***Changes are below*
***Assignment REQ creative requirements below. Changes to enemy factory is explained below.*

The game introduces map partition, more enemies and additional weapon is introduced. Gamemap has been divided into West (Right half) and East (Left half). Three environments in each half spawn different enemies but carries some similar characteristics. For example, Graveyard on the West spawns HeavySkeletonSwordsman while Graveyard on the East spawns SkeletalBandit but they are still the same type regardless of their geographic difference. An additional weapon is introduced for SkeletonBandit called Scimitar.

The additional features model around design rationale of REQ 1. As a result, features would be modularised with existing package and classes to prove the extendibility of REQ 1 (Open-closed principle) and obey the importance for abstraction (Dependency inversion principle).

A. Environment

SpawningGround---<<has>>--->Location,
SpawningGround---<<has>>--->Direction,
The partition of GameMap for spawning grounds can be done with identification of the location of ground. We inquire a parameter of Location in the constructor of spawning ground and identify the direction from the x and y coordinates. The identified direction would be set with an enum type of Direction in spawning ground Direction type attribute. An alternative would be to create two separate maps, but this would require a combination of map for better experience or peripheral methods to transition character movement which can be complicated. Spawning ground has an association relationship with Location as an attribute to identify the direction and Direction enum type is used to prevent Magic Strings (String can be used but may cause error).

HostileCreature---<<has>>--->Direction,
HostileCreature---<<has>>--->Status
Hostile creatures have Direction enum type to help spawning grounds for setting the correct enemy set. Every hostile creature has a Direction on the game map and so it has association relationship. Status enum type tracks which type of enemy an enemy can attack. However, further identification would require display character when performing attack action. Using Direction enum type avoids Magic Strings error.

SkeletonBandit extends HostileCreature,
GiantDog extends HostileCreature,
GiantCrayfish extends HostileCreature,
SkeletonBandit---<<uses>>--->Status,
GiantDog---<<uses>>--->Status,
GiantCrayfish---<<uses>>--->Status,
The three additional enemies extend Hostile Creature class due to similarities with the three other existing enemies. They inherit the class and establish association relationship. The dependency relationship of Status type is for identification of other enemies' Status.

SkeletonBandit---<<uses>>--->PileOfBones
Just like HeavySkeletonSwordsman, SkeletonBandit can turn to Pile of Bones once killed. The temporary existence of Pile of Bones meant creation of Object to track the progress for revival or death action.This usage is dependency relationship.

SkeletonBandit---<<has>>--->Scimitar,
SkeletonBandit---<<uses>>--->AttackAction,
SkeletonBandit---<<uses>>--->SpinningAttackAction,
SpinningAttackAction---<<uses>>--->AttackAction,
AttackAction---<<uses>>--->Scimitar
Skeleton Bandit is the same type of enemy from graveyard but on the East side. Unlike Heavy Skeleton Swordsman, Skeleton Bandit carries Scimitar. This weapon belongs to the enemy, and it has an association relationship with the enemy. Skeleton Bandit attacks the same as Heavy Skeleton Swordsman. Skeleton Bandit can opt for AttackAction with Scimitar object or call SpinningAttackAction method with multiple AttackAction with Scimitar objects. The approach adheres to the one for HeavySkeletonSwordsman due to similar attack method codes and prove of reusing SpinningAttackAction benefit code reusability (DRY principle).

GiantCrab---<<uses>>--->SlamAttackAction,
GiantCrab---<<uses>>--->AttackAction,
GiantCrayfish---<<uses>>--->SlamAttackAction,
GiantCrayfish---<<uses>>--->AttackAction,
SlamAttackAction---<<uses>>--->AttackAction,
Giant Crab is same type of Lone Wolf but attacks like GiantCrayfish (the type of Puddle of Water). Both Giant Crab and Giant Crayfish attack via sole target attack or area attack via SlamAttackAction. Sole target attack consists of calling AttackAction object to actionlist, giving dependency relationship. The area attack is through SlamAttackAction which calls itself to action list and create multiple instances of AttackAction on all nearby enemies. Approach is like SpinningAttackAction to reduce repeated codes and allow modularity in the attack execution with repurposing methods from AttackAction class (extendibility).

Scimitar extends WeaponItem
MerhantKale---<<has>>--->WeaponItem
Scimitar can be sold by Merchant Kale and purchased by the player. Since Merchant Kale keeps weapon item as goods (association relationship), Scimitar inherits WeaponItem allows Merchant Kale to sell it. The inheritance of abstract class of WeaponItem makes it extendable for further weapon items (Open-closed principle). The inheritance is an association relationship as it derives as an Item that can be used as weapon for damage. Merchant Kale stores WeaponItem class objects for sale.

The design rationale for rest of the links within the game remain similar. Purchase Action is still the same approach but with additional item for purchase or sale. Interaction within the game is similar as there is no modification to the logic of game but only extension of game elements.

Changes to REQ 5

Majority of the general classes in engine package has been removed due to redundancy. Exisiting features from REQ 1 design would not be repeated in the REQ 5. This is reflected in removal of attack style for the new hostile creatures as they still have same attack behaviours to determine the attack action/style needed. As for trading of Scimitar (a new weapon), it will be an additional weapon item by Merchant Kale and something player can sell for in sell action. The new addition to the design would be the creation of NPC hostile creatures from spawning grounds on two different sides.

