# Lab03Group03 A3 Diagram and Design Rationale

# Contribution log:

https://docs.google.com/spreadsheets/d/1itMLRX1DOQDyBiHIsicFYc-CcjWs0eaZE00CWoSnUc8/edit?usp=sharing

**If the diagrams are hard to see, we also added copies of the diagram in the docs folder of our submission**

# Requirement 1

The following feedbacks were given to us for the last assignment regarding this requirement

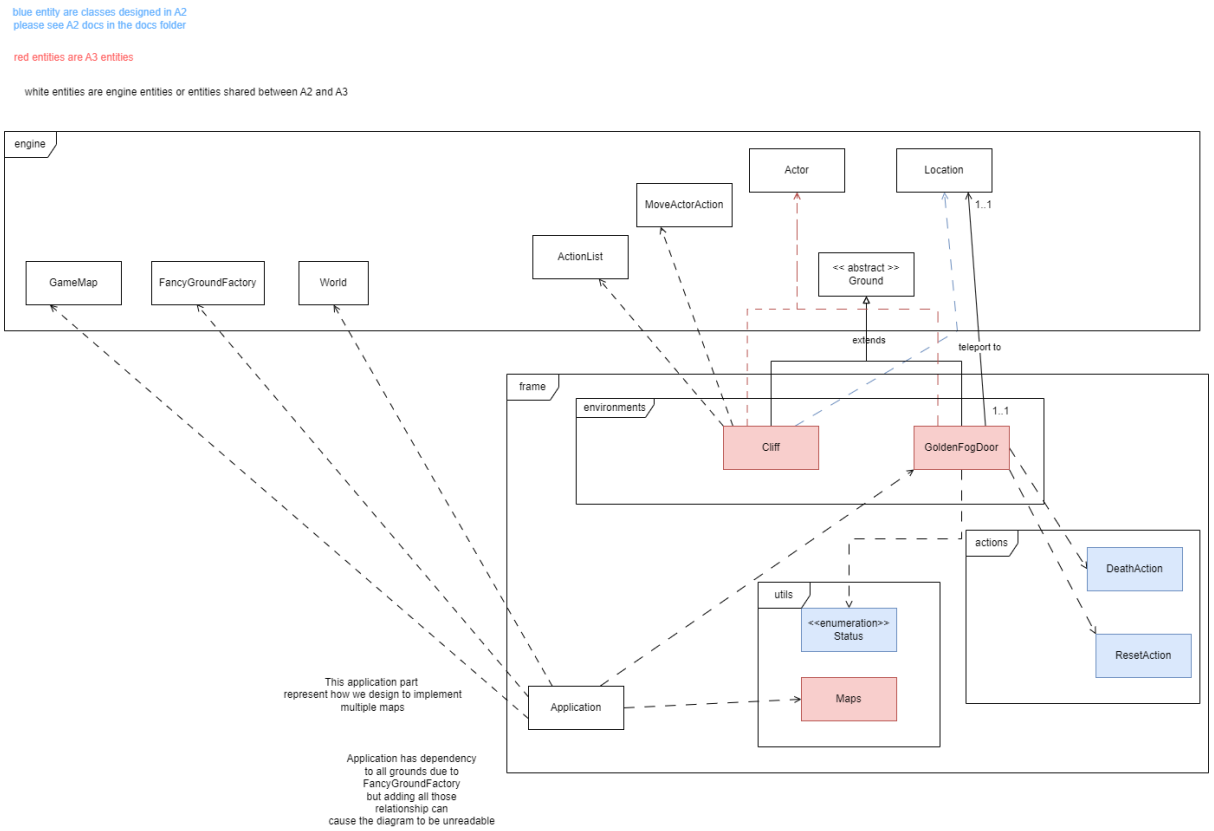
+ The rune should give to the player directly when player kill the enemy.
  When player died, the rune should be dropped to the previous location.
+ The spawning enemy function could be approached by using abstraction(DRY).
+ Some of the enemy behaviours (Follow) could be implemented in the Enemy abstract class(DRY. For pile of bone just clear all the behaviour).
+ Despawn behaviour is fine, but a little bit overdesign, could just use the playturn to check the status of enemy first then follow the original behaviours

For the first feedback, we had implemented this feature in A2, the rune is directly transferred into the player **if the player kills an enemy,** runes however will drop on the ground if an enemy is killed by another enemy. We implemented it like this due to ambiguity in the specification, which doesn't state what happens to enemy runes when killed by an enemy, in this version, we have removed that feature

For the second feedback, as the new ground introduced in A3 doesn't share the same East West spawning functionality, introducing another common abstract superclass can introduce extra complexity that is unnecessary.

