Assignment 3 UML Diagrams and Design Rationale

REQ 1:

- **Cliff extends from <<abstract>> Ground (Inheritance / Generalization).**
  - ➢ By extending the abstract Ground class, we can create new ground classes with their appropriate display character.
  - ➢ This also enables us to override the method tick so that we can perform intended behaviour each play round.
  - ➢ In the Cliff class, we have used a decoupling method by using the “Actor” and “Location” object.
  - ➢ This design allows changes to the “Actor” and “Location” classes without significantly affecting the Cliff class.
  - ➢ This implementation also complies with the Dependency Segregation Principle (DIP).
  - ➢ As the Cliff class depends on the ‘Location’ and ‘Actor’ abstractions rather than the concrete class. Thus, changes in low-level code won’t mean that high-level code has to be refactored.
  - ➢ This adherence to the Dependency Inversion Principle allows us to change the underlying implementations of Actor and Location without having to modify the Cliff class.
  - ➢ Meanwhile promoting flexibility and modularity, as high-level modules (Cliff) are not directly dependent on low-level modules (Actor and Location), which can change more frequently.
  - ➢ As a result, changes in low-level code won't mandate refactoring of high-level code.

  - ➢ For future extension and scalability examples.
  - ➢ Let's say if we have added new actors as Actor’s subclasses into our game. That can survive on stepping on a Cliff.
  - ➢ We can simply just add a new capability such as (CAN_CLIMB_CLIFF) to this new actor subclass.
  - ➢ So under the tick method we can check whether the actor has this specific capability or not. And this doesn't require any modifications to the cliff classes’s dependencies (Actor and Location).
  - ➢ Instead, we are leveraging the abstraction provided by these classes, demonstrating our design’s scalability and possible future extension.
  - ➢ However, after all the pros and future extensions, there are still cons in this design.

- ➢ While the Dependency Inversion Principle leads to more flexible and maintainable code, it may also result in hidden dependencies if not implemented carefully.
- ➢ Such that we have multiple references to different classes' objects that create multiple dependencies.
- ➢ This could make code harder to understand, debug and difficult in terms of tracking class's relationships.
- ➢ Overall this design is viable and logical, as we know that engine classes' won't ever be modified. Which reduces the likelihood of cons affecting our overall design quality.

❖ GoldenFogDoor extends from <<abstract>> Ground (Inheritance / Generalization).

- ➢ By extending the abstract Ground class, we can create new ground
- ➢ classes with their appropriate display character.
- ➢ This also enables us to override the method tick so that we can perform intended behaviour each play round.
- ➢ Also, using the idea of encapsulation to ensure the variables can only be accessed and changed within the GoldenFogDoor itself. Which improves data integrity.
- ➢ The reason why we designed in this way is because we don't want any other class to potentially change the variables' attributes.
- ➢ Since this can lead to change of behaviour of the Door class. Which is the thing we want to avoid.
- ➢ Moreover, the overall class design complies with the Open/Close principle.
- ➢ This is because the GoldenFogDoor is open for potential extension (as it can be subclass) and closed for modification (as we are not allowed to change methods in Ground as it is located in the engine package).
- ➢ This enables us to add features, but in a way that doesn't change the way we use existing code.
- ➢ Which promotes future scalability. For example, if in the future there are several doors that behave differently. This can be turned into a subclass of the <<abstract>> Door extends from Grounds.
- ➢ And we are still able to access the methods in Ground and general methods from the Door without refactoring or changing any code that we currently have.
- ➢ Although the class itself can be further extended and scaled. There are still cons that exist.
- ➢ There is an inflexibility with door location updates. Currently, the GoldenFogDoor assumes that the currentLocation and destinationLocation will never change after the door is created.

- ➢ If a feature is added in the future that allows the relocation of doors, this could require significant modifications to the GoldenFogDoor and any other similar classes.
- ➢ However, this is done due to the fact that this project's map is seemingly fixed.Thus, we implement in this way that might sacrifice a bit of scalability but improve the simplicity of code and overall readability.

REQ 2:

❖ Barrack / Cage extends from <<abstract>> Environment (Inheritance / Generalization).

