Overview and Design

Tile IDs

Every object in the game is a Tile with a unique Tile ID. Available Tile IDs are defined in the tileIDs.h. This allows us to a) understand the type of a specific tile at a certain coordinate, and b) allows greater flexibility in our graphical output.

Command Interpreter - Controller

The main responsibility of the Controller is to serve as a mediator between the user and GameModel.

When the game starts, the main() prints out the welcome message and reads the user's input in a loop. First, the user is requested to select a character's race, which is then set using the setPlayerRace(race) method provided by the GameModel.

Players interact with the game through commands. Movement commands call the GameModel::movePlayer(direction) method, potion use calls GameModel::usePotion(direction), and attacks call GameModel::playerAttack(direction). After each successful player's action, GameModel::updateGame() is called to take action for enemies (either move or attack the Player), and notify observers about the change in the state of the game.

Special commands may also alter the game loop's behavior. If 'f' is pressed, the controller triggers GameModel::freezeEnemies(), which freezes all enemies until 'f' is pressed again. Pressing 'r' restarts the game from the beginning by calling GameModel::restart(). Pressing 'q' ends the game with GameModel::endGame(). These commands allow the player to navigate and enjoy the game.

GameModel - Model

The GameModel class is responsible for managing and updating object's data and game's internal state. Its functions can be subdivided to the following:

* gameMap Generation:

The GameModel class uses the initializeMap method to set up the map from a provided file or generate it from a blank one. It calls readMap to set up the map structure and then places the Player, Stair, potions, gold enemies,in random positions if a map file isn't provided. Otherwise, it constructs the map from the provided mapFile.

To group tiles into rooms, we use regex and a flood fill algorithm, starting from an unprocessed tile and recursively marking all connected floor tiles as part of the same room. The function checks adjacent tiles and continues the flood fill process if they are valid (e.g., floor tiles, player characters, items, or enemies). This ensures that we can extract a full room from a file regardless of the dimensions, and also initialize room random generation regardless of the number, size, or shape of the rooms. The corresponding values for processing random room selection are stored in the GameMap.

The GameMap class is central to managing the in-game map's grid and gameObjects on top of it. It maintains a mapping of coordinates to tile vectors, tracks room-to-coordinate mappings, and provides utility functions for interacting with the map, such as adding, removing, and moving tiles.

This is the main deviation from our original design plan. Our original design plan utilized a decorator pattern to decorate every single floor tile, but we realized that it would be too complex and inefficient to save and access each floor tile, not to mention the memory management. Therefore, we opted to replace floor decoration with storing tiles in a separate GameMap object (std::map<std::pair<int, int>, std::shared_ptr<Tile>>) for a faster and more efficient access. This also allowed us to have a neat interface from which to make changes to the map. Furthermore, it allowed us to make the GameModel more cohesive, ensuring it can access objects more easily. We also decided to change a few class names, such as the old “Floor” -> “Tile” for easier readability.

* Spawning and Random Spawn:

To abstract away the process of constructing new Tiles, we used a Factory Design Pattern. More specifically, we implemented PlayerCreator, EnemyCreator, ItemCreator, and CellCreator classes which all inherit from a single abstract Creator class.

The PlayerCreator provides a spawnPlayer method that takes in x and y coordinates and an ID of the Player race. This method implements a switch statement to decode the provided ID, calls the constructor of the desired Player race, and returns a shared pointer to the GameModel. Other creators implement a spawnTile method, which implement the same logic for enemies, items and cells respectively.

Inside the GameModel, we use the spawnObject method to place various game objects on the map. This method distinguishes between different types of objects—player, cells, items, and enemies—by checking the input character type and using corresponding creator classes. Depending on the type of object, the method sets attributes such as room number, player position, and special items like the dragon hoard (when initializing a dragon) We also implement the spawnRandObject method to generate a random enemy/potion/gold at specified coordinates. More on that can be found under 5.2: Enemies.

Both methods ensure that newly created objects are added to the GameMap and corresponding vector array in the GameObject, allowing us to easily move and interact with them during gameplay. For example, updateGame() only calls Tile.takeAction() on the tiles which need to “take an action”. This design abstracts the object creation process using factory patterns and creator classes, ensuring that the GameModel remains modular and easily extensible for future additions and modifications.

*** Handling Movement/Combat between Player/GameObjects**

The movePlayer method handles player movement on the game map. It first calculates the new position based on the specified direction and checks if the tile at the new position is passable using the canMoveHere method. If the player can move to the new position, the method updates the player's position. Any Gold at a new position immediately gets picked up and despawned using the removeTIle method from the GameMap class. If there is a staircase at the new position, the nextFloor method is called which resets objects on the current level and notifies observers to render the change.

