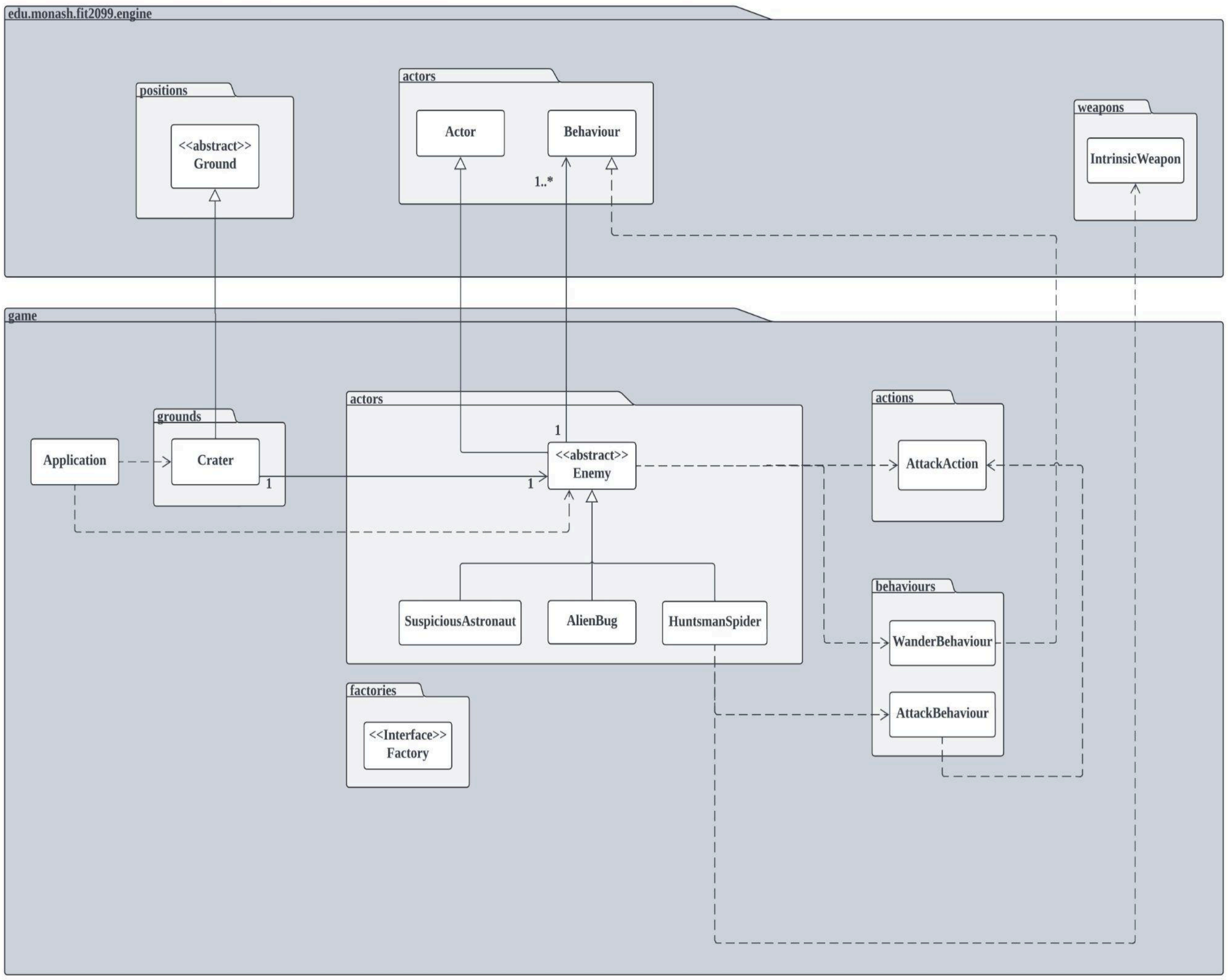
Assignment 2 : Design Rationale

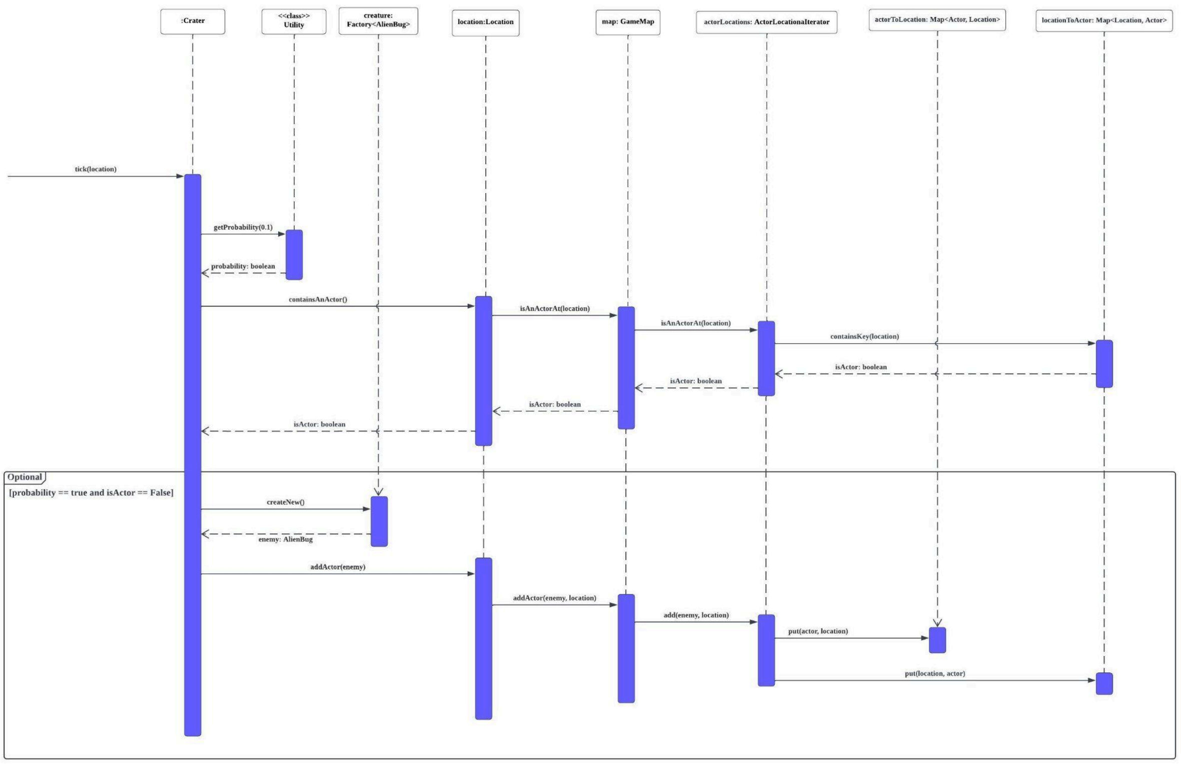
REQ1: The moon’s (hostile) fauna II: The moon strikes back

UML Diagram



Sequential Diagram

Chosen Scenario : Crater spawns an AlienBug



Changes Made to Design Proposed in Assignment 1

For Assignment 2, we have removed the usage of ActorInjector interface due to there will be many corresponding injectors being created with the increased amount of actors being introduced to the game. To replace this implementation, we have designed a Factory interface. With this implementation, we do not need to create an injector for each actor being introduced as the createNew() method in Factory accepts no parameter and returns an object of type A (Initialised based on its usage). Because of that, Crater class now does not accept an ActorInjector object at its constructor but an object of Factory<Enemy>. When we want to spawn a new enemy, we can just call its createNew() method and an object which is the same as the enemy we accept at constructor. Besides, we have cancelled out the implements of Crater on Spawnable interface as we found redundancy on it with the Factory way of implementation above. Therefore, we have moved the spawning of new creatures to tick() method instead of overriding the spawn() method from Spawnable interface.