- ➢ The reason why we decided to explain both classes together is due to its similarity and implementation.
- ➢ By extending the abstract Environment class, we can create each environment with their appropriate display character.
- ➢ This enables the abstract Environments class to supply a generic method for initialising the correct corresponding symbols and generic shared methods for each child environment class.
- ➢ Both subclasses manage the record of the enemies (Actors) currently residing in their domain to prevent the spawning of new enemies in the presence of existing ones.
- ➢ Upon the generation of an enemy, a check is performed to verify whether the enemy is pursuing the player. If affirmative, the determineSpawn method is invoked directly.
- ➢ Alternatively, if an Actor is already stationed at the location, the environment initiates a despawn process where the specific enemy holds a 10% chance of being despawned.
- ➢ The relationship between the environment subclasses and the enemies within them is categorised as an association relationship, where each environment could potentially be linked to zero or many enemies.
- ➢ Enemies have the ability to pursue the player across different environments, thus, the relationship designed between the environment and the enemies facilitates zero or many associations.
- ➢ Each environment holds the responsibility of spawning an enemy given its specific environment and direction with a particular probability.
- ➢ Both classes extend from abstract Environments class, thus adhering to the Open/Closed Principle (OCP) where it extends the functionality of the Environments class without modifying its implementation.
- ➢ Since both classes basically modify the general method tick within the parent class, then perform modification under each subclass to achieve desired behaviours under different environments.

- ➢ The pros and potential scalability of doing this is that when there are new environments being added into the game. We can simply repeat the process of how we implement Barrack and Cage.

- ➢ If there are new behaviours that only a new environment can perform. We can just create new methods under the new environment. Which avoids potential refactoring and dependencies issues.
- ➢ And this helps to accomplish reusability since the environment is placed in a superclass, helping reduce code duplication.
- ➢ However, there might still be cons.
- ➢ If there are new features that are added into the game, for example more complex spawning rules, player levels, number of existing enemies, etc. And the tick function wouldnt support this without significant modification.
- ➢ Which is not deemed to be that flexible and might potentially limit the way of implementing new features in the future.
- ➢ Nevertheless, based on the scope of this assignment and the progressive implementation of new features and new rules of spawning.
- ➢ The way we implemented it is still fairly viable and reasonable in terms of overall design. And we barely ran into errors when implementing new features.
- ➢ Therefore, justified the way of this design and implementation.

❖ Dog / GodrickSolider extends from <<abstract>> Enemy (Inheritance / Generalization).

- ➢ The reason why we decided to explain both classes together is due to its similarity and two lines of code.
- ➢ By extending from abstract Enemy class, this enables us to set up the basic attribute of both new enemies. Such as hit points, runes drop bounds, name, displayChar.
- ➢ Initialising all the essential attributes that both enemies possess.
- ➢ There is nothing to discuss much based on how minimal the code both Dog and godDrickSolider sub classes have.

REQ 3:

❖ FingerReaderEnia extends from <<abstract>>Actor(Inheritance/ Generalization).

- ➢ Enia extends from abstract actor class because Enia can perform actions.
- ➢ Extending from actor also allows us to override AllowableActions and PlayTurn, this allows us to modify these methods based on our requirements.
- ➢ Extending from the abstract Actor class and implementing Merchant interface will prevent breaking open/close principles. Even when we add a new feature to Enia class, the Actor class won’t break, since the code is extended by interface and abstract Actor class.

- ➢ This may cause the system to be too complex in larger projects by having too many abstract classes, but in this assignment, this should be a problem.
- ➢ There is also an UpdateMerchantInventory method for future extensions, when Enia is going to sell or purchase more items.

❖ **ItemTradeAction extends from <<abstract>> Action (Inheritance).**

- ➢ ItemTradeAction has association with class Merchant, Item and WeaponItem.
- ➢ Instead of having Enia use SellAction and PurchaseAction, we created an ItemTradeAction, trades item, which does involve any runes.
- ➢ By separating ItemTradeAction to purchasing, this follows the Single Responsibility principle(SRP), Purchase or Sell Action should handle trade.
- ➢ Sometimes following SRP can cause the classes to become too small and over complicates the design.
- ➢ This also doesn't break OCP, because any other merchant can also use ItemTradeAction.

❖ **ConsumeAction extends from <<abstract>> Action (Inheritance).**

- ➢ Consume action calls the consumable's consume method.
- ➢ This follows the OCP, because adding other consumables is easy when they can have their own consume method in their own class, instead of modifying the consume action. Prevents modification in the consume action class.
- ➢ ConsumeAction extends from Action, anything modification done to ConsumeAction will not affect Action, therefore it follows OCP.

