REQUIREMENT 4 - Design Rationale

Classes Modified/Created	Roles/ Responsibilities	Rationale

Puddle	A class that represents a random puddle of water.
	Consuming the water restores mana and has a chance to spawn a scarab.
		Solution
		• When creating the base for the assignment fire-resistance wasn't added to the puddle class constructor so I made sure to add that
		• In the previous assignment I knew it was important to change to make sure there was a "Consumable" interface rather than a "ConsumableItem" Abstract class.
		  ○ This was because puddles only one class can be extended in Java, meaning the puddle class couldn't extend both ground and consumable
		  ○ "ConsumableItem" would have extended item and given puddles aren't a tangible item this wouldn't have made sense
		• This futureproofing allowed the "consume"method from the consumable interface to be easily implemented. The difficult part was ensuring that the puddle could only be consumed if stood on top of, rather than stood next to like other classes
		• I thus decided to create a new method named "findScarabSpawnLoction"
		
		RELEVANT SOLID PRINCIPLES
		• **SRP:** The consume method handles the effect of consuming the puddle, while findScarabSpawnLocation determines the scarab's spawn location, keeping responsibilities separate and adhering to the Single Responsibility Principle.
		• **ISP:** The class only implements the focused Consumable interface, avoiding unnecessary methods. This keeps interactions simple and relevant to consumption actions.

Scarab	it wanders around and can be consumed. The creature contains many treasures	**Solution**
		• I decided not to extend "Enemy" but instead extend "Actor". This is because "By default, an enemy has an attack behaviour with high priority and a wander behaviour priority with a low priority". Due to the fact that the Scarab can't attack the tarnished, it made more sense in terms of both code and lore to extend the Scarab as a Actor. This also helps adhere to LSP as the Scarab is not tied up to a class it isn't closely tied with, making it much more substitutable in the future
		• I wanted to create a similar design to the FurnaceGolem specifically in terms of the use of hashmaps so that behaviours can be ordered
		• I also extended the unconscious variable from the actor superclass. By doing so, I was able to access its game map parametre and thus create an instance of this game map from which I can cause an explosion on the specific place the Scarab dies. Considering the unconscious method already removed the player from the map, this route made the most sense to alter given this was the condition (i.e death) for an explosion to occur
		  ○ Alternatively, I could have made a new "explosion" or "death effect" interface
		  ○ Pros of the current method:
		  ■ Simplicity: Directly overriding the method is straightforward and easy to understand.

		■ Single-Class Responsibility: All behaviours related to the Scarab are contained within the Scarab class, making it easy to locate related logic. ■ Immediate Access to Actor State: You have direct access to the state and attributes of the Scarab class and can modify its behaviours more easily. ■ Minimal Overhead: Since the logic is contained within a single class, there's no additional complexity or dependency management. ○ Cons of the current method ■ Partial Violation of Single Responsibility Principle (SRP): The Scarab class is responsible for multiple behaviours—movement, death effects, and interaction handling—making it harder to maintain and extend. ■ Limited Reusability: The explosion behaviour is tightly coupled to the Scarab class, making it difficult to reuse for other classes that might have similar effects. Given the Golem can also can also cause an explosion, this is even more so true ■ Potential for Code Duplication: If you need to create other classes with unique unconscious behaviours, you might end up duplicating similar code in multiple places. ■ Tight Coupling: Changes to the explosion or unconscious behaviour (in the engine) directly impact the Scarab class, making it less flexible. **RELEVANT SOLID PRINCIPLES** ● **SRP:** The Scarab class has a focused responsibility: representing a wandering creature in the game that can be interacted with in various ways (e.g., attacked, consumed). It encapsulates behaviours related to its movement, interactions with other actors, and its specific death mechanics. ● **OCP:** The Scarab class is open for extension but closed for modification. New behaviours or interactions can be added without altering the core implementation by adding new entries to the scarabBehaviours map or extending the Consumable interface. For example, if more complex death effects are needed, they can be added by overriding or extending the unconscious method in a subclass.
HealingStatusEffect	Represents a status effect that causes an actor to heal overtime or heal temporarily depending on whether the ability is labelled as recurrent or not as the last parameter	For this I had two ideas: 1) Create Two classes that use the tick method. The difference: a) Heal __ hp every turn over the course of __ turns (needed for the crimson tear functionality) b) Heal __ hp in one turn that lasts __ turns (needed for the straight consumption of a scarab) 2) Create One class that uses the tick method but use a boolean named "recurring" to differentiate between the needs of the two functionality depending on the scenario I went with the 2nd option because I felt the pros outweigh the cons: **Pros of the 2nd option:** 1. **Simpler Code**: One class handles both types of healing, reducing duplication and making the codebase smaller.

		2. **Flexible**: Easily switch between recurring and one-time healing using a single boolean. 3. **Easy Maintenance**: Fewer classes to manage, making it easier to update or refactor in the future. **Cons of the 2nd option:** 1. **More Complex Logic**: The code can get harder to read and manage as it handles both behaviours with conditionals. 2. **Less Readable**: Future developers might find it confusing since the class handles two different healing behaviours. 3. **Harder to Scale**: If new healing types are added, the boolean approach might not be enough, making the class more complicated.