Design Rationale: Updated for Assignment 2

General changes

During the implementation of Design O' Souls, we have made some changes towards the way that the Actor, Item and Ground execute some actions. Previously, what we did was that an Actor, Item or even Ground object can straight away do an action without an Action class or Behaviour class. However, when we fully understood how the existing code works and how our implementations should work, we decided to add a Behaviour or Action class if an Actor, Item or even Ground needs to execute an action. We did this because it follows a SOLID design principle. Single Responsibility Principle, that ensures one class is responsible for only one action or object. This also allows us to reuse some action classes, which resulted in the code having less repetition of code. Furthermore, to follow the Interface Segregation Principle, we implemented interfaces to some classes so other classes do not have to implement methods they do not use.

UML Class Diagram 1 - Player

In this Player class diagram, we show all the actions' relationships with other classes and what will happen when they get executed. The actors can execute certain actions if they have the capability or behaviour that allows them to do so. We added the capabilities in our enum package to represent all the abilities and states the player and other actors can have. The reason we used capabilities the way we did was that we needed a way to make the player know what they can or can't do as a result of an action being executed as well as whatever actions they're able to perform according to the capabilities they possess.

We implemented a KillPlayerAction class in order to invoke all the necessary changes that occur when a player dies at once in one concise action because when a player dies, the game map resets and their souls will be saved at the token of souls at their death location.

We gave the Player has a private estusFlask variable which holds the Player's Estus Flask because it is a special item that can't be dropped and there will only be one, so if we ever need to point to it to check its remaining charges, we can simply use the Player's methods to do so easily.

We also gave the Player a private spawnPoint variable which holds the Player's latest Location where they used the Bonfire so that when they get reset, they are moved to the latest Bonfire used.

We also decided to implement a UseEstusFlaskAction so that every turn throughout the game the player can be prompted to use the EstusFlask and notified about the changes that occur when it is used. We did this because EstusFlask is able to be used throughout the game.

UML Class Diagram 2 - Enemies

In this Enemies class diagram, we have shown how an Enemy can execute an action. Basically, each enemy can have a list of behaviours that stores all behaviours that the enemy can own. For example, every enemy has an AttackBehaviour and FollowBehaviour. The reason we did this is that in every turn the code will run and check whether an enemy has a certain behaviour or not. If the enemy has a certain behaviour then it adds the corresponding action to the list of actions and executes it. For example, if an Undead is given the behaviour to follow the Player when the Player, then a FollowBehaviour action will be added into the list of actions and by doing so, every turn it will follow the Player.

Additionally, YhormTheGiant has an EmberFormBehaviour and with this behaviour, it will execute the EmberFormAction to make the adjacent square change to a Fire ground and resultantly a new instance of YhormsGreatMachete will be returned with a higher hit rate.

Therefore, by having different classes for each behaviour and for each action, we can ensure that we follow the Single Responsibility Principle where one class is only responsible for performing one action.

UML Class Diagram 3 - Vendor

In this Vendor class diagram, we have shown that the Vendor can have several different actions in order to interact with the Player when they are at adjacent squares. If the Player wants to trade with the Vendor, it can execute the BuyMenuAction, so that after the execution, a list of subsequent actions will be shown to the Player for them to buy weapons and improve health (BuyWeaponAction and IncreaseHPAction). When the Player finishes buying from the Vendor, the Player can execute the ExitBuyMenuAction to continue his journey.

We decided to do this in order to keep our implementation consistent throughout the game (a similar implementation is done in the Enemies class diagram). In implementing the menu this way, the actions displayed by the Vendor are much clearer since the options displayed in the Player's possible actions menu will be significantly less.

UML Class Diagram 4 - Grounds

In the Grounds class diagram, there is nothing much to show about the relationship between the Grounds classes and the Action class. It is because not every type of ground has a specific action to execute. The only two grounds that contain actions are the Bonfire and Valley. Basically, the Bonfire has only one action which is UseBonfireAction that runs the ResetManager to reset the world, and Valley has an action of ValleyDeathAction that kills the Player if the Player tries to go into it.

UML Class Diagram 5 - Weapons

In this Weapons class diagram, we have shown all the weapons in the game as well as the different capabilities that are required to use the weapon actions classes.

We did this because, as mentioned before, each turn we will check if a player has certain capabilities or not. In relation to the weapons classes, if the player has a certain capability then they can perform certain weapon actions. E.g. If the Player has the *CRITICAL_STRIKE* capability, then they are able to perform the AttackAction.

We also implemented several charging classes for StormRuler because in order to use the WindSlashAction we have to charge the weapon in 3 turns before the action is executable. Therefore, in order to charge the weapon, we execute the ChargeAction class, and in the ChargeAction class, it will return a ChargingAction to ensure the charging process will only execute for 3 turns. Then, after it is charged fully, a ChargedAction will be returned and this action will give the Player the WindSlash action that the Player can use to attack YhormTheGiant.

We implemented the charging action in this way because it was the simplest way we were able to do it, however, we understand that it is not an optimal implementation.