❖ **GoldenRunes extends from <<abstract>> Item and implements <<interface>> Consumable.**

- ➢ Having GoldenRunes class comply with the SRP(Single responsibility principle). The golden runes only have methods related to the golden runes.
- ➢ GoldenRunes extends from abstract class item and implements interface Consumable, this follows the OCP, because when we modify the GoldenRunes class, it does change anything in Consumable and Item.
- ➢ Also, by implementing Consumable, GoldenRunes will have the method consume, which every consumable can implement based on its own way to be consumed, instead of having consume action handle all different types of consume process.

REQ 4:

- **Invader extends from <<abstract>> Enemy and Ally Extends from Friendly.**
  - ➢ To maintain OCP, Invader class extends from Enemy class to maintain all the required behaviours and characteristics without having to re-implement it.
  - ➢ Since the Ally class is the first friendly class to exist, utilising abstraction would be appropriate here to maintain OCP. As a result, a new abstract class, the 'Friendly' class is made.
  - ➢ By having a new Friendly Class instead of creating just a concrete class for Ally, when new friendly classes are created in the future, the Friendly class can provide the general behaviours expected for a Friendly character. This as a result maintains OCP, as we can now create new friendly character extensions efficiently without modifying the parent code.
  - ➢ The benefits of this approach are that code can be reused and maintained well due to better readability.
  - ➢ The downsides this presents however, are the possibility of inflexible code or complex hierarchies. Any unique friendly characters with different behaviours may require different code and thus, extra steps may need to be taken to fulfil the behaviour requirements.

- **Summon Sign extends from Environment and SummonAction extends from Engine Action class**
  - ➢ To maintain OCP, Summon Sign extends from Environment to use the functionality from environment whilst also being a ground class (from the engine package)
  - ➢ Summon Sign uses Summon Action class to spawn an Ally or Invader on the same tile.
  - ➢ Summon action is used to spawn an Ally or Invader on the command of the player.
  - ➢ To maintain SRP, this action is separated from the Summon Sign and instead uses dependency injection in the Summon Sign class. This way the summon sign does not hold all responsibilities.
  - ➢ Although Summon Action could have been implemented inside the summon sign, this would be difficult to integrate into the current system as that would mean the summon sign would have to inherit two classes which should be avoided for possible conflicts.
  - ➢ Having a separate summon action class can also maintain OCP as different new summon environments can be created without having to rely on SummonSign. If Summon Sign was removed (in a scenario where Summon Action is implemented inside it), the new environments with summoning capabilities would fail.

❖ ChooseAstrologerAction Class and ChooseRandomClass Class

- ➢ For the new character archetype, the Astrologer, a new action class is made to be interacted with within the menu.
- ➢ As the implementation of Astrologer's Staff was optional, the Astrologer's starting weapon was replaced by a great knife.
- ➢ As the construction of Allies and Invaders are formed using existing character archetypes, the ChooseRandomClass class handles this responsibility.
- ➢ By separating these classes, SRP can be maintained as creating one action class containing all the archetypes, could be built into a god class.
- ➢ Although abstraction could have been used, since the Archetype action classes did not require a complex structure (nor are they expected to), these classes were created as concrete classes. The methods from the Action class from the engine package were sufficient.

REQ 5 - WeaponItems with Status Effects:

❖ Creative Requirement Rules

- ➢ This requirement covers special weapons with status effects that are applied on hit. Status Effects are effects that last a few turns after a successful hit. The Status Effects created during this requirement are Poison Damage and Burn Damage, applied by the poison dagger and fire blade respectively.
- ➢ This creative requirement adheres to SOLID principles as
  - ■ This new feature requires us to separate responsibilities between an attack, apply effect, activate effect and weapon creation functions.
  - ■ Future implementation needs to be considered for new forms of weapons with status effects.
  - ■ The creation of status effects and the weapons using them need to also be created so that it integrates into the current system and uses the engine to implement.
  - ■ The implementation of this new requirement should not affect the current system.
- ➢ This requirement uses multiple classes from the engine to achieve its functionality and integration to the current system which are Action, Behaviour and WeaponItem Classes.
- ➢ This requirement reuses the AreaAttackAction (A function reimplemented in A3) specifically for the FireBlade's Burn status effect where it will deal Area of Effect Damage.