For the third feedback, since multiple enemies still share the same behaviours as the enemy in assignment 2, we have reimplemented the Follow and common behaviours to be in the abstract Enemy superclass, reducing much bulk from the child enemy classes. For classes that don't share the common behaviour, we have implemented a clearBehaviour() method that the developer can use to set up a custom behaviours chain, like in Ally and PileOfBone. This however does introduce some slight connascence of execution as clearBehaviour must be called before setting the new behaviours, but the benefit outweighs the cost, as it is only one added line, and for that we reduce significant bulk from multiple enemy classes

For the fourth feedback, since the DespawnBehaviour was working fine and all team members found it easy to understand, we decided to keep our approach from A2, since reimplementing something we already understood takes extra time and mental capacity to worry about for little benefit.




As for A3 requirement 1:
For the Cliff, to adhere to DRY, it extends from the Ground class due to similar functionalities, in the tick methods, every turn it checks to see if any enemies is standing on it, if there is and it is a player, the player is instantly kill by applying Integer.MAX_VALUE damage using hurt(), this guarantees the player always dies because they can't have health higher than Integer.MAX_VALUE. We need to do this for the player before executing a ResetAction because the conscious state of the player decides some behaviour during reset (as specified in Assignment 2). If an enemy is standing on cliff however, a DeathAction is execute to kill it off

For GoldenFogDoor, similar to above, GoldenFogDoor extends from Ground and extends some functionality from its superclass to comply with DRY. Since GoldenFogDoor is a part of the map and the location of teleportation doesn't change, we have teleportTo as a private Location attribute that doesn't change in the GoldenFogDoor class. In the allowableActions() method, it allows the player to interact with it, if the player chose to do so, it will use the MoveActorAction given by the engine to move the player to the location of teleportTo.
Since the location of the doors themselves doesn't change and to prevent the main() method in Application from being too long and becoming a god class (a code smells). We've moved the setting up of maps for the game to a separate method called setUpMaps(). Here the maps themselves are created from stored strings array (from utils.Maps) and things like traders are added. The doors, their location and the location they teleport to is also set up here as they are not expected to change during the gameplay.

Optional fast travel requirement was not implemented.

# Requirement 2

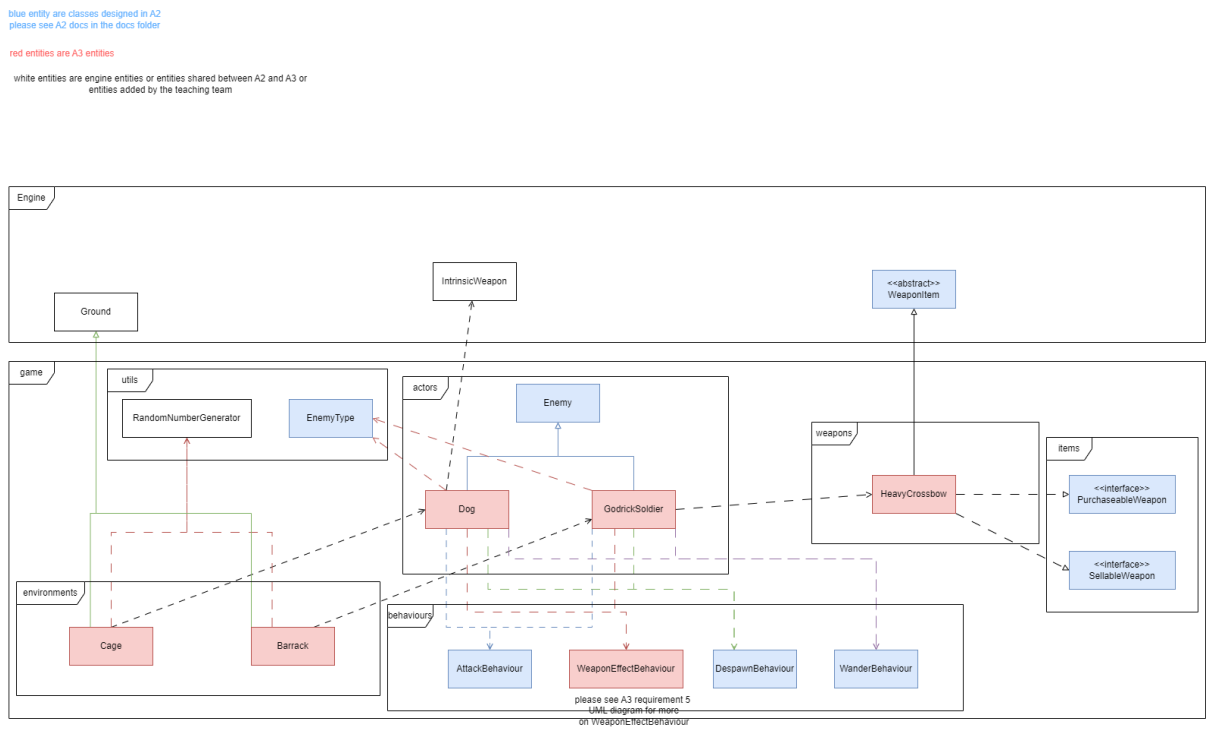
The following feedbacks were given to us for the last assignment regarding this requirement