Design Goal :

In REQ1, we are tasked to create 2 more actors, AlienBug and SuspiciousAstronaut. Besides, we need to create a crater which can spawn more kinds of creatures: AlienBug, and SuspiciousAstronaut, while following the SOLID principles

Design Decision :

AlienBug class has been created to represent AlienBug which is a newly added actor in the game. SuspiciousAstronaut class has been created to represent Suspicious Astronaut actors. Factory is created to spawn new Actor object of the same type as the Actor object accepted by the constructor.

Final Design :

The SOLID principles used for the code design are Single Responsibility Principle (SRP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP) and Don’t Repeat Yourself. (DRY)

The AlienBug and SuspiciousAstronaut have been designed as concrete class and extend Enemy abstract class. AlienBug and SuspiciousAstronaut share common attributes and methods with Enemy, thus extending from Enemy helps to reduce redundancy of codes (DRY). Besides, instead of combining them into a ‘god’ class, we separate them into respective classes as each class should only have a single responsibility. (SRP)

The constructor of Crater accepts Factory<Enemy> object instead of a specific Enemy subclass. Therefore, it can be used to spawn any object of the subclass of Enemy by just substituting it without any modification.(LSP) The Crater class does not have an association with concrete creature classes, but on the abstraction provided by the Factory interface. (DIP)

Advantage of this design is that by extending from the Enemy abstract class, we will not have repetition in codes for AlienBug and SuspiciousAstronaut. Besides, Crater can be used to spawn different creatures without modification as it has an association with Factory<Enemy> objects. Furthermore, we do not need to create multiple injector classes for each actor introduced to the game which has highly increased our codes’ readability.

Disadvantages is that Crater can only spawn objects which extends Enemy abstract class. This means that if there are some other actors which do not extend Enemy abstract class and need to be spawned, Crater will not be able to handle this.

Alternative Design :

Instead of extending AlienBug concrete class and SuspiciousAstronaut concrete class from Enemy abstract class, we can opt to not extend them from Enemy abstract class but just create a concrete class for them. To be spawned by crater, we can create a crater for each of them by modifying the constructor of their respective crater to accept an object of them instead of a general Enemy object from above.

Analysis of Alternative Design :

The alternate design above is not ideal because it violates various Design and SOLID principles :

1. Don’t Repeat Yourself (DRY)
    - Since we are not extending from Enemy class, we will need to rewrite many similar things every time we create a new class for an actor which is similar to Enemy
    - As we will create a crater for each actor which can be spawned, we will repetitively create many similar crater just to achieve the goal of spawning a specific actor

