality of a character or enemy is implemented as a member function of their class (e.g. usePotion, tryKill (attacking), etc.). The board is represented by a vector of Floors. Floor is a class with a 2D array of Cell objects, and the Cells contain information about their type (wall, door, passage, etc.) and their occupant (whatever is occupying the cell). The Game class acts as the main controller in that it contains the board, the player character, and handles the logic at the highest level, and it is what calls all these methods.

Design

In developing our CC3K roguelike, we encountered several design challenges that required careful consideration. Below, we describe the specific techniques employed to address these challenges.

Initially, we faced the challenge of enabling lower-level classes, such as Enemy, to interact with a higher-level entity, the Player, while ensuring that there was only one instance of the Player created by the Game class. To address this, we attempted to implement the Singleton design pattern. However, our inexperience with this pattern led to substandard outcomes during testing. As a solution, we modified our approach by allowing the Game class to call a Notify method, which passed a reference to the Player. This method allowed enemies access to the Player only when necessary, thereby reducing coupling and enhancing cohesion.

To ensure that enemies surrounding the Player were notified and could attack when the Player was in range, we utilized a modified version of the Observer pattern. This approach enabled efficient communication between the Player and nearby enemies, facilitating timely enemy responses to the Player's movements.

During the development process, we recognized the need for a comprehensive list of enemies to manage their movements effectively. Initially, our design did not have this list, complicating the tracking of the enemy positions. We resolved this issue by generating a list of enemies when the floor was created and passing it to the game logic. This allowed us to update enemy positions systematically each turn, ensuring accurate and dynamic enemy behavior.

Our design included a Protected class structure where dragons guarded their respective gold hoards. This required a bidirectional reference to ensure there were appropriate interactions. For a dragon to attack the Player, both the dragon and gold hoard needed to be notified. The gold hoard required a reference to the dragon to signal when to become hostile, and conversely, the dragon needed to inform the gold hoard of its death, allowing the Player to pick up the hoard only after the dragon was defeated.

For managing potions and other pickable items, we determined that storing them in a linked list would be the most effective approach. This structure allowed us to handle the addition and removal of items seamlessly, ensuring smooth gameplay and efficient memory management. This also allowed us to clear the list of temporary potions when a player traversed through the floor.

For the random generation of the Floor, we encountered challenges with the irregular shaping of some of the chambers and determining which coordinates were valid to place enemies. We solved this problem by storing the dimensions of each chamber (split into rectangles as necessary), and creating a helper function randomFloorTile(Floor* f, int chamber) , that takes in a Floor pointer and a chamber. The function begins by generating a random tile index within the range of available tiles in the specified chamber. It then iterates through the rectangles that make up the chamber to find the rectangle containing the randomly chosen tile index. This involves calculating the number of tiles in each rectangle and adjusting the tile index accordingly until the correct rectangle is identified. Once the correct rectangle is found, the function generates random row and column coordinates within the dimensions of the rectangle. These coordinates are then adjusted to the coordinates of the floor by adding the starting row and column of the rectangle within the chamber. Finally, the function checks if the selected tile is empty. If the tile is occupied, the function increments the column (and row if necessary) until an empty tile is found.

Through these design strategies, we addressed key challenges and created a cohesive and functional game environment. We emphasized reducing coupling, enhancing cohesion and ensuring efficient communication between game entities, resulting in a (hopefully) fun gameplay experience!

Resilience to Change

Our design supports the possibility of various changes to the program specification.

The different player races are implemented as subclass to an abstract `Player` base class, with overrides to the functions `getGoldValue(Gold* gold)` and `usePotion(Potion* potion)` for any unique race features. As such, any new player races can easily be added simply by inheriting from the `Player` base class and adding the race as an option at the start of the game. The new race's specific stats are simply initiated in the constructor for the class. If the new race has any unique features, the existing functions can be overridden, and `Player` could even have new ones added to it.

Our enemies are also implemented as subclasses to an abstract `Enemy` class, and this makes it very easy to add new functionality and new types of enemies. The main function which allows for this is `notify(Entity* player)`, which each type of enemy can override for unique behaviour when attacking the player. Any subclass can receive new special traits like `Dragon` and `Merchant` with only a couple additional lines in `Game` if the trait involves movement or needs to be changed every time the game resets. It would also be very simple to add new subclasses by simply inheriting from the `Enemy` class, and subsequently give these enemies unique features.

