Requirement 3 - Game Reset

Our diagram for requirement 3 involves a few key components. There is a special ground The game reset occurs when the player dies or rests, which resets the player, enemies, and certain items. We also had to consider how to distinguish between the two types of resets - certain items such as runes would only be reset during the player's death.

Update: Reset is now a single unified action that despawns enemies, resets the player's HP, and resets the uses of the Flask of Crimson Tears. The dropping of runes was considered part of the death action, and similarly setting the player's spawn point is considered part of the rest action. Thus, there is no need to distinguish between "resets" as the events exclusive to resting/death are simply part of that action. With this, ResetAction was also created, so that it is easy to call on a reset from any event that triggers it.

PlayerDeathAction and Site of Lost Grace

These are the two main events that trigger a reset. The first is the Site of Lost Grace is simply a special ground (extends Ground) that returns a rest action that, if the player selects it, triggers a reset.

The second event is the player's death. The PlayerDeathAction is currently the only death action that resets the game, it makes sense to extend the DeathAction and specify the implementation details for the player which calls on a reset. This is in line with the Open Closed principle, since the DeathAction is closed for modification, but open to extension via its child classes.

Update: The Site of Lost Grace remains the same, but PlayerDeathAction was removed. Instead, there is a unified DeathAction which does something different depending on the capabilities of the actor. This allows us to trigger a reset only during the Player's death while using the same DeathAction as enemies, which is highly extendable if there are more actors whose deaths cause a reset in the future.

Resettable Interface

The resettable interface is implemented by actors such as Player and Enemy, and by items such as Flask of Crimson Tears and Runes. This means that when an action is triggered that is supposed to reset the game, it will call on the ResetManager, so that all resettables that have been added to the ResetManager will use their reset method.

The Resettable interface only includes two things: a ResetType enum, and a reset method. This closely follows the interface segregation principle, as only classes that are affected by a reset will implement Resettable, while all other classes will not need to implement this.

ResetType Enumeration

The ResetType enum is used by anything that implements resettable. There are two types, rest and death, and while most actors/items have both, items like Runes only reset upon the player's death. This enum makes it easier to distinguish between which objects should reset.

Update: This was removed as it was considered unnecessary. Additionally, Runes are not resettable but will simply change upon the player's death.

ResetManager
This is the main class that uses reset actions. The ResetManager is a singleton class that stores everything that can be reset, so any Resettable will be added upon their creation. Then, when a reset is called, the ResetManager is used to call their respective reset methods, which are part of the Resettable interface.

The ResetManager adheres to the single responsibility principle (SRP) as it is the sole class responsible for resetting everything in the World, and does not have any other responsibilities. This makes it more cohesive as there is only one reason to change this class. It also follows the open closed principle since it is closed to modification, but open to extension (by implementing more Resettables with new functionality, or Reset Enums). It also follows the dependency inversion principle, as it depends on the Resettable interface rather than concrete classes. This means that Resettable objects can derive from any class (Actors, Items, etc) and allows for easy extendability of the program (such as a new type of Resettable) without affecting the ResetManager class.

Update: This has not changed, but by adhering to the dependency inversion principle, we can reduce coupling between ResetManager and resettables. The functionality can change at any time, so it is more modular, flexible, and maintainable.

RunePile
We initially envisioned the concept of runes as a class that Player and Enemy have an instance of, so that it could be dropped upon an Enemy or Player's death. However, we realised that Enemies do not really drop runes in practice, but they "transfer" the amount if the player was the killer. Hence, we thought it was simpler to implement runes as an integer value innate to Player and Enemy. We still had to consider how the player would drop runes once he died, so that is how we conceived the idea of a RunePile.

The RunePile is an Item which is created upon the Player's death, and implements resettable. The RunePile is essentially just an object that stores the value of the player's runes before he dies, and if picked up, the player will regain the runes he dropped. If the player dies without picking it up, the old RunePile is discarded and a new one is created at the latest location of death.

The decision to make RunePile a singleton was due to the fact that only one can exist at any given time. This intuitively makes sense given our understanding of how the RunePile works - as soon as a reset is called that affects the player's runes, it will discard the current instance of RunePile on the ground (if any) and create a new one. Of course, if there are changes in the future where multiple sources can drop runes on the ground, we will revisit this design choice.

Update: RunePile is now managed by a single RuneManager, which is also globally accessible. This makes it easy to deal with RunePile since only one can exist at a time. Additionally, a new action RecoverRunesAction was created which overrides the original abstract PickUpAction. This is an abstraction that also helps improve the extendability of our code. By default, items will return a PickUpItemAction, which simply adds the item to the inventory of the player. However, if certain items behave differently (such as RunePile giving a certain number of runes to the player), this can be overridden for a custom effect - otherwise, the default action is used and no extra implementation is necessary. Thus, we are able to use the Open Closed principle since existing items will not need to be changed in order to accommodate the RunePile object with a unique, extended design.

Pros and Cons
This design is highly extendable with regards to the core reset mechanic. The ResetManager works with any object that implements Resettable, meaning that it is very easy to create new objects that can be reset (or take an existing one and add functionality to allow it to be reset). ~~Additionally, we can use the ResetType enumeration to include more events that trigger a reset in the future, such as beating the final boss, and only select certain Resettables to reset during this event.~~

Update: There is some coupling involved, such as the player's location being tracked by the RunePile, but this also helps us adhere to the Single Responsibility Principle and is considered a worthy tradeoff.

Flask of Crimson Tears, ConsumeAction, and Consumable
It was not realised initially but the Flask of Crimson Tears has a unique mechanic, being an item that can be consumed. Initially, we created a ConsumeFlaskAction, but we realised that more items could feature a similar mechanic in the future. Thus, a new interface Consumable was created, with a new action ConsumeAction. Each Consumable is consumed by an actor to give certain effects to that actor. The exact effects of consumption are different - this means that we can use abstractions and the Dependency Inversion Principle so that any Consumable will be able to work with the ConsumeAction, thus not requiring separate actions for each item, reducing coupling, and improving the maintainability and extendability of our program. Adding new Consumable items will be much easier to integrate with the existing Actions system because of the SOLID principles followed.