The playerAttack method enables the player to attack enemies in a specified direction. It first verifies if the target tile contains an enemy using the isValidAttack method. If an attack is possible, the method performs the following steps: 1) The player attacks the enemy, and the damage dealt is calculated. 2) If the enemy is killed, it is removed from the map, and the player may collect gold. It also handles special cases of spawning gold for specific enemies by checking its type (e.g. humans spawn 2 piles, while merchants spawn a merchantHoard).

The usePotion method allows the player to use a potion in an adjacent tile. It verifies if a potion is present using the canUse method and then applies the potion's effect to the player, removing the potion from the map.

The enemyAction method allows enemies to move and respond to Player’s actions. It first checks if the Player is within the Enemy’s reach, in which case it will perform an attack. Otherwise, it will move in a random available direction, which is handled within a switch statement.

3. Display - View

The Display class completes our MCV model, where the controller is the main function, the model (and subject) is GameModel. The observers (InfoDisplay and TextDisplay classes) monitor state changes of the GameModel. After every valid Player move, the main() calls the updateGame() method provided in the GameModel, which notifies both observers to refresh the image and info. The score is displayed at the end of the game.

Using the MVC model allows us to significantly reduce coupling and increase cohesion of each unit of our code.

Designing for Resilience to Change

When designing the structure of the programs and how we would adapt our program to fit any hypothetical last-minute changes to the program specifications, we adopted a few strategies such as maximizing cohesion, and making various .h files to have “pseudo global” constants.

The main thing we focused on when designing the program was localizing all the values a class would use to describe itself to its own fields. This effectively meant that the class itself was responsible for handling almost all changes, with various “interaction” functions which could serve as a method to modify them. Most notably, this is how we handled combat in this game. Each of Player and Enemy has their own methods to handle their attacking and receiving attacks. In their case, the function that receives attacks would modify its class’ fields, rather than having an attack function in another class do so (although a similar effect could be done with getters/setters, this wouldn’t provide us with as much control over each element of the interaction). This improves resilience to change as it allows us to modify code in specific places where the change takes place while keeping the interface largely the same, thereby not impacting parts not affected by the change. For example, if the “special ability” of an enemy changed, we would identify the part in Enemy where that change would apply (or in certain cases pretty obvious parts outside of Enemy), then simply find another suitable function to put the new ability in (or override a base class function). Another place where we did something similar was in the processing of player abilities, where each player has a function bool triggerAbility() that specifies a race and an “ability number”, thereby allowing us to simply call that function whenever we needed to process within Player, and also get whatever we needed to perform any processing that would come with the ability outside of Player. This is a virtual function within Player, so every player would have one, allowing us to generalize the Player pointer.

Of course, while each subclass implemented their own subclass-specific function overrides, any core logic and fields were implemented in the base classes. For example, the logic for default Enemy actions (moving, attacking, and deciding in between) are implemented in the Enemy abstract base class so that should say the formula for calculating damage change, we would only have to modify it in a few places instead of every subclass. These defaults also allow us to easily implement new Enemies and Players, as they’d simply inherit a base class which already contains default fields and implementation; we’d only have to modify what needs to be modified.

On the topic of overriding functions, another interesting thing we did was include functions in between the main “components” of the core logic. This is simply an implementation detail that doesn’t change the interface, but by overriding these “pre/postprocessing” functions, we save having to completely reimplement the core logic if the modification isn’t very big.

We also made use of default constructor arguments to augment class-specific logic without impacting most of the outside code. Of course, this came with the caveat of having to implement those default arguments on every subclass in the chain up to the specific class, but we made use of this design choice even during development, where we realized we needed some what to determine between gameObject "types" that wasn't just checking the tileID to every single tile of that "type".

Another notable thing we implemented to adapt to change is including various constant files, such as a tileID enum, maps to and from tileIDs and chars/strings for output/input, constants for the 8 cardinal directions, etc. This allows us to not only avoid any hardcoding in obscure places in the code that may need to be tracked down and changed if, say, the char for inputting a specific tile in a map file changed, but also is easily expandable, especially with enums, where inserting an extra tile in the middle wouldn't change the behaviour of the program. The tileID enum specifically was a core part of our program, as it allowed us to easily identify the type of whatever tile we wanted to.

We also made many design choices specifically with the intention of potentially having DLC (although that portion never came to fruition). For example, when designing the base Tile class (which almost everything "physically present" in the game inherits from), we made many classes which wouldn't initially seem to apply to something like a wallTile, such as takeAction() and move(). This is because we recognized that enemies and players weren't the only things that could move, or could require turn processing. What if there was *anything* turn-gated? What if we had items that moved? By having takeAction in the base Tile class, we could call it on whatever we wanted, and by having a default implementation of *do nothing*, we wouldn't need to worry about it if we didn't need it.