+ In trader user compare string to check item is not a good design.
+ The rune system for the user could use a general global rune manager to do(Currently have a lot runes' object in player's inventory is not good)

For the first feedback, even with extra weapons introduced in Assignment 3, there has been no items with identical names, meaning that our current approach to checking the identity of items during trades are using name-comparison. Similar to REQ 1 fixes, we decided to use our old approach due to the time constraints. Although we also planned 2 different approaches to our current one.

First let's go over the advantages and disadvantages of our current approach, with the main benefit being its simplicity and that all team members are well aware and well-understood the implementation, and the main drawback being that the item does not ensure true uniqueness so that 2 item with identical name can be mistaken for each other but no error are raised.

In my opinion, the drawback of our approach is not substantial, as two items with different identities but having the identical name doesn't make sense by intuition and from a game development perspective. Additionally, the uniqueness and identity of an item can be protected by having team members be aware of the design rules we have in place through good communication (which we do). Lastly, and most importantly, the biggest concern of not throwing an exception when an anomalous trade (not failing fast) is made can be mitigated by good and thorough play testing (which we try our best to do).




Now our 2 alternative approach are:

- Our first and most compelling approach is to have a new enumeration which has unique cases for all item types (similar to EnemyType). All item types can only have one of this enum as a capability, then we check for identity using the hasCapability method provided by the engine. We think that this is the ideal approach if we were to implement this again as an item's identity is still protected even if a developer modifies the toString() method to deliberately cause the current approach to fail.

- Our second approach is to have all items contain an attribute which has its unique identifier as a string, this protects identity better than our current approach as we can intuitively know that IDs are unique through explicitly naming our attribute. However, this involves a large amount of refactoring and is a major time sink with only are marginal improvements

As for the second feedback, reworking our entire rune system would take too much time to gain a non-noticeable performance increase. Our current rune systems is extremely intertwined with rune-dropping, trading, golden rune, etc.

As for Assignment 3 requirements,
Cage and Barrack is similar to the last assignments environment except without East West Spawning, they extend from Ground class to enforce DRY. The tick() method has a chance (corresponding to specification) to spawn their enemy type.
An alternative approach could be to create an abstract GroundSpawn class for grounds that spawn enemies like stated in requirement 1 feedback. That might adheres better to the DRY principle, but with how differently each ground behaves (i.e some has East West Spawning, and spawns different things on different sides, some only spawns one type of enemies, etc) the abstract class could become too complex or to catered towards one use case to gain any real benefit from DRY. Our approach to let a Ground fully determine its tick behaviour is most flexible, simple and reusable even though it can slightly violate DRY.

As for Dog and GodrickSoldier, even though they are of different type, they are not hostile towards each other, because of assumption made during assignment 2 where different type must attack each other, we have reworked our AttackBehaviour to both work with these 2 new enemies (and Ally and Invader, see requirement 4) and past enemies.
We modified how the actor determines which actor is considered hostile by having a new attribute called friendlyTo, which is a list containing capabilities that the current actor is friendly to. All old enemies have an empty list, meaning that all enemyType that are not their own are considered hostile. But Dog has EnemyType.STORMVEIL_SOLDIER (GodrickSoldier's enemy type) capability in their friendlyTo list in AttackBehaviour, so they won't attack the GodrickSoilder.
A drawback with our approach is a slight connascence of execution, where the user must add to the friendlyTo list after creating the AttackBehaviour. But the AttackBehaviour would still works even if the friendlyTo list is empty (attacks all type except for it's own)
Dog and GodrickSoldier other implementation is similar to the ones in A2, they extends from Enemy abstract class to adhere to DRY and have the common behaviours except for the modified AttackBehaviour