Additionally, it would be simple to add new items to the game. All of the items inherit from the abstract base class `Item`, which can be picked up by the `Player` and then filtered from there. Any new items (or existing items which we're adding functionality to) could easily override `notify(Entity* player)` to gain unique behaviour when a `Player` is near the `Item`. Then, in `Player`'s `pickUpItem(Item* item)` we could easily alter the effects of the `Item` on the `Player`.

There is also our `FloorGenerator` class. Currently the floor layout is hardcoded into `Floor`'s constructor and for efficiency we use those dimensions for the chambers in our `FloorGenerator`, but by adding a method to randomly generate those dimensions in `Floor`'s constructor and then give those dimensions to `FloorGenerator`, our existing code for randomly generating enemies and items would work. We can also easily alter that generation depending on which floor is being generated to make the late game more difficult. Plus, it would only require changing one number to change the balance of the game by increasing/decreasing gold, potions, and enemies.

Currently, our game takes player input as strings from the terminal. To enhance user experience and provide more intuitive controls, we can easily implement the `curses` library. This allows us to capture input from the arrow keys, making player movement more natural and responsive. Since our input handling is very modular, this change can be implemented smoothly without extensive modifications to the existing codebase. `Game` contains a method called `movePlayer()` that evaluates the string direction from the terminal input. With the `curses` library, we simply need to review the keystrokes called instead of the string sent into the terminal.

Our adoption of the Model-View-Controller (MVC) architectural pattern enhances the resilience of our design. By separating the game logic (Model) from the user interface (View) and input handling (Controller), we can make changes to one component without affecting the other. For instance, in the model we can update or extend the game logic such as adding new enemy types or modifying player abilities, as described earlier, without impacting the user interface or input handling. Any graphical changes can be handled independently within the View component. This separation can allow us to upgrade from ASCII graphics to a more sophisticated graphical output with minimal changes to the underlying game logic. Input methods can be updated or expanded to include new input devices or methods without requiring changes to the game logic or display mechanisms.

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?

We have designed our system so that each race is a derived class from a base class, called Player. Player contains the virtual methods that would vary between races, such as `getGoldValue()` and `usePotion()`. Player inherits from the `Character` class, which contains all the characters stats (HP, Atk, Def). Each race is then generated by calling that class's constructor with their stats as the arguments. Adding additional classes becomes simple as all that needs to be done is to create a new derived class for the race with overrides of Player's virtual methods, and any additional specifications.

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

Generating different enemies is handled in our `FloorGenerator` class's `generateFloor` method, which handles the random generation of all enemies and items for each floor. When it comes to enemies, we randomly pick which enemy it will be and its location based on enemy spawn rates, whereas our player character is generated before the floors, by the overarching game class. This is because the `Game` class contains a reference to the player character, to facilitate the game logic such as moving the player, attacking, etc. Since the player character has already been instantiated when the floors get generated, generateFloor() takes a Player& as an argument, so that when the player character's location gets randomly generated, it doesn't create a new `Player` object that is different from the one created by `Game`.

How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

Each enemy is its own subclass of the Enemy superclass . To give the enemy types distinct special abilities, we would just create methods of the subclass, or appreciate member variables. E.g. for gold-stealing for goblins, we would give the Goblin subclass a stealGold() method. For instance, in the context of the current game, only Merchants and Dragons have distinct behaviour/special abilities. The Merchant class has a static bool that makes all of them hostile once one instance is attacked by the player. Dragons have a bool which determines if they're currently hostile, do not move randomly, and check the item they're protecting to see if the player is nearby. These methods/fields then would get called by each subclass's notify() method, which gets called by the Game class's notifyCells() method, which notifies the 8 cells around the player once the player moves or attacks

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 treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?

To reuse as much code as possible, we have a function to generate a random number between a lower bound and upper bound called generateRandomNumber. We call this function as necessary to determine a chamber and a floor tile that an item will spawn on, via functions randomFloorTile and randomChamber. We then have functions to generate each item type, such as spawnLoot, spawnEnemy, spawnPotion, etc. which use

randomFloorTile and randomChamber, and the only difference between these functions is the different constructors that get called. For example, spawnPotion calls the Potion constructor whereas spawnEnemy calls the appropriate Enemy constructor. We call these functions the necessary amount of times to generate enough potions, treasure and major items, calling it again to determine gold value or item type.