Finally, we implemented map file input and the gameMap with the mindset of adaptation in mind. Almost nothing in our map code is hard-coded. The map file input uses a flood fill algorithm to determine the number, sizes, and coordinates of the rooms, so we don't need to hard-code that. We use regex to automatically determine the beginning/end of lines for a specific floor, so we don't need to hard-code a width (for reference, the regex expression we decided on was "\|-*\|", which would identify any line of the form "|——————-|" as a sort of "top/bottom border" of a floor. We implemented the gameMap class to even accept mapFiles that weren't perfect rectangles. In short, our gameMap supports arbitrary shapes with arbitrary rooms, as long as it's within reason and has proper "top/bottom borders", which we believe is quite satisfactory.

Answers to Project Specification Questions

<u>1: Player Character</u>

Question: 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?

Following our original plan, we create a base class for the Player. This class would be abstract and include fields such as HP_Now, HP_Max, Atk, Def, ID, and Gold. Each player race inherits from this base Player class, initializing its specific HP, Atk, Def, and tileID fields.

To abstract away the process of constructing a Player character from the GameModel and increase cohesion, we introduced a PlayerCreator class that inherits from a basic Creator. The PlayerCreator provides a spawnPlayer method that takes in x and y coordinates and an ID of the Player race. This method implements a switch statement to decode the provided ID, calls the constructor of the desired Player race, and returns a shared pointer to the GameModel. For easy access and manipulation, the Player is stored in the appropriate field of the GameModel, which handles any changes to the Player's state.

The above setup allows us to easily add new Player races. To do this, one would: 1) Inherit from the base Player class and initialize its fields in the constructor. 2) Redefine triggerAbility, attackEnemy, and attackedBy. 3) Add the new tileID to the tileIDs enum. 4) Add a clause to the switch statement in PlayerCreator to call the new race's constructor. This design enables minimal coupling, easy race generation, and high cohesion, allowing for straightforward expansion and maintenance.

2: Enemies

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

Changing our original plan, Enemy generation is abstracted away from the GameModel and implemented within the EnemyCreator class, which also inherits from an abstract Creator class. The EnemyCreator provides a spawnTile method that takes in x and y coordinates, an ID of the Enemy type, and a isRandGenerated boolean. This logic mirrors the process of generating a Player, returning a Tile shared pointer to the GameModel, where it is stored inside the game map and enemies vector. This makes turn processing more efficient, as we can simply process the turns for each GameObject that requires it.

Notably, the spawning of Dragons is now handled by the GameModel within the createDragonAndHoardAtRandPosn() method, since constructing a Dragon requires knowing the position of the DragonHoard, which is spawned first, followed by the Dragon around it.

For random generation of enemies, a helper function in the EnemyCreator uses the provided rand() function to get a random number and selects a pseudo-random Enemy race based on a probability map. This approach ensures that the GameModel remains agnostic to how the Enemy is created, fully utilizing the Factory Design Pattern.

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

Enemies and the Player no longer share a common Decorator class for implementing abilities. Player abilities are handled in the triggerAbility() method, called as needed by the GameModel. For enemies, abilities are self-triggered through methods called on them (e.g., when a player attacks, when an enemy moves, or when an enemy dies). Each enemy object independently handles its abilities.

3: Items

3.1: Potions

Question: 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?

In our design, the EffectHandler class manages all potion effects for the player, allowing us to use the Decorator Design pattern to add features on top of the Player. The EffectHandler owns an Effect class instance, decorated by AtkModifier and DefModifier. To use a potion, the player's usePotion(int id) method is called from the GameModel, which augments the EffectHandler's decorator object. This adds the player's base attack to the EffectHandler's getAtkModifier() return value, giving the final value. When moving to a new floor, calling EffectHandler's removeAllEffects() from the nextFloor() method defined in the GameModel resets the decorator and clears all temporary effects.

3.2: Treasure

Question: 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?

Generating Items (Potions and Gold) is handled by ItemCreator, which provides the spawnTile(int x, int y, int id) method. This method constructs the appropriate Item based on the ID and returns a shared Tile pointer to the GameModel. For random item generation, we use separate probability maps for Gold and Potions, following the procedure described in 5.2. Enemies. This keeps the GameModel agnostic to item creation, fully utilizing the benefits of the Factory Design Pattern, reusing as much code as possible.

Final 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?

1. **Importance of documenting and modularizing code:**

With increased project complexity compared to previous assignments, implementing various mechanics of the game required writing a highly documented and modular code, which allowed us to build new features on top of already implemented ones.

2. **Git and version control**
Using version control was a crucial part of the project , which enabled us to collaborate in real time and stay on updated with each other's progress. Furthermore, flexible and cheap branching enabled us to experiment with new features without risking the stability of the master branch.

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

1. Early Design Planning:
Given a bit more time, we would pay closer attention to the initial design phase, especially to creating a more detailed UML diagram to better conceptually understand the links between project components before starting with implementation. This would help in identifying potential issues early and provide a clearer roadmap for development.

2. **Refactoring**: Refactoring the code to improve its structure and readability would be the next priority. This would involve breaking some classes down into smaller sub-classes to further improve cohesion.