HeavySkeletonSwordsman extends GraveyardEnemy,
SkeletonBandit extends Graveyard,
LoneWolf extends GustOfWindEnemy,
GiantDog extends GustOfWindEnemy,
GiantCrab extends PuddleOfWaterEnemy,
GiantCrayfish extends PuddleOfWaterEnemy
Each spawning group has two types of enemies. They belong in the same enemy type, just spawn on either east or west side of the spawning ground. To minimise code repetition, three abstract classes of spawning ground enemy type are created to merge any similarities. This new layer offers new set of identification in status and follows Single Responsibility Principle. Future enemy of the same type can inherit any of the abstract classes and only define the unique values in its own class. It can help to reduce dependency on each enemy class.

GraveyardEnemy extends HostileCreature,
GustOfWindEnemy extends HostileCreature,
PuddleOfWater extends HostileCreature
The three enemy type classes inherit the functionalities in hostile creature class. Hostile creature class still manages most of the action in enemy play turn and allowable actions.

EastMapEnemyFactory implements EnemyFactory,
WestMapEnemyFactory implements EnemyFactory
The east and west enemy factory classes implements newEnemy function in EnemyFactory interface. The newEnemy function is designed to return a hostile creature based on direction of spawning ground. EnemyFactory can be extended in the future with more new methods for implementation but restricted modification due to its abstraction (Open-closed principle).

EastMapEnemyFactory---<<uses>>--->SkeletonBandit,GiantDog,GiantCrayfish
WestMapEnemyFactory---<<uses>>--->HeavySkeletonSwordsman,LoneWolf,GiantCrab
In each of the enemy factory, it will return a new instance of enemy based on the location of spawning ground provided and the spawning ground type itself.

SpawningGround---<<uses>>--->WestMapEnemyFactory,EastMapEnemyFactory
Inside spawning ground spawnHostileCreature method, new instance of either WestMapEnemyFactory or EastMapEnemyFactory is called based on direction of spawning ground. The subclasses of spawning ground will call the spawnHostileCreature method and utilises a broader search of enemy factory to determine enemy instance. Since all subclasses

of spawning grounds needs to know which enemy to spawn, calling a common search prevent code repetition in subclasses and reduces tight coupling on each subclass to determine on their own which creature to spawn.

SpawningGround---<<uses>>--->Direction
To determine direction of a spawning ground, it calls a static direction method from Utils class and returns a enum direction type (EAST or WEST). Having enum type is preferable over returning direction in String since the value is final and easily prevents magic string issue.

SkeletonBandit---<<has>>--->Scimitar
Skeleton bandit would have a scimitar as a weapon in its inventory. Scimitar would be considered as weapon item to be linked with Skeleton Bandit and it can be removed from the enemy. A new Scimitar instance is created and added to Skeleton Bandit inventory.

Changes to REQ 5 (for assignment 3)
The restrictive uses of generating enemies from West/East Enemy factory alone did not help the spawning of Stormveil castles enemies as they do not have direction requirement. Though Stormveil Castle enemies can adhere to the requirements of West/East for creating new instances, such practice forces class to use the generative tool (which does not follow Interface Segregation Principle). Hence, enemy factory related class and abstraction will only be tied to enemies that have direction. In the case of Stormveil Castle enemies, new instances will be called directly instead of going through Enemy Factory.

Assignment 3 REQ creative requirement

Conversation implements Conversable,
TraderConvo extends Conversation,
TraderConvo---<<uses>>--->Player,
Player---<<uses>>--->Status,
ConvoManager---<<creates>>--->TraderConvo,
ConvoAction extends Action,
ConvoAction---<<uses>>--->ConvoManager,
Trader---<<has>>--->ConvoAction
For this creative requirement, a short conversation feature is created between player and trader. Trader will converse with player with narration and player will respond through numbered options for response. This design takes into account of future extension of more conversation with other friendly actors or anyone and use of OOP and abstraction to easily segregate heavy and complex functionalities.

Conversable interface is created to store the two actions in a dialogue (which are talk and respond). An argument scene that involves more than dialogue , as an example, can utilise this conversable interface (Open-closed Principle) to act out a scene in the black actions.

The idea of having an abstract class of Conversation is to branch out to convos with different actors instead of implementing Conversable interface directly. This method helps ConvoManager to return different convos that are under Conservation abstract class

(Dependency Inversion Principle), without needing to set different convo getter. Through speaker display character, ConvoManager returns the right instance of convo class.

To perform conversation, it is done through ConvoAction which inherits Action class due to its integration within the game in play turn. Player will have a Talking Status to allow conversation. Trader would have ConvoAction in allowable actions for actors with Talking status to initiate (this case , the player). Sticking to Single Responsibility Principle, ConvoAction would execute the flow of a conversation solely. It uses ConvoManager to choose Convo class and does the talk and respond execution.

In the trader convo, trader will ask player for 5 runes for a conversation ; hence the use of Player instance to decrease runes amount.