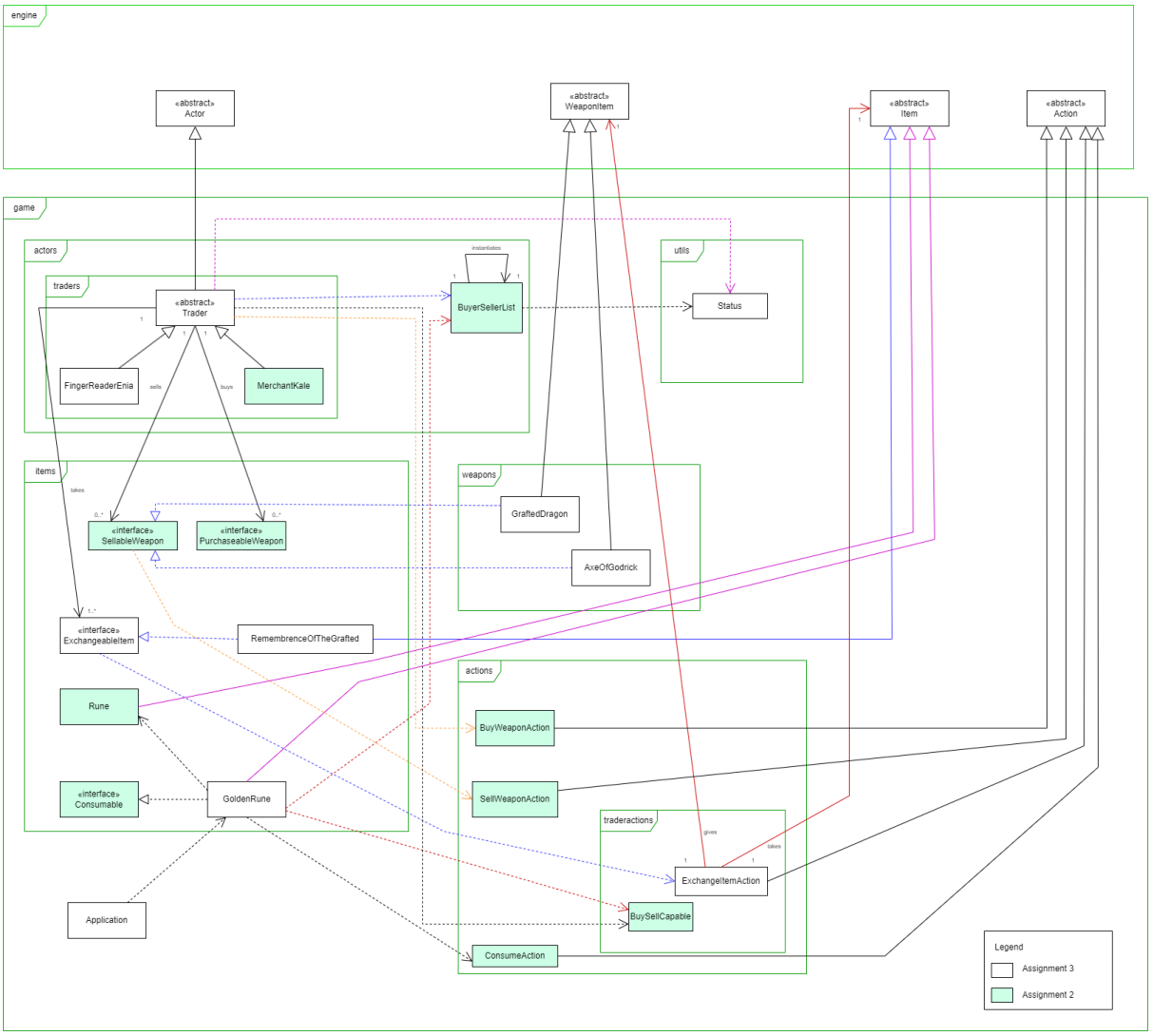
# Requirement 3

The following feedbacks were given to us for the last assignment regarding this requirement

+ The rune of play should dropped to the previous location when player is dead.
+ When player consumed all the flask, they should still been able to see the consume action, so just let the consume action decide the usage and the execution

For the first feedback, we have implemented this change as it is needed for Cliff to behave correctly, in our current implementation, the player has an attribute which stores the location they were at the last turn. When the player dies, the runes will be dropped at this location instead of the player's current location, like in the last implementation.

For the second feedback, we believe that having an option to consume when you have no uses left can be confusing and unintuitive, as it doesn't make sense to be able to choose to consume something that you don't have, then wasting a turn. And having ConsumeAction being responsible for handling the uses left of an Consumable might violate SRP as we believe that the ConsumeAction should just be responsible for applying an effect to the player when consumed.



