***Changes are below*
***Assignment 3 REQ extension details below*

A. Environment

Three different types of environments (graveyard, gust of wind and puddle water) are concrete classes that classified as spawning grounds. These spawning grounds share common attributes to Dirt & Floor classes where each dictates allowable actions apart from displaying their character on game map. In addition to these functions, spawning grounds have ability to create specific enemy based on type.

SpawningGround extends Ground
SpawningGround---<<uses>>--->Utils
SpawningGround inherits Ground. Spawning ground shares the common similar methods of displaying the ground character and set allowable actions. Abstraction of Ground allows for environment to be generated dynamically based on enemy spawn. Further abstraction of spawning ground gives open extension for different spawning grounds of different enemies. In spawning grounds, there are methods for despawn and spawn. These are calculated in using a static probability calculator in Utils. A dependency relationship is formed through calling an instance of Utils.

Graveyard extends SpawningGround,
GustOfWind extends SpawningGround,
PuddleOfWater extends SpawningGround
Three different spawning grounds inherit two additional common methods that every spawning ground contains, which are despawning and spawning. Calculation of these actions is based on probability on each turn and can accept various inputs of probability. Reusability of the methods adheres to DRY principle.

Graveyard---<<contains>>--->HeavySkeletonSwordsman,
GustOfWind---<<contains>>--->LoneWolf,
PuddleOfWater---<<contains>>--->GiantCrab
Each spawning ground spawns one kind of enemy. Every time an environment is called, the environment will generate an enemy object based on a probability.
An enemy object (whether null or exists) will be stored in the environment class as attribute permanently to keep track on its existence. This creates an association relationship between enemy creation and its place of creation highlights the sole purpose of breeding enemies (Single responsibility principle).

B. Enemies

Three different enemies are presented in the game as non-playable character (NPC). They exist as an Actor with hit points and items inside their inventory. The non-playable state does not restrict its motion as they contain interactable behaviours and capabilities.

HostileCreature extends Actor
HostileCreature inherit Actor class as the three different hostile creatures have identical methods and attributes (just not playable by user). The core state of Actor gives HostileCreature class full necessary adaptation (Liskov's Substitution Principle) for extending itself to creatures with specific behaviours and capabilities.

HostileCreature---<<has>>--->Status,
HeavySkeletonSwordsman---<<uses>>--->Status,
LoneWolf---<<uses>>--->Status,
GiantCrab---<<uses>>--->Status
Hostile creature can attack other hostile creature but not of their own kind (except Giant Crab and Heavy Skeleton Swordsman). Status enumeration helps to identify actors that are hostile to enemies. The status is kept as an attribute of abstract class so other type of hostile creatures can have access to its status when called for comparison in individual classes. Enumeration class helps to prevent magic strings error since status type can be represented as String. To further distinguish attack on hostile creatures' type, other attribute of enemy (display character) would be used for identification. An alternative of creating a new enumeration class of attack type would made subclasses tight on coupling while not utilising enemies' common attributes as identification.

HeavySkeletonSwordsman extends HostileCreature,
LoneWolf extends HostileCreature,
GiantCrab extends HostileCreature
Three types of hostile creatures inherit HostileCreature class since they share attributes commonly exist among hostile creatures.

HeavySkeletonSwordsman---<<has>>---> Behaviour,
LoneWolf---<<has>>--->Behaviour,
GiantCrab---<<has>>---> Behaviour,
WanderBehaviour---<<implements>>--->Behaviour,
FollowBehaviour---<<implements>>--->Behaviour
Hostile creatures have three behaviours implemented in their classes as attributes. These behaviours act as collection of action that cannot be chosen by the user. Directly, these behaviours are part of the creatures and are associated. Interface of Behaviour is chosen as attribute class as it can take in three other behaviours. Two separate behaviours implement the method of getAction, where in their own classes, the method is tailored to the structure of motion. Follow behaviour gives enemies action to follow actors within their bound and Wander behaviour gives enemies to move on itself.

LoneWolf---<<uses>>--->AttackAction,
GiantCrab---<<uses>>--->AttackAction,
GiantCrab---<<uses>>--->SlamAttackAction,
SlamAttackAction---<<uses>>--->AttackAction
Giant Crab can carry out attack on a sole target and area attack but only sole target attack for Lone Wolf. For a sole target attack, AttackAction is called as instance through allowable

actions for later play turn. Creating an instance would be a dependency relationship. This occurs to Lone Wolf and Giant Crab. However, Giant Crab has a unique area attack that slams any actors around its surroundings. Hence, a SlamAttackAction instance would be called in allowable actions for later use. The SlamAttackAction would consist of multiple of sole target attack, meaning multiple of AttackAction. SlamAttackAction can be made anew with new attack methods instead of utilising AttackAction but elements needed for attack would be similar to AttackAction and it will be repetitive (DRY principle). One drawbrack would be memory use of object call. If there are little enemies, reusing the codes would benefit than need for optimization.

HeavySkeletonSwordsman---<<uses>>--->PileOfBones
Heavy skeleton swordsman can turn into pile of bones once it gets killed. Pile of bones can revive back to swordsman if not hit successfully in three turns. In this case, piles of bones cannot be considered as a standalone Actor fully but as extension of Heavy Skeleton Swordsman. Heavy skeleton swordsman would create an instance of piles of bones as a temporary object to count the number of hits and proceed to death action or revival depending on successful hits. The use of dependency relation is sufficient in keeping track of progress.