We've reused the code used to protect both dragon hoards and the Barrier suit by making them children of the same abstract class Protected, which has a Dragon* to their protector. In the generation of the items, when a dragon hoard or the Barrier suit get generated, their position gets added to a vector called protectedPositions such that a Dragon will be generated for each of them.

What design pattern could you use to model the effects 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?

We are using a modified version of the Decorator pattern (in that Potion and Player do not inherit from some base superclass that gets decorated) to model the effects of temporary potions. Every time the player picks up a potion it gets added to the linked list stored in their tempPotion* field. When a player picks up a potion, the effect is immediately applied (the appropriate stat is increased/decreased), and if that potion is a temporary potion, it gets added to their linked list of tempPotions. When the player moves up a floor, the player's removeEffects() method gets called. This method goes through the tempPotions linked list, reverts the effect of each potion (e.g. if it is a BA potion, the player's Atk stat will get decreased by 5), and then deletes it.

Extra Credit Features

The first extra credit feature we added is the ability to buy potions from merchants (all potions cost 2 gold). This is done via the command b ⟨ direction ⟩ ⟨ potionType ⟩: buys a potion of type potionType from the merchant in the specified direction, where potion type is either rh (restore health) ba (boost attack), or bd (boost defense). When this command is entered, the game will check if there is a merchant in the specified direction, and then call Merchant::sellPotion(potionType), which returns a Potion*, and then calls Player::buyPotion(Potion* p), which checks if the player's gold is greater than or equal to 2, and if it is, subtracts 2 from their gold, and then calls Player::usePotion(Potion* p), which applies the effects of p via the method described above in the last question.

Additionally, if the merchants are hostile, every potion sold to the player will be a potion with a negative effect (ph, wa, wd).

A second, similar extra credit feature we've added is health regeneration for vampires. Every time a vampire **successfully** attacks the player character, it regenerates 5 HP. This is done in Vampire::notify( ), which checks if the vampire has attacked, and if that is true it adds 5 to the vampire's current HP (making sure not to go over the max HP allowed for vampires).

Another extra credit feature we've added is the action string, which is a part of the `View` class. This string reflects the actions that turn. If the player attacks an enemy that turn, the string mentions which enemy the player attacked and the amount of health it has left with a unique message for if the enemy is dead. If any enemies attack the player that turn, the type of enemy shows up in the message saying they've attacked the player. When the player moves in a direction, it is shown in the action string. The action string will also say whether the player picked up gold or used a potion, and the corresponding value of the gold or type of potion respectively. The action string also mentions when the barrier suit or compass is picked up, or when the player buys a potion.

A third extra credit feature we've added is potions that affect the player's health over time: a regeneration potion and a poison potion, both of which do +5/-5 respectively to the player's health (with the exception of elves, who have potion effects always positive) per turn for 4 turns. These each have a ⅛ chance of spawning, as do the other potions now.

The final extra credit feature we've added is the (toggleable) ability to use the `curses.h` library to enable WASD controls. This does mean when using it, the user can no longer move, attack, and use potions in the 8 cardinal directions though, instead limited to NSEW. This allows for much faster testing of the game, and overall a more streamlined experience compared to entering strings and pressing enter.

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?

This project taught us several lessons about developing software in teams.

Firstly, we definitely learned about the importance of version control software such as Git. Everyone was assigned a sort of section of code that they were mainly responsible for editing, but even with this precaution we still ran into several merge issues. We sometimes forgot to make sure only changes relevant to the commit were staged, and were only able to resolve issues thanks to Git. Having the ability to revert to previous commits was also very helpful when it came to debugging and adding extra features, since we were always able to return to a working state.

Secondly, this project taught us a lot about the need for planning. We perhaps didn't spend as much time on the UML as we should have, and so had to change a lot of our previous classes as we worked when we realized that our logic was flawed. We added a lot of abstract base classes such as Player and Potion when we realized that it allowed for future additions more easily. We also had to change a few of the design patterns we were planning to use to more closely fit our needs – or even scrapped using them entirely, like the Singleton pattern for Player, since inheritance did not work well with it.