Optional requirements were not implemented for this requirement. A new Trader Finger Reader Enia has to be added. The existing trader Merchant Kale from previous assignment can also allow the player to sell weapons like the Finger Reader Enia. Since there are similarities between the 2 traders, an abstract class Trader inherits from Actor. This helps the traders to share common code reducing code repetition adhering to DRY principle. Moreover the benefit of making the Trader abstract allows more traders to be added in future with specific implementations. The Trader abstract class has a dependency with Status. This helps it identify if the nearby actor is a Player and should be allowed to perform trade actions or not. The Trader abstract class has an association with SellableWeapon, PurchaseableWeapon and ExchangeableItem. This allows the Trader to maintain a list of exchangeable items and tradable weapons. GraftedDragon and AxeOfGodrick weapons inherit from WeaponItem abstract class to reduce code repetition because they have the same properties as WeaponItem. GraftedDragon and AxeOfGodrick weapons implement SellableWeapon interface because they are allowed to be sold to traders and this helps adhere to LSP as they can be added to the Trader without causing error when methods related to selling weapons are invoked on them. The trader inheriting from Trader class just needs to use the protected getters to add their own allowed exchangeable items and tradable weapons and the playturn inherited from Trader will ensure the correct options to Player. This way reduces code repetition and increases extensibility as more traders can be added. This way might not be good because each trader must maintain their own list of allowed exchangeable items and tradable weapons which increases memory usage. The Trader depends on BuyWeaponAction to be able to allow an actor to purchase weapons from it.

The ExchangeableItem interface is implemented by items that can be exchanged for weapons with FingerReaderEnia. The Exchangeable has a dependency on GraftedDragon and AxeOfGodrick weapons to allow the exchangeable item to be exchanged to these weapons. ExchangeableItem interface has a dependency on RemembranceOfTheGrafted is exchangeable and it implements the ExchangeableItem interface. To allow the exchange of exchangeable items for weapons, ExchangeItemAction inheriting from Action was created. ExchangeableItem interface has a dependency with ExchangeItemAction, this allows the ExchangeableItem to return a list of exchange actions for each weapon that it can be exchanged to. Moreover, exchangeable items implementing the ExchangeableItem interface adhere to Liskov Substitution Principle (LSP). This is because exchangeable items are stored by the trader and the trader expects to receive exchange actions by invoking a method on them to allow the player to perform an exchange. By implementing ExchangeItem interface for exchangeable items only the items implementing the interface can be stored by the Trader and the getExchangeItemAction can be invoked on them without any error to get the exchange actions. This approach is good because if more items are exchangeable in future they can implement the ExchangeableItem interface and be added to the traders without modifying existing code. The getExchangeItemAction can be overridden by exchangeable items in future if any new exchangeable item should be exchanged for certain weapons only. However, if exchangeable items can be exchanged for something other than weapons like items then new actions are needed because the current implementation only allows items to be exchanged for weapons.

The GoldenRune class inherits the Item abstract class and implements the Consumable interface. The Application class depends on the GoldenRune class to be able to scatter the

Golden Runes randomly across the map at each game. The GoldenRune class depends on the Rune class to be able to generate a random amount of Runes to be added to the Player consuming the GoldenRune. The GoldenRune implements the Consumable interface and depends on ConsumeAction to be able to be consumed by the Player. This adheres to LSP because the ConsumeAction can invoke methods on consumable items without getting an error and receive expected output to allow the actor performing consume action to consume the item correctly. This is achieved by making ConsumeAction only accept items that implement the Consumable interface. This implementation is good because in the future if more items need to be consumable, they just have to implement a Consumable interface and the existing ConsumeAction will allow it to be consumed by an actor greatly improving extensibility of the feature and maintainability.

# Requirement 4

To understand the design implementations for Assignment 3, please revisit the Assignment 2 rationale, which states the following.

“The requirement 4 UML diagram represents the implementation of the combat archetypes and the associated methods and skills that are associated with those archetypes. The archetypes determine the starting hitpoint of an actor, in this case the player, as well as the starting weapon and the skill associated. This diagram reflects the implementation of the code in the game, and the following rationale will discuss the implementation and changes made to implement the requirement.

Firstly, following the comments of the previous design feedback, we implemented an abstract class called Archetype, from which we can extend different combat archetypes classes from and initialise their respective hit points and weapon. In this case, we have the Bandit, Wretch and Samurai class all extending from Archetype. While the previous design had ArchetypeManager storing the different combat archetypes attributes, this design separated the combat archetypes. The benefits from this is that the archetype is much more open to extension, particularly if we want archetypes to be accessible for other actors. If we used the old design, this extension would violate the Open-Closed Principle, because we would need to directly modify ArchetypeManager to access the archetypes. Additionally, the sub-classes extending from Archetype all have an association with a weapon that corresponds to that archetype, and they also hold the hit point which the actor would have, if that combat archetype was selected. For example, Bandit uses a GreatKnife.

Moving on, the ArchetypeManager still has the task of setting up the Player through the createPlayer() method. However, instead of storing the combat archetype details, the ArchetypeManager has an association with each combat archetype, creating an instance of these archetypes and storing them in a dictionary for use. The ArchetypeManager class also has another method called selectArchetype() which requires UI to return the selected archetype in the form of a key. This key is used by createPlayer() to access the instance of the combat archetype that can be used to make the player.