ActorLocationsIterator---<<uses>>--->Actor
Enemies can be despawned at each turn and so removing a hostile creature from a location on a map can be done through remove method.

AttackAction extends Action,
DeathAction extends Action,
AttackAction---<<uses>>--->DeathAction,
AttackAction---<<has>>--->Actor,
DeathAction---<<has>>--->Actor,
AttackAction---<<has>>--->Weapon
Hostile creatures have to abilities to attack other enemies with certain conditions. However, the procedure of attack remains similar for all enemies regardless. Attack and death action inherits Action class as Action class provides needed abstract methods for definition of attack (Dependency Inversion Principle). Attack action needs actor as target and weapon of choice for combat, hence the need of keeping these as attributes for accessing the target counterattack behaviour and weapons attack behaviour (e.g., hit points). Hence, they have association relationship. Once a targeted hostile creature has been defeated after an attack, an instance of Death Action would be called. Death action only requires dependency relationship to perform any item drop from target and updates target existence in the game map.

C. Weapons

HeavySkeletonSwordsman---<<has>>--->Grossmesser,
WeaponItem---<<implements>>--->Weapon,
Grossmesser extends WeaponItem,
WeaponItem extends Item

Heavy skeleton swordsman carries a weapon item called Grossmesser. This weapon becomes an attribute for the swordsman with association relationship. Grosmesser needs methods and attributes in Weapon Item abstract class to be classified as one (Dependency Inversion Principle). Therefore, Grossmesser inherits Weapon Item. Weapon Item shares similar methods in Weapon interface (Weapon Item implements Weapon) and extends itself as Item that an Actor can have.

HeavySkeletonSwordsman---<<uses>>--->AttackAction,
AttackAction---<<uses>>--->Grossmesser,
HeavySkeletonSwordsman---<<uses>>--->SpinningAttackAction,
SpinningAttackAction---<<uses>>--->AttackAction,
Grosmesser is a weapon used by HeavySkeletonSwordsman for attack. It can be used for sole target attack or area attack. For sole target attack, HeavySkeletonSwordsman would call an instance of AttackAction with Grossmesser object, and it establishes a dependency relationship for the attack and the use of Grosmesser weapon object. Alternatively, HeavySkeletonSwordsman may perform a SpinningAttackAction with Grossmesser. SpinningAttackAction would be created as an object in action list and utilises multiple call of AttackAction with Grossmesser as an object parameter. Like SlamAttackAction, we achieve DRY principle through reusing the codes from AttackAction. Same drawback goes to larger use of memory space if many enemies exist around.

Changes to REQ1

The majority of the existing classes in engine illustrated in the UML diagram has been removed (includes the capabilities package, dropWeapon and dropItem Action, Weapon Interface and Instrinsic Weapon classes). The big change to this design would include Attack Behaviour class to facilitate attack action carried out by NPC hostile creatures to their enemies and proper implementation of Pile of Bones that was missing last time.

Some dependency/relationship arrows have been removed due to design change or new classes that made the previous relationship redundant.

AttackBehaviour---<<uses>>--->AttackAction,SpinningAttackAction,SlamAttackAction
After a better understanding of engine, it is decided to dedicate a separate method in interacting NPC hostile creature attack action. Player would carry out attacks in action list while NPC hostile behaviour would obtain attack behaviour as default. Attack behaviour would be the priority of enemy, followed by follow behaviour and wandering. The choice of going for behaviour over setting conditions in allowable actions in each hostile creature remains complex to configure with existing behaviours and NPC cannot choice actions. Since player has only simple attack style, it is best to emphasize design on attack style from NPCs. Hence by nature, designing attack behaviour is the way-to-go. In Attack Behaviour, we can specifically determine which attack style for the NPC attacker onto its enemy (whether that enemy be NPC creatures or player). Through series of conditions and looping all exits available, we find the attack action to be executed if any. Overall, this design enables us to further extend any new attack style for NPC creatures since NPC creatures have diverse attack behaviours. With each creation of attack action style, it has dependency relationship for the three attack styles.

DeathAction---<<uses>>--->PileOfBones
Pile of Bones can be formed after Heavy Skeleton Swordsman is killed. The nature of printable display for the game map shows the implementation of pile of bones as actor makes sure the converted form is shown. With an actor onsite, it is automatically uncrossable for others and suspectable for attack by others. The existing features of Actor makes it possible to achieve the two forementioned requirement. With that, heavy skeleton swordsman would be removed and replaced with pile of bones. The placement of pile of bones in death action ensures the effect happens swiftly instead of replacing it during location tick. DeathAction has a dependency relationship with Pile of Bones.

Assignment 3 REQ Extension (no optional)

Cliff extends Ground,
GoldenFogDoor extends Ground,
Application---<<has>>--->GoldenFogDoor
Application---<<uses>>--->FancyMessage
Cliff is the new ground on a map and instantly kills player when stepped on it. It inherits Ground as Cliff is like subclasses of Ground where, for example, it can be traversed over with restrictions like Floor. Golden fog door also inherits Ground due to similarities in giving ability for player to move to another map. As total number of doors are limited in the world and setting automated destination to the doors is very complex, they will be set at precise location of a map under Application class manually. The other maps (represented by list of String) will be added to world in Application class as well. When Site of Lost Grace is discovered, fancy message of “LOST OF GRACE DISCOVERED” will be revealed.