- ➢ Through creating a new abstract class, StatusEffectWeapon extending from WeaponItem, all the behaviours and functionality can be implemented through this class and further extended in future implementation.
- ➢ New capabilities added through the enum class EffectCapability which contains the values AFFECTED and HAS_STATUS_EFFECT.
- ➢ AFFECTED is to notify that an actor is affected by a status effect
- ➢ HAS_STATUS_EFFECT is a WeaponItem capability that tells the code that the weapon can use status effects.

Changes Made:

❖ Making Kale not a singleton.

- ➢ Merchant Kale will only be created once, because there is only one merchant Kale on the map. Therefore there is no need to create Kale as a singleton.

❖ Removed DropRunes.

- ➢ This class is only accessible from RunesManager calling it. Therefore, it makes sense to just add it in RunesManager.

❖ Added Consumables interface.

- ➢ Allows more consumables to be added in the future.
- ➢ Added all necessary methods for Consumables, such as consume, which allows Consumables to handle consume based on its own way.

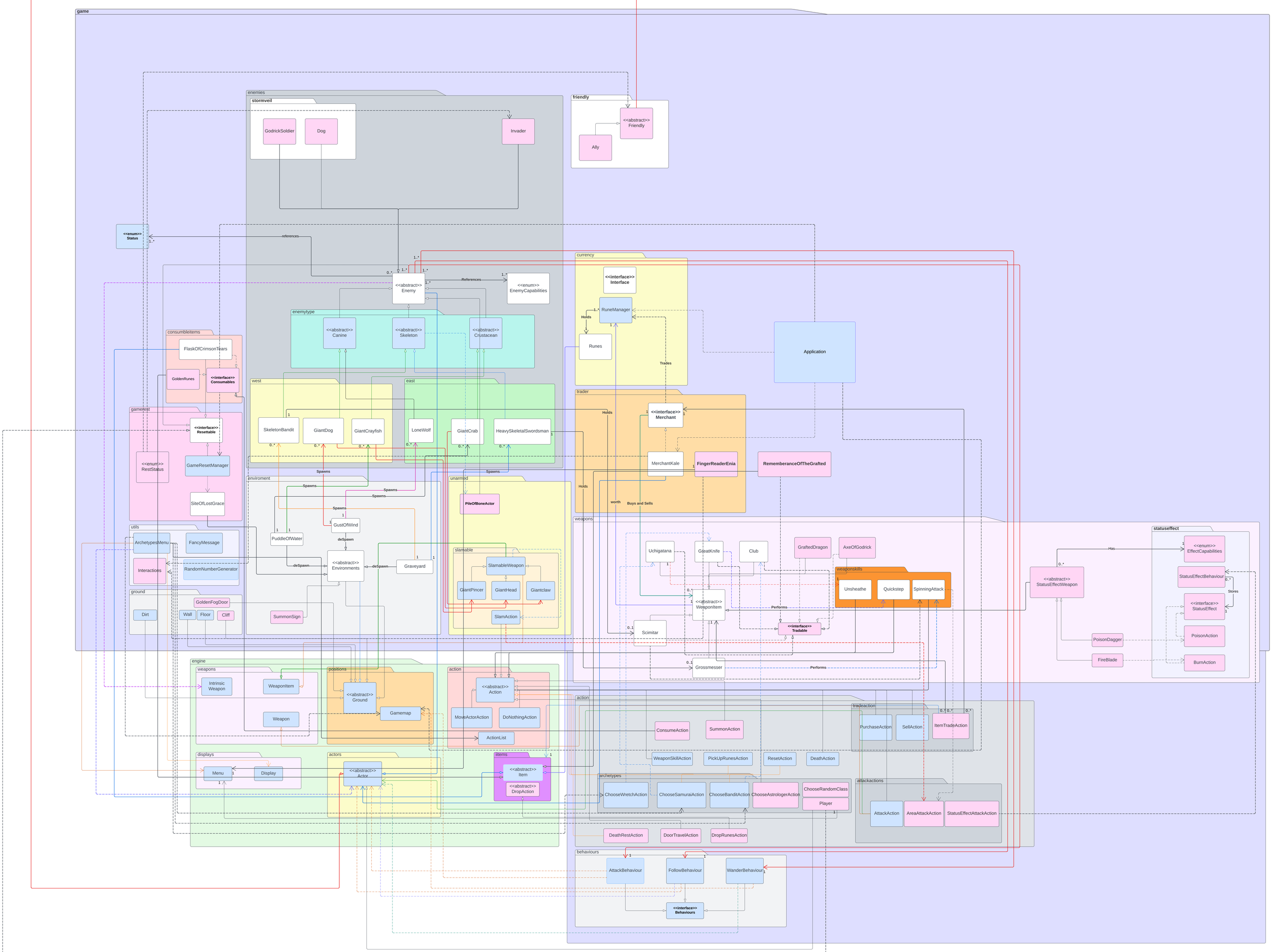
❖ Removed instanceof.