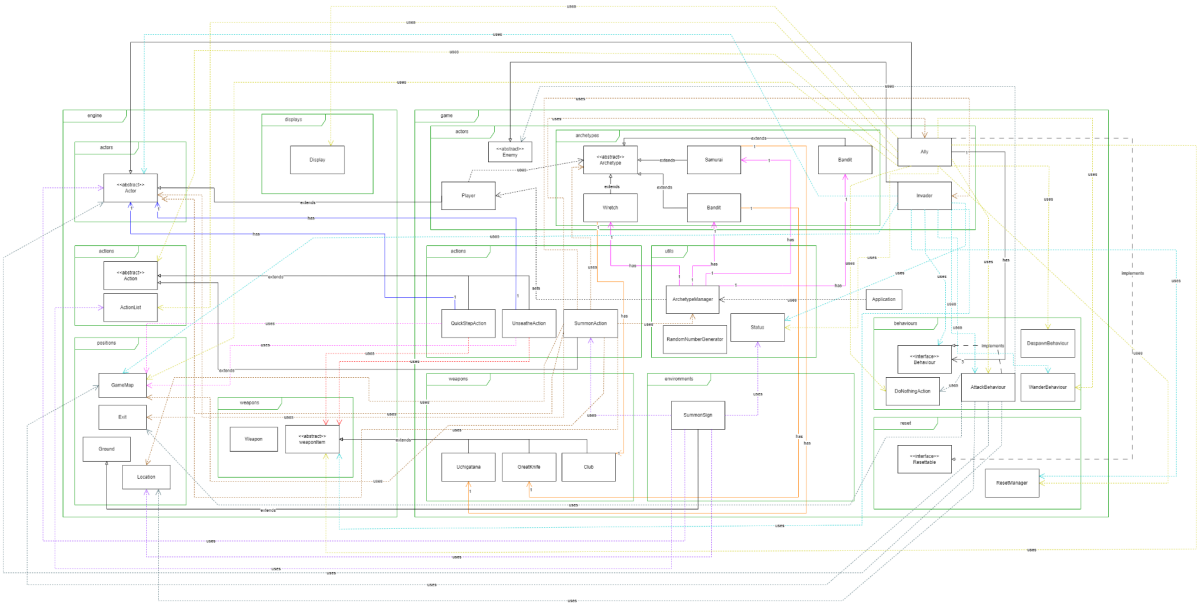
From there, the Application class can call on the ArchetypeManager to run and set up the Player instead of initialising the Player in the Application class. This follows the Single Responsibility Principle (SRP) whereby the Application class should focus on running the main code for the world, instead of also having to determine the Player's archetype.

Moreover, the Player constructor is modified to include the archetype, where the player's hitpoint and weapon can be accessible from the input archetype. Hence, the Player has a dependency on Archetype class.

The Player will be directly extended from the abstract class Actor (from the engine) as it has the similar features and attributes. This will reduce repetition of re-coding the attributes and methods (DRY).

The special skills, i.e. QuickStepAction and Unseathe Action, which are associated with the weapons of the unique combat archetype, will be extended from Action (DRY). These skills will be accessible to the player via the getSkill() implementation within the weapon. These actions also rely upon the GameMap, to identify the target, movement, attack, etc. Also each action has an associated Actor that represents the target.

The unique weapon associated with each archetype, i.e. Uchigatana, GreatKnife and Club, will be extended from the abstract Weapon class (DRY), and the special skill associated with these weapons will be implemented."



Moving on to the requirements for Assignment 3, we have the following.

For the astrologer, we have created a new subclass of Archetype which has the stats of the Astrologer. Similarly to the last assignment, if the player chooses this archetype at the start of the game, a Player object will be instantiated with the data retrieved from the Astrologer archetype class.

For the SummonSign, allowableActions method gives the Player a choice to execute a SummonAction() when standing near the SummonSign. The SummonAction() picks a random archetype (using a method from ArchetypeManger) then applies their stat to the spawned Ally or Invader if there is an unoccupied exit. If there is no valid exit to spawn then the spawning will fail and display a message.

For Ally and Invader, Invader will extend from Enemy to enforce DRY and has all the common behaviours. But since Ally doesn't attack the player but does attack the other enemies, we didn't extend it from Enemy but we do have a HashMap that works like the one in Enemy. The Ally has the AttackBehaviour like the Enemy, but it has Status.HOSTILE_TO_ENEMY in its friendlyTo list so the Ally won't attack the Player.
We could have created an abstract class FriendlyActor and have Ally extend from it in order to account for having many different types of actors that are friendly to the player in the future. But in this assignment, only Ally can move around, attack and is friendly to the Player, so having an abstract that has almost identical functions and only 1 class extending from it can violate SRP.