This project also taught us about the importance of communication amongst a team. Since we did a lot of refactoring, as mentioned above, we needed to carefully communicate when interfaces were changing, especially to the person working on Game since that was the controller. We had many discussions in general about design, trying to decide what implementation would be the best for both present and future. The necessity of good communication was also very clear when we integrated all of our sections together for the first time so as to efficiently debug the program.

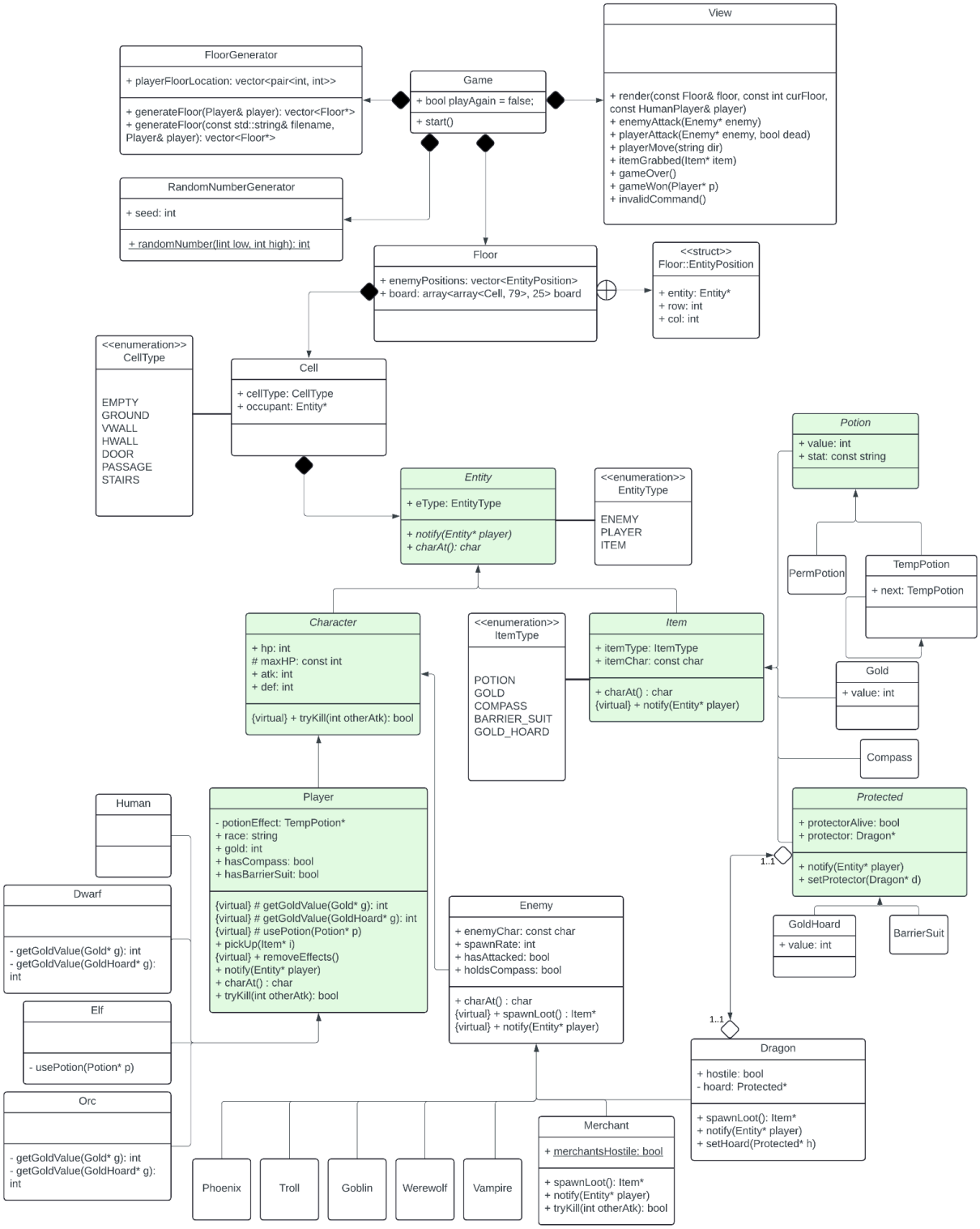
What would you have done differently if you had the chance to start over?