- ➢ Instead of checking if the actor died was Player using instanceof, I gave player a capability ALIVE, which can be used later with hasCapabilty to replace instanceof.

❖ Changes have been made from the feedback based on previous assignments.

- ➢ According to the feedback, we have removed the Spawn interface. As spawn method can be created inside an abstract Environment.
- ➢ So now all the child classes can just modify the general version of spawn and modify the codes. So it matches the behaviour that each subclass intends to perform.
- ➢ The Skeleton class pile of bones process is simplified as the previous implementation had over-complicated the process. Instead of using Actions and Behaviours, the Skeleton class now uses playturn to see if it is unconscious and uses this as an indicator to place the PileOfBones Actor at its location.
- ➢ An AreaAttackAction was implemented this time, to be used by SpinningAttack and SlamAction. The last implementation had code repetition between spinning attack and slam action and thus, this new

AreaAttackAction class fixed this issue. The implementation of AreaAttackAction also took a different approach, using location exits instead of radius. This benefits the code as it is much more readable and easier to integrate as it does not use nested for loops and coordinate calculations. The downside of this is that any AreaAttackActions affecting a radius larger than 1 tile, cannot be achieved as easily.


game
enemies
stormveil
GodrickSoldier
Dog
Invader
friendly
<<abstract>>
Friendly
Ally
<<enum>>
Status
references
1..*
0..*
<<abstract>>
Enemy
References
<<enum>>
EnemyCapabilities
enemytype
<<abstract>>
Canine
<<abstract>>
Skeleton
<<abstract>>
Crustacean
currency
<<interface>>
Interface
RuneManager
Holds
Runes
Trades
Application
consumbleitems
FlaskOfCrimsonTears
GoldenRunes
<<interface>>
Consumables
west
east
SkeletonBandit
GiantDog
GiantCrayfish
LoneWolf
GiantCrab
HeavySkeletalSwordsman
trader
<<interface>>
Merchant
MerchantKale
FingerReaderEnia
RememberanceOfTheGrafted
Holds
worth
Buys and Sells
gamerest
<<interface>>
Resettable
<<enum>>
RestStatus
GameResetManager
SiteOfLostGrace
Spawns
enviroment
GustOfWind
PuddleOfWater
deSpawn
<<abstract>>
Environments
Graveyard
SummonSign
unarmed
PileOfBoneActor
slamable
SlamableWeapon
GiantPincer
GiantHead
Giantclaw
SlamAction
weapons
Uchigatana
GreatKnife
Club
GraftedDragon
AxeOfGodrick
weaponskills
Unsheathe
Quickstep
SpinningAttack
<<abstract>>
WeaponItem
Performs
<<interface>>
Tradable
Scimitar
Grossmesser
statuseffect
<<enum>>
EffectCapabilities
StatusEffectBehaviour
Has
Stores
<<interface>>
StatusEffect
<<abstract>>
StatusEffectWeapon
PoisonDagger
FireBlade
PoisonAction
BurnAction
utils
ArchetypesMenu
FancyMessage
Interactions
RandomNumberGenerator
ground
Dirt
GoldenFogDoor
Wall
Floor
Cliff
engine
weapons
Intrinsic
Weapon
WeaponItem
Weapon
positions
<<abstract>>
Ground
Gamemap
action
<<abstract>>
Action
MoveActorAction
DoNothingAction
ActionList
displays
Menu
Display
actors
<<abstract>>
Actor
items
<<abstract>>
Item
<<abstract>>
DropAction
action
ConsumeAction
SummonAction
WeaponSkillAction
PickUpRunesAction
ResetAction
DeathAction
DeathRestAction
DoorTravelAction
DropRunesAction
tradeaction
PurchaseAction
SellAction
ItemTradeAction
archetypes
ChooseWretchAction
ChooseSamuraiAction
ChooseBanditAction
ChooseAstrologerAction
ChooseRandomClass
Player
attackactions
AttackAction
AreaAttackAction
StatusEffectAttackAction
behaviours
AttackBehaviour
FollowBehaviour
WanderBehaviour
<<interface>>
Behaviours