In order to let Ally and Invader spawn with a random archetype. We have added extra functionality to our ArchetypeManager, being the randomArchetype method. This method is used in SummonAction to randomly get an instance of a random Archetype subclass which is then used to get data to instantiate Ally or Invader. This works because Ally and Invader differ from other Actors being that they have a similar constructor to Player where their health and starting weapon can be chosen upon instantiation, the only thing needed to give an Ally or Invader their archetype is to set these constructor argument to the weapon and health of the Archetype we want upon instantiation.

In our UML diagram, we show all the dependencies associated with the implementation of the new functionalities required for Assignment 3. While the UML diagram is chaotic, you should focus on the colour code of the dependencies at each class to understand the interaction of classes involved to implement the new functionality.

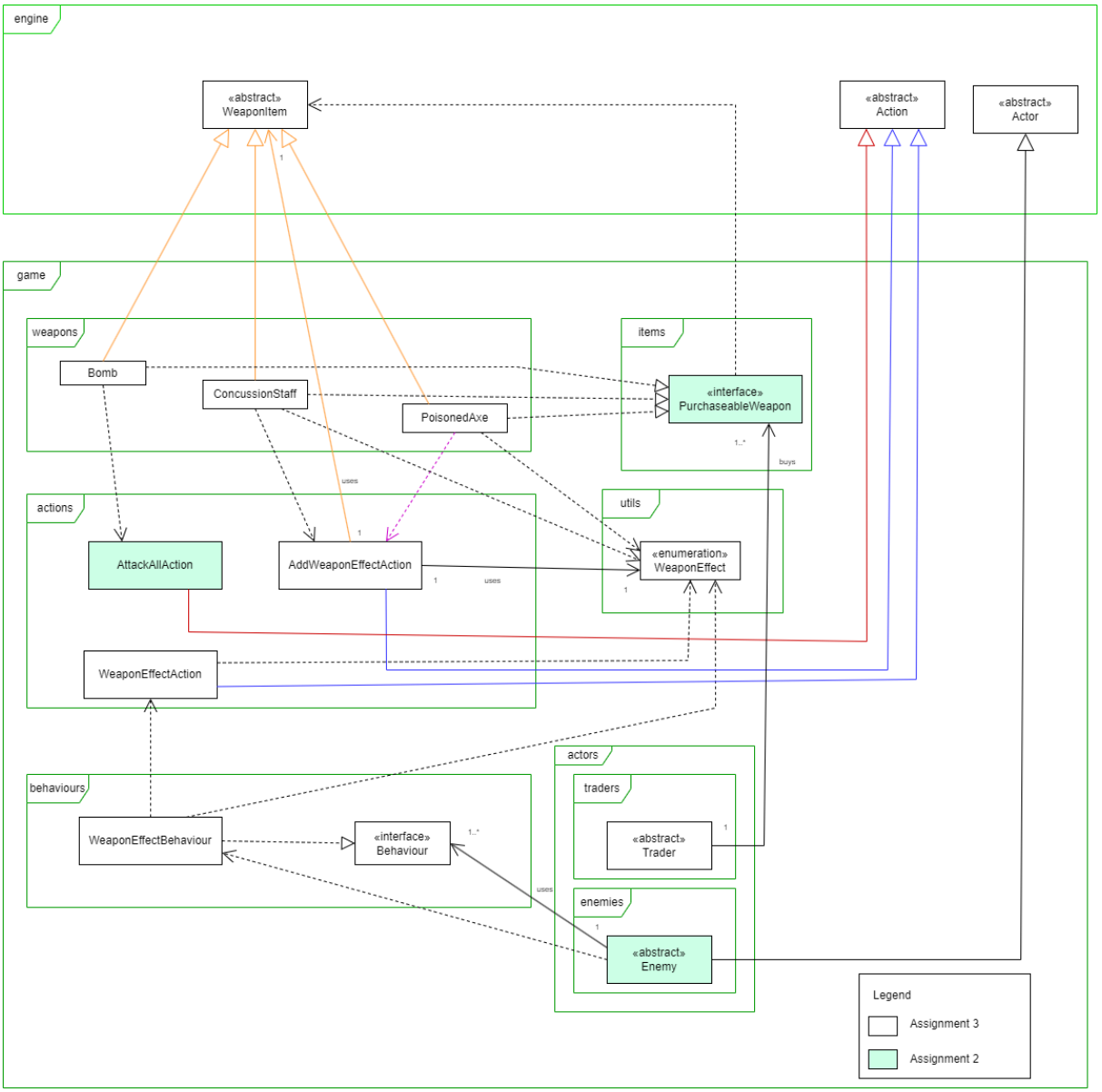
# Requirement 5 (Powerful Weapons)

The following feedbacks were given to us for the last assignment regarding this requirement

