Design Rationale REQ5

Design Rationale

Implementation & Why:

Created Respawner Class

- Responsible for respawning the actor that stores the Respawner object, adheres to Single Responsibility Principle.
- Fields:
    - Actor to respawn
    - All the active game maps
    - The destination to respawn the actor
- Resets health and stamina of the actor (Player)
- Respawns the player at the respawnPoint
- Goes through each active map for the map and kills all enemies except the Boss, locks all gates, removes all runes, and drops actor's balance where the actor died last.

- Created Respawnable interface:
    - Defines a method (respawn(Location deathLocation)) that all Actors that implement Respawnable must have
    - Adheres to Interface Segregation Principle
- Modified Player:
    - Player now stores a Respawner object
    - Player implements Respawnable
        - Defined respawn method that calls Respawn's respawn
    - When both unconscious() methods are called, the Respawner's respawn method is called. Adheres to DRY as we don't have to repeat respawning code for both unconscious() methods, we can simply call Respawn's respawn
    - Added a fancy message to show the player was respawned
- Modified Application
    - Called player.setRespawner() to create the respawner for the player

Pros:

- Respawning functionality can be changed without affecting Player code, adhering to good encapsulation
- Through the effective use of capabilities, Respawner will work for any amount of bosses. For example, if we had used 'instance of' we would need to check for each boss before we reset its Health, but one capability condition works for all bosses
- Respawner's respawnPoint allows the actor to be respawned at any point defined in application. Hence, different actors can have different respawn points,

Cons:

- There is nesting of functions by using Respawnable interface as Player.unconcious() first calls its own respawnActor() which then calls Respawner.respawn(). This is the side effect of having a Respawnable interface, however, the benefits of the interface (extendability and abstraction) outweigh this small downside.

Extendability:

- Since Respawner is its own class, any Actor that needs respawning functionality can utilise it. This creates very extendable code and DRY as we won’t need to define respawning functionality again
- Any Actor class that wants to respawn can implement the Respawnable interface, ensuring they have the respawn method. This allows consistency when extending code