If we had the chance to start over, we would have spent more time on design. That way the actual coding of each class would have gone much more smoothly; any change in implementation would likely not have reached outside of the scope of one person's section of code. With more thought put into our design, we could have properly implemented design patterns instead of attempting to implement them and later realising they weren't viable (as with the Singleton pattern). We likely also could have improved cohesion and reduced coupling much more, for example by giving the responsibility for changing the model to the `Floor` and merely having `Game` call the necessary functions.

Another change we would have made if we were to start this project over would be how we approached testing our project. We originally planned to implement the basic functionality as a prototype, test it, and then add on features as we went. Instead, we ended up basically coding everything at once and not testing or beginning debugging until everything was implemented and integrated. As a result, this was a difficult process.

Luckily, almost all of our code worked right from the beginning, but we likely could have saved ourselves a lot of time had we done unit testing and made a prototype first.

Updated UML


View
+ render(const Floor& floor, const int curFloor, const HumanPlayer& player)
+ enemyAttack(Enemy* enemy)
+ playerAttack(Enemy* enemy, bool dead)
+ playerMove(string dir)
+ itemGrabbed(Item* item)
+ gameOver()
+ gameWon(Player* p)
+ invalidCommand()
FloorGenerator
+ playerFloorLocation: vector<pair<int, int>>
+ generateFloor(Player& player): vector<Floor*>
+ generateFloor(const std::string& filename, Player& player): vector<Floor*>
Game
+ bool playAgain = false;
+ start()
RandomNumberGenerator
+ seed: int
+ randomNumber(int low, int high): int
Floor
+ enemyPositions: vector<EntityPosition>
+ board: array<array<Cell, 79>, 25> board
<<struct>>
Floor::EntityPosition
+ entity: Entity*
+ row: int
+ col: int
<<enumeration>>
CellType
EMPTY
GROUND
VWALL
HWALL
DOOR
PASSAGE
STAIRS
Cell
+ cellType: CellType
+ occupant: Entity*
Entity
+ eType: EntityType
+ notify(Entity* player)
+ charAt(): char
<<enumeration>>
EntityType
ENEMY
PLAYER
ITEM
Potion
+ value: int
+ stat: const string
PermPotion
TempPotion
+ next: TempPotion
Character
+ hp: int
# maxHP: const int
+ atk: int
+ def: int
{virtual} + tryKill(int otherAtk): bool
<<enumeration>>
ItemType
POTION
GOLD
COMPASS
BARRIER_SUIT
GOLD_HOARD
Item
+ itemType: ItemType
+ itemChar: const char
+ charAt() : char
{virtual} + notify(Entity* player)
Gold
+ value: int
Compass
Human
Player
- potionEffect: TempPotion*
+ race: string
+ gold: int
+ hasCompass: bool
+ hasBarrierSuit: bool
{virtual} # getGoldValue(Gold* g): int
{virtual} # getGoldValue(GoldHoard* g): int
{virtual} # usePotion(Potion* p)
+ pickUp(Item* i)
{virtual} + removeEffects()
+ notify(Entity* player)
+ charAt() : char
+ tryKill(int otherAtk): bool
Protected
+ protectorAlive: bool
+ protector: Dragon*
+ notify(Entity* player)
+ setProtector(Dragon* d)
1..1
Dwarf
- getGoldValue(Gold* g): int
- getGoldValue(GoldHoard* g): int
Enemy
+ enemyChar: const char
+ spawnRate: int
+ hasAttacked: bool
+ holdsCompass: bool
+ charAt() : char
{virtual} + spawnLoot() : Item*
{virtual} + notify(Entity* player)
GoldHoard
+ value: int
BarrierSuit
Elf
- usePotion(Potion* p)
1..1
Dragon
+ hostile: bool
- hoard: Protected*
+ spawnLoot(): Item*
+ notify(Entity* player)
+ setHoard(Protected* h)
Orc
- getGoldValue(Gold* g): int
- getGoldValue(GoldHoard* g): int
Merchant
+ merchantsHostile: bool
+ spawnLoot(): Item*
+ notify(Entity* player)
+ tryKill(int otherAtk): bool
Phoenix
Troll
Goblin
Werewolf
Vampire