+ Some enemies have similarities should be better have same sub-abstact class(Like skeletonBandit and swordsman).
+ The method to spawn enemy is a bit hard to extend

Regarding the first feedback, we believe that adding a sub-abstract for enemy types wouldn't be an optimal design choice in our case. Currently, the difference between the enemy types (i.e Skeleton, Crabs, Wolfs ) aren't major enough to warrant for a sub abstract from Enemy. For example, the only difference between Crabs and Wolfs is that Crabs can slam, and that get taken care of by AttackBehaviour anyways. Adding a sub abstract class can make the code more complex than it needs to be. We do agree that adding a sub abstract will helps with adding more enemies of the same kind in the future, but giving the time constraint of this assignment, implementing it wouldn't be practical.

Regarding the second feedback, similar to the first one, in A3 and our creative requirement, there isn't a need to further subdivide the map, so these behaviours are unique to the 3 environments in A1. And given time constraints, it wouldn't be practical to implement them




For the creative requirement the title would be Powerful Weapons. For Powerful Weapons 3 new weapons were created. Firstly, the ConcussionStaff is a weapon with a special skill where it could cause the target being attacked faint for 3 turns. When the target has fainted for 3 turns the target will not be able to perform any action for the 3 turns. Secondly, the PoisonedAxe is a weapon with a special skill where it could cause the target being attacked to be poisoned for 3 turns. When the target is poisoned for 3 turns where the target will be hurt with 50HP at each turn the target will not be able to perform any action for 3 turns. Thirdly, the Bomb Weapon has a default skill that kills any instantly with a 100% hit rate. The target will receive full damage to its hp to be instantly killed. The ConcussionStaff and PoisonedAxe special skills only work on enemies and invaders. The Bomb also has a special skill where it can be used to instantly kill all targets in its surrounding regardless of

type, similar to the default skill of Bomb but it kills all the surrounding targets. Moreover, to make sure this weapon does not make the game easy, this weapon can only be used once and it is automatically removed from the actor's inventory. The 3 weapons created are very powerful, so they are not available to be collected by the Player, they need to be purchased from MerchantKale trader at a very high Rune price.

The powerful weapons inherit from the WeaponItem abstract class since they share the same properties and this helps reduce code repetition adhering to DRY. By making use of inheritance each class deals with a single weapon ensuring a single responsibility for each of the 3 new classes representing 3 powerful weapons and this adheres to Single Responsibility Principle (SRP). These weapons are purchasable, hence they implement the PurchaseableWeapon interface. By doing so these weapons will have relevant methods to allow them to be purchased by the Player and these methods can be invoked on them without any error. The trader stores PurchaseableWeapon and these weapons implement the interface allowing them to be stored in the trader without any error. The methods related to purchasing can be invoked with expected execution and this adheres to the Liskov Substitution Principle (LSP). The Bomb weapon has a dependency on AttackAllAction because its special skill has similar logic to it and this helps reduce code repetition adhering to DRY. The weapons ConcussionStaff and PoisonedAxe have a dependency on WeaponEffect enumeration to be able to add them as capability to their targets when attacked so that the weapon effects can be applied on them for 3 turns. This way of design makes it easier to add more new powerful weapons in the future.

The AddWeaponEffectAction and WeaponEffectAction inherit from the Action abstract class as they have the same characteristics of an Action and are required for ConcussionStaff and PoisonedAxe to work correctly. The AddWeaponEffectAction has an association with WeaponItem and WeaponEffect enumeration to allow it to store the weapon used for the attack and the effect of the special skill on its target. The PoisonedAxe and ConcussionStaff depend on the AddWeaponEffectAction to execute its special skill when used. This helps adhere to SRP because multiple responsibilities required for this feature are stored in different classes representing one responsibility each making the code easier to maintain and modify. However, this can make the code complicated because there are many classes. The WeaponEffectAction depends on WeaponEffect enumeration to check the WeaponEffect capability added to the target via AddWeaponEffectAction to determine the special skill such as faint or poison to be performed on the target.

To ensure that these special skill weapons affect enemies and invaders WeaponEffectBehaviour was created. The WeaponEffectBehaviour implements Behaviour interface because this behaviour must be executed without user intervention and the Enemy abstract class has an association with Behaviour. This allows the Behaviour to be stored in enemies and invaders and the methods for the behaviour can be executed without any error to apply the weapon effect on them for 3 turns only. This adheres to Liskov Substitution Principle (LSP). The WeaponEffectBehaviour depends on the WeaponEffect enumeration to be able to determine whether the target should faint or be poisoned. The WeaponEffectBehaviour depends on WeaponEffectAction to be able to return the Action for executing the effect of the weapon special skill on the target. This design makes it extensible if more weapons need to be added in the future with special skills having weapon effects.