Jonathan Boyle

Dungeon Battle Simulation: Design Decision Report

April 25th, 2024

The idea I selected for this project was to make an adventurer battle simulator. The simulation creates a grid that represents a battleground and puts four adventurers, each with a different class and specs, into this area. Every action the adventurer takes is either a movement in a random direction or a battle with another adventurer. This occurs if the two adventurers step on the same tile (same x and y coordinates). These four subclasses inherit from the Adventurer class, which further inherits from the general Unit class. The Adventurer class defines the common variables for attack, health, luck, x and y coordinates, a boolean to check if the adventurer is alive, and a mood state. The specs (attack, health, luck) for these adventurers are initialized in the main constructor. The Adventurer class also defines a times moved integer, which will increment every time an adventurer moves to track which move they are on in the log. Besides the previously discussed three specs, the name and simulation input for the adventurer are also initialized in the constructor. The Adventurer class, defines the general performAction method for all the adventurers, which is simply to move, if the unit is dead it will be moved off the grid. There are four adventurer subclass types: Knight, Mage, Rogue and Healer. The specs for these adventurers are initialized in the main constructor, but each type has a different unique skill. This uniqueSkill method is defined in the Adventurer class, but it is overridden differently in each subclass. Finally, the adventurer class holds many getters and setters for the variables discussed above, as well as some methods that control the mood of the adventurer. The BattleGrid Singleton class controls the main movement and battle logic for the simulation. It defines the grid instance for the simulation, and accepts adventurers onto the grid via an addAdventurer method. This class also handles weapon switching and potion creating, and the logic that dictates an adventurers mood after a battle. When the move method is run from the Adventurer class, the BattleGrid class moves the adventurer in a random direction, and then checks if the adventurer is engaged with another adventurer via another method isAdventurerEngaged. If the adventurer is engaged, the BattleGrid class moves into a different method for battling, where the two adventurers are given weapons and potions and battle. During battle, the adventurers have a chance to activate their unique skills, which gives them a specialized bonus in the battle. The battling goes until one of the two, or both adventurers are dead. The winning adventurer is then given a mood based on their health. When the simulation is over, the BattleGrid class outputs the surviving adventurer(s).

In terms of design patterns, the Adventurer class implements the Template Pattern. The Adventurer class implements the majority of logic for the subclasses, however the uniqueSkill method is only a general implementation and as such does not actually call any specific unique skills. This is because each subclass overrides this method in order to add its own implementation of the unique skill. Furthermore, The BattleGrid class implements the Singleton Pattern as mentioned earlier. This class implements a private constructor and a getInstance method. This allows the instance of the grid and its methods to be called from any other class. A helpful example is how the simulateMovement method is called from the performAction method in the Adventurer class through the Instance of the grid. The third design pattern used in this project was the Strategy Pattern. This pattern was used to allow for dynamic switching between weapons for the adventurers. Each battle, the adventurers are assigned a new weapon by the BattleGrid logic, which calls a new weapon strategy based on a random number using the WeaponSwitcher class. The weapon is then used which calls a method from the specific strategy that it has overridden from the interface. Adding onto the extra features, the Chain of Responsibility Pattern is implemented to include potions into the simulation. The PotionBuilder works through a chain of PotionHandlers that implement the PotionHandler interface. Each handler in the chain adds a new ingredient to the potion, until after all three handlers the potion is outputted into the pre-battle log. Finally, the fifth design pattern is the State Pattern. This was used to implement post-battle moods to the adventurers. Once an adventurer is dead and the battle is over, the surviving adventurer's mood is calculated based on their health after the battle. If the adventurer is above 40 health, they are happy, and if they are below 40 they are angry. Since when the surviving adventurer starts another battle they will be back at full health, they have the potential to shift into a happy mood from an angry mood if they do well on their next battle. If the adventurer that the algorithm is analyzing is dead, they are put into a neutral state, as the dead cannot exert emotions.

In terms of logic, I used a lot of switch cases with random integer calculations to decide what would happen in the simulation. This system not only decided movement, but also if a unique skill would trigger, what weapon an adventurer would get and what ingredients went into the potion. I wanted to take a very luck (RNG) based approach to this simulation to a level of roleplaying and unpredictability to the battling. For Data Structures, I used arrays to feed the specs of the two adventurers into the battling algorithm, one array for each adventurer with a size of 3 for the three specs. I also used an ArrayList to store the adventurers that were going into the grid. In this project I stuck to the principles of DRY, and did not make any repeat methods or code. For the subclasses of Adventurer and the inner classes that implemented the interfaces, I made each of them unique in their own way so as to not violate any rules. This practice also reinforced SOLID, where I followed Single Responsibility by making every class have a specific function or represent a specific object. This also helped to keep the Open/Closed rule and Liskov substitution with the Adventurer superclass. The three interfaces for weapons, potions and moods also agree with Interface Segregation and Dependency Inversion. Finally, the movement and battle actions each require a semaphore, of which there is only one. This way, no adventurers can move while two adventurers are in combat.