## Requirement 1 - Environments & Enemies

The diagram represents an object-oriented design for requirement 1 and displays the necessary classes and their relationships with each other in order to fulfil the needs of the overall system. The diagram is organised by package, in the top half we have classes from the provided game engine while the bottom half displays the classes and packages within the game package which have been modified and extended to implement the required features.

The first design choice I would like to highlight is the inclusion of the abstract Spawner class that extends the Ground abstract class in the engine. I chose to design the system in such a way so that the Ground classes that spawn enemies such as Graveyard, PuddleOfWater and GustOfWind will inherit from this Spawner class and are forced to implement its abstract methods. This adheres to the DRY (Do not repeat yourself) design principle as all 3 of these concrete classes use similar logic to spawn enemies which can be defined within the Spawner abstract class itself. This also makes the design highly extensible as if we decide to change how spawning behaviour is managed in the future, we can simply make changes to the Spawner class directly instead of having to modify every Ground class which spawns enemies. This approach also adheres to the open and closed principle, as the Spawner class is meant to be extended by its children to spawn specific types of enemies instead of directly modified itself. Additionally, this design obeys the LSP (Liskov substitution principle) as Spawner objects can be used in Place of any Ground object without breaking the system. The one con of this design is that we end up coupling the Spawner class and its children quite tightly due to their inheritance relationship, however this is not a significant disadvantage as the Graveyard, PuddleOfWater and GustOfWind classes all follow the SRP (Single responsibility principle) and are not overly complex or bloated as most of their needs have been abstracted into the Spawner abstract class.

The next design choice I want to highlight is the use of the factory design pattern by implementing an EnemyFactory class. This class follows the SRP principle as its only responsibility is to create and return new Enemy objects through the use of static methods. I have chosen to handle the creation of objects in this manner as it allows us to make use of IoC (Inversion of control) and DI (Dependency injection) patterns when dealing with other Classes that depend on Enemy Objects such as the Spawner class and its children as well as a few other classes that spawn enemies. This makes our design more testable and robust as the objects which depend on the Enemy class won't have to worry about creating the object themselves resulting in loosely coupled classes. The alternative to this approach would have been to implement the EnemyFactory class as a singleton object with a static getFactory method, however due to the stateless nature of this class it is unnecessary to maintain an instance of it as the creation of new EnemyObjects can be done statically. The con of this design is that it creates some mental debt as the creation of enemies has now been abstracted into the factory, thus every time a new Enemy class is created, the developer must remember to add a static factory method to the EnemyFactory class.

The third design choice I would like to highlight is the inclusion of a Mortal abstract class which extends Actor and Enemy abstract class which extends Mortal. The idea behind this is that the mortal abstract class defines an actor who is capable of dying, and hence should have some logic that supports DeathActions while the Enemy class defines an actor who is an enemy to the player which will involve the use of Behaviours, specific enums, and SkillActions. Since all enemies can die, it would also make sense to have Enemy extend Mortal as now all children of the Enemy class must implement support for DeathActions. I have chosen not to implement Mortal as an interface for the key reason that death and actors are closely related in that many children of the actor class will need to support DeathActions while many objects that implement death require methods from the Actor class such as the hurt method, hence it makes sense to involve it in the inheritance hierarchy. You may then ask however, why I choose to separate this functionality into two separate abstract classes

rather than simply baking it all into the Enemy abstract class, this decision was made in order to adhere to the Interface segregation principle, not every Actor who can die is necessarily an Enemy, hence why should Actors such as the Player have to implement methods from the Enemy class which is does not depend on or use. Furthermore, this architecture also adheres to the DRY principle, as a lot of code in concrete enemy classes is reused from abstract classes, the LSP principle, as all children of Mortal and Enemy can be substituted for any Actor object, the SRP principle as the Enemy and Mortal classes have a single well defined responsibility, and finally the DI (Dependency inversion) as other parts of the system such as DeathActions and Spawners don't have to depend on concrete implementations of a class and can instead depend on the abstractions provided by Mortal and the Enemy class respectively. Ultimately, this approach allows for greater extensibility as now that death and enemies are not coupled, we can extend the system more freely as not every Actor needs to extend Enemy to be Mortal. A major con of this approach however is that the system has become more complicated due to the higher number of abstractions.

Another design choice worth highlighting is how behaviours have been implemented. Since all behaviours implement the Behaviour Interface, they must implement the GetAction method which returns an Action. This follows the design by contract paradigm as we have defined a formal, precise and verifiable interface within the Behaviour interface that will allow other components of our system to extend and use it easily.

The fourth major design choice made is the inclusion of a DeathAction abstract class, SkillAction abstract class and Skilled interface. DeathActions and SkillActions are both a form of action and naturally extend the Action class, however they are both abstract as there is no single concrete definition of a death or skill action in the game that would be sufficient to cover all required features. For example, when a normal enemy dies, it simply drops runes and weapons, however when the HeavySkeletalSwordsman dies, it spawns a PileOfBones instead of dropping runes and weapons. Hence, it's cleaner to loosely define how a DeathAction or SkillAction should behave through abstract methods and leave the detailed implementation to the concrete children of the class to implement. This fulfils the open and closed principle as both abstract classes are closed to modification while their children are open to extend their functionality. This also adheres to the LSP principle as DeathActions, SkillActions and their children can be used in place for any Action object without breaking the system. The Skilled interface is similar to the Mortal abstract class such that it was designed to allow other classes to use a specific action, in this case it was created as a way of providing other classes a common interface for interacting with SkillAction objects. However, it was designed as an interface instead of an abstract class as the classes that use SkillActions are Grossmesser which is a WeaponItem and GiantCrab, an Enemy. Since these classes are not closely related, it would not make sense to create an inheritance relationship between them. The use of the Skilled interface applies the DI (Dependency inversion) principle. In this case, the high-level modules GiantCrab and Grossmesser do not depend on the concrete implementation of skills but rather on the abstraction layer provided by the Skilled interface. The con behind this design however is that it introduces a lot of extra classes and complexity into the system, though I believe this will ultimately be worth it as this approach will allow for greater extensibility in the future as more Skill and Death actions can be added in the future without worrying about higher level modules breaking.