Requirement 4

Astrologer and AstrologerStaff

In Assignment 1 & 2, we were introduced to the idea of players having a "class" which determines their HP and weapon that is assigned to them when the game starts. We decided since then to use the composition technique to handle this, meaning that players will have a combat class which determines their HP and weapon in the constructor of this class. Composition provides several benefits which make it easier to extend our design compared to another common approach, such as using Inheritance to make Samurai/Bandit/Wretch child classes of player. This made our Assignment 3 implementation very straightforward as it essentially extends the foundation developed in Assignment 1 and 2.

The first task in REQ4 is to create a new combat class, Astrologer, which has a new weapon, AstrologerStaff, and a new HP value. Here we already see the benefits of using composition for our CombatClass, as we simply need to create a new class that extends CombatClass and has the two values which matter: HP and weapon. We do not need to extend Player and add any unnecessary methods, as the CombatClass so far only overrides those two values.

For the AstrologerStaff, this weapon was made optional as part of the assignment requirements. Thus, we opted to take its base stats (hit rate and weapon) but implement it as a melee weapon, with no ranged attacks, for the sake of simplicity.

Thanks to composition we were able to reduce coupling between the Player and their combat classes, Samurai/Bandit/Wretch/Astrologer. This is also helpful for another way to extend the functionality of combat classes, in Ally and Invader.

Using CombatClass with Ally and Invader

Ally and Invader classes present a new form of extension for our CombatClass. Previously, it was demonstrated that creating a new CombatClass was very simple when Astrologer needed to be created. Now we can show the real benefit of using composition with the player - applying CombatClasses to other actors.

If Samurai, Bandit and so on were child classes of Player, this would be very tedious to implement. We would have to rename some classes and create new ones which serve the same purpose. SamuraiPlayer, BanditAlly, AstrologerInvader and so on would all need to exist and would be a waste of time to implement as it is unnecessarily repeating code.

Thus, the advantages of using composition become increasingly apparent when it comes to new actors who also have CombatClasses. This is how extensibility is improved; we can simply have the new actor use the CombatClass when it is instantiated. We are also encouraging code reuse, because we are able to reuse the same CombatClasses when instantiating a new Player, Ally, Invader, and so on, and no additional classes need to be created. Additionally, we are able to improve maintainability - since Player, Ally, and Invader reference the same HP value and weapon of a particular CombatClass, it would be easy to make changes - for example, changing what weapon all Bandits have, or the HP value of every Samurai, and this will apply to all actors with the CombatClass. This would be a very easy, one line change which promotes maintainability.

Ally and Invader
A few other key design choices have been made with regards to Ally and Invader. Allies and Invaders are unique NPCs with special behaviours. For Allies, a few additional capabilities and methods were created.

These are summoned using a special SummonAction which only the player can use, on any SummonSign. This SummonAction will randomly choose whether to generate an Invader using the existing EnemyFactory, or generate an Ally using the existing NpcManager.

Firstly, allies will wander around and will not attack friendly actors, such as other allies, the player, and traders. Because of this, a new species was created, "Allied", to simplify attack actions for non-enemies. Now, allies and players will not attack each other, themselves, or traders. This was implemented using the existing behaviours mechanism, previously used for enemies. For Invader this does not need to be overridden from the default Enemy behaviour, while the Ally has its own set of behaviours as it is meant to work differently.

Allies and Invaders have a number of limitations, such as not being able to buy and sell, interact with the site of lost grace, fast travel, and so on. Another new capability created was the "NPC" capability. This is done to equally deny advanced actions from all non-players, thus if an actor is NOT an NPC, only then they are able to perform certain special actions such as entering certain grounds, or even summoning more NPCs using the SummonSign. Our group agreed this was better than creating specific capabilities such as "CAN_SUMMON", "CAN_RECOVER_RUNES", etc, as it is a blanket way to deny most actions that should be only possible to players or other playable characters.

This is of course less modular and it becomes harder to allow or deny particular permissions, but the advantage is that it is easier to highlight actions that only playable characters can do. Given how the game is designed, it is unlikely that we would need to introduce something like an Enemy that can rest, or an Ally that can buy and sell, since their actions rely on primitive behaviours. However, if this were to be the case in the future, we could always omit the NPC capability from a particular entity in order to give it certain actions typically reserved for playable characters.

Our group agreed that this was a good use of capabilities, because not being an NPC does not necessarily guarantee the actor is the player, nor does being an NPC guarantee that it is not the player - it is not entirely the same as type checking. NPCs are simply a way to allow and deny certain actions that Allies, Enemies (including Invader), and Trader all have in common.

One place where we could've used NPC capability, but chose not to, is the DeathAction. Here we do something else by checking for the "HOSTILE_TO_ENEMY" and "FRIENDLY_TO_PLAYER" capabilities to trigger the action that drops items and doesn't respawn. This is because we don't want to include Traders, but we still want to include all non-player characters. These two enums are combat related enums so it made sense to rely on these instead.

For Invader, the first obvious design choice was to extend Enemy - it follows the player, attacks the player, cannot enter floors, and drops runes when killed. It also does not attack enemies of the same type - thus all

Invaders were given the same species "Invader". Invader does have one unique trait which differentiates it from typical enemies, but this will be explained in more detail below.

Both Allies and Invaders can despawn, but only during the player's death. This is similar to Enemy but enemies can despawn when the player rests as well. Since Invader extends enemy, this presented a unique design challenge - how to make Allies and a particular instance of Enemy despawn only during player death, whereas the rest of enemies always despawn when a player rests or dies.

The idea was initially floated to create a new class to store only Allies and Invaders, and create a new interface such as Despawnable. However, this was considered unnecessary and we eventually settled on making Allies and Invaders Resettables, with some modifications to the reset method.
For Invader we had to essentially override the non-abstract reset method which already exists in Enemy. This has some pros and cons, but our group decided that this was best as Invader was a special case and most enemies would still perform the same way. This could lead to issues with debugging as it is hard to tell which reset is actually being called, if this is done too many times. If such a case would occur my group would refactor our code in order to better accommodate this, if overriding an existing reset implementation becomes the norm.

ResetEvent
This was an idea (formerly ResetType) that originated from our Assignment 1's design, but scrapped during our Assignment 2 implementation. However, with Assignment 3's new requirements, we feel that it is once again necessary to implement.

The ResetEvent is a simple enum (REST/DEATH) that is passed to the ResetManager, then to the respective resettables, whenever a ResetAction is called. This was done in order to differentiate between the two events. Most Resettables such as Enemy and Player will ignore this as they will reset in the same way regardless of the event; however, this is relevant for allies and invaders as they can now despawn only during player death.

Finally, there was one final change made to Invader - their reset method is overridden from the Enemy reset method so that they only despawn after checking that a death triggered the reset method. This is in line with the open-closed principle - we do not need to modify Enemy to implement a special behaviour of the Invader class.