Legends: Monsters and Heros

By Sean Brady

11/13/2020

Intro

This is my quick as possible overview of my Legends: Monsters and Heros (MAH) implementation and design. In my design, I tried to fit the design patterns learned in class, (and a few others not yet covered) to several areas in my code where I felt that it would be appropriate. As this is my first time using most of these design patterns, I expect some of them will

This readme/ Document is organized in the packeges I implemented to complete the design. Although the graders have requested, we remove packages during submission, I have kept mine to help organize the various aspects in the game during this explanation and upload to our GitHub. The packages in this document are order from the basic to the more coupled aspects of the game.

MandH.items

The items package implements all the items in the game.

The Item abstract class is the super class for all items. Each item contains the basic stuff all items have, a name, cost and required level to use. Another small note is that the item class implements the namable interface, indicating objects with names in the game. Each subclass Item also implements a ‘item’Componenet interface. These are poorly named interfaces that have default methods to help pull out specific items from an item list and display them to the user in the game menus.

The first and most basic extension to item is Armor. Armor has an addition field to store an armors damage reduction. Armor also extends the equipable interface. All Armor in the game is equipable by a Player Character. getInfo is the first of many “toString” like methods I used. I tried to tailor string methods to produce a different depending on the situation. This is something we may want to clean up in our hybrid code.

The second extension is Weapon, remarkably like Armor. One might question why Weapons are Equipable, and not useable. This is because in my implementation attacks will always use the equipped weapon by a player. The Usable interface is currently only being used by potion and its signature only targets PlayerCharacters, which I think is another issue we could address to allow weapons, potions and spells all implement usable and target any character.

The next item extension is potion. Potion implements Usable, to use on PlayerCharacter. I think this class is another straightforward one, but I don’t like the use of the switch statement, which could become cumbersome as we add more affects types. Perhaps I could have used the strategy pattern for all the various effects?

Spells is the final item extension and the most interesting. One note before I begin is that I change the name of lightning spells to shock, because it is just what I am used to it being referred to. I use the Strategy pattern here to make concern strategies for the various spell types, such as fire spell, ice spell, etc. The class structure is interesting here as I even extend the effect interface to spell effects to include a constant reduction multiplier that all spells have to stats in the game. Each spell object “has a” spell effect of its type to execute on the enemies. DoSideEffects and get damage are used by Character classes in combat and get info is used in selection.

Item Database is the last class I should talk about. This class just reads all the given text files and puts them in Array Lists of their given type.

That’s my Items!

MandH.Characters.*

This package is just mostly some setup for the packages below and to be honest, I should have just combined all of them into one. But here we are.

The MandHCharacter is the super for all “character” classes in the game. Monsters are MandHCharacters, and Heros are MandHCharacters. All these characters have names, levels, strength dexterity and agility. Monsters in the assignment were said to have attack damage, defensive stat, and dodge chance. I found those to be too like the already existing stats for Players, so I just included them here. So, going forward in Monsters:

- Monster base damage = Strength
- Monster def stat = Dexterity
- Monster dodge chance = Agility.

Ok back to the super MandHCharacter. ManHCharacter also has 2 static vars, HP multiplier, which calculates all the hp of the characters during creation and level up and dexterity_Def_mutiplier, which is used to scale down some of the Armor stats in the game. I was having a hard time having fights end without this. MandHCharacter has getters to get all these stats and setters to change them if appropriate.

MandHCharacter implements the ICombatcomponent. This interface ensures that all characters created can attack, handle an attack towards themselves, and check to see if they are dead. ICombatComponent extends Namable since it is useful to have name available when working with Combat types.

Finally, before moving on to Monsters, I created a custom exception call CharacterDeadException to throw is an action is attempt by a dead character.

All Monsters are created using the Mob class. Since all monsters are only distinct due to their type and not functionality, a String is created to hold type. (EX, Dragons have higher damage, but that does not affect how they are created, die, or attack). Each Mob also has an integer to contain its currentHealth.

Mob also implements all the methods to handle combat from the combat interface. As well as a few helper methods. Another general note here, All of my random number generation is done using the Random classes next Int method, see Generate Random class for full implementation. The helper methods calculate if the attack was dodge and the reduced damage of the attack based on the monster's stats.

There is a database class to read the given monsters from the text files called MobDatabase.

Now for PlayerCharacter Class. The class is made up of a lot of other objects of classes. The Profession class encapsulates all the functionality of the class of the player. This includes level up behavior which uses the strategy pattern to determine how the profession levels up, and the String indicating the type. The MarketComponenet encapsulates Players interacting with the market, as well as the money the player has. InventoryComponent just stores all not equipped items. The "PlayerxComponent" encapsulate the player using/casting/ or equipping all the items using the previously mentioned 'item'Components. Finally each player object stores its current mana, total mana, current hp, current EXP, and currently equipped Armor and weapons. Lets move on to the methods of this class.

Starting with the getter and setters. All of these are needed for interactions with other classes. The PlayerCharacter class also implements the ICombatComponent and uses similar helper methods to the Mob class. PlayerCharacter classes implements usePotion, castSpell, equipArmor, equipWeapon and uses the encapsulated logic of the Components above. The next few methods handle the post fight logic of characters and in fight logic. The next two methods check for level ups use the level up behavior given at construction. The final few methods are various toString methods used depending on the situation.

There is a database to read the

This one might need more explanation but I hope you get the wide strokes.

MandH.Map

This package is much easier to explain than the last one. The 3 core classes are marker, MandHtile and gameboard.

Markers are just objects that indicate what type of tile the tile is. They have names and a display to display in the tile. This will make more sense in tile.

ManHTile is a tile which makes up the gameboard. Each tile has a rootmarker, a current marker, a display, a constant to indicate where the marker display should go and tileAction, which is a behavior that occurs any time a player party moves to that tile (Strategy pattern again) . The root marker and current marker allow the tile to remember what type it is as a player party marker inhabits it.

Gameboard is 3D ArrayList of tiles. The gameboard class also remembers which tile the current player is on. This could be derived from the marker, since there is only one marker, but I was running out of time to complete the project. Gameboard has 2 methods. movePlayer method takes advantage of the command pattern. Basically, what this does is inject which logic I would like to execute in the method by passing an object (ex, Move Up Command). I have create 4 MoveCommands to allow the party to move in each direction. Once a party successfully leaves a space, the remove Player marker is executed to restore the marker back to its root.

The gameboard is created using a factory method, randomly assigning the spaces on the board in the way specified in the assignment.

MandH

This is the last package and hopefully I haven’t bored you to tears yet. I

The fight class encapsulates the overhead fight logic.