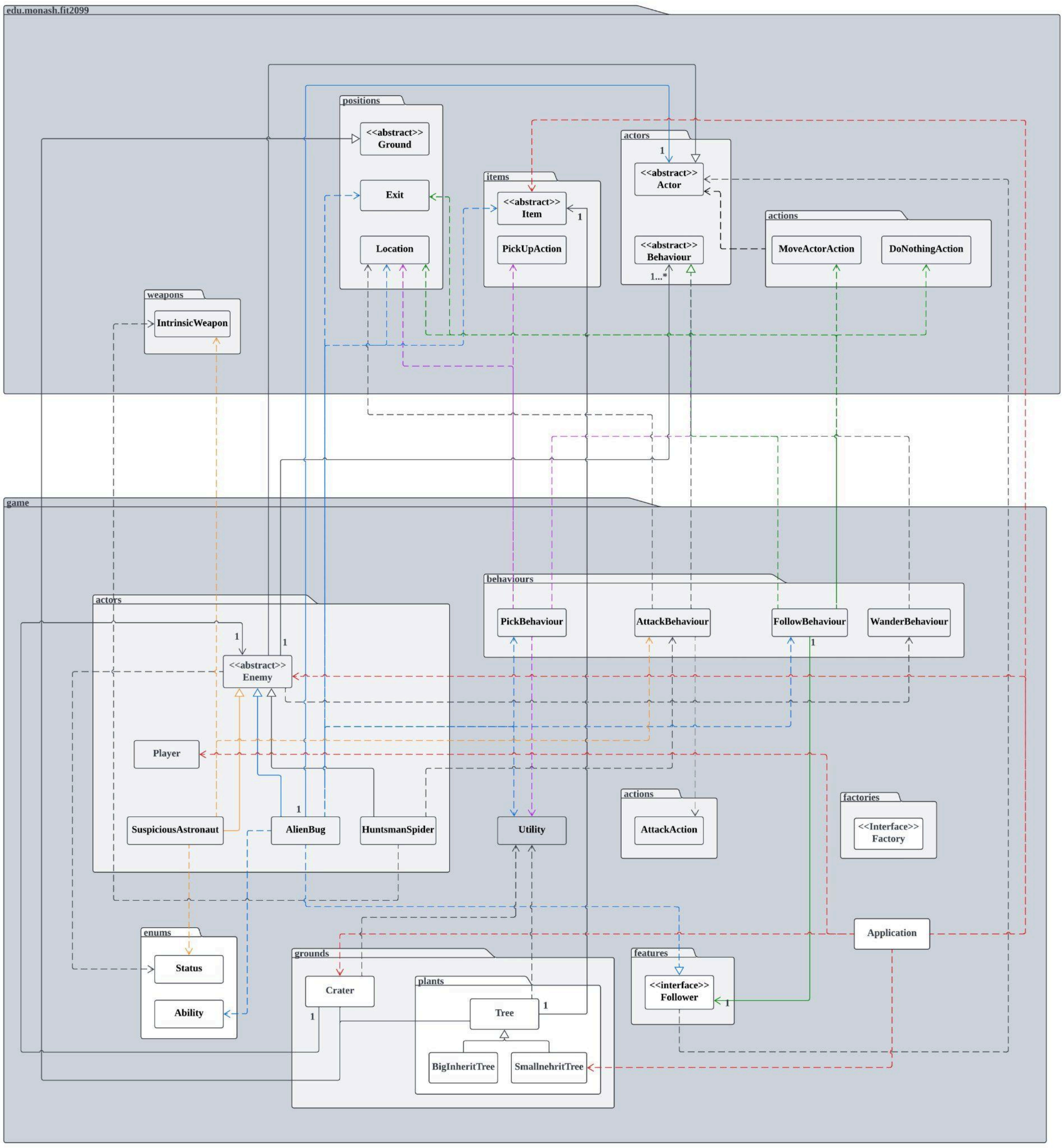
Conclusion :

The reason for choosing this design is that we can ensure each class has only a single responsibility and enhance the code maintainability when we are required to add new methods for respective classes in the future. Besides, we can also ensure that we do not have repetitive code by extending AlienBug and SuspiciousAstronaut from the Enemy abstract class. Furthermore, we choose to let Crater accept an Factory<Enemy> object at its constructor instead of a specific Enemy subclass object to ensure we adhere to DIP and most importantly we do not need to create a crater for each actor which will be spawned.

For future extension, if we need to add some new methods for AlienBug or SuspiciousAstronaut, it can be implemented by adding these new methods at their corresponding classes without violating any SOLID principles. Besides, when we want Crater to spawn a new type of enemy, Crater can make this happen as it has an association with a Factory<Enemy> object and the new type of enemy can be spawned. As for the disadvantage discussed above, when there is a new type of actor which does not extend Enemy abstract class but needs to be spawned, it is totally solvable by just creating a new Crater which accepts Factory<new class> at its constructor.

<u>REQ2: The imposter among us</u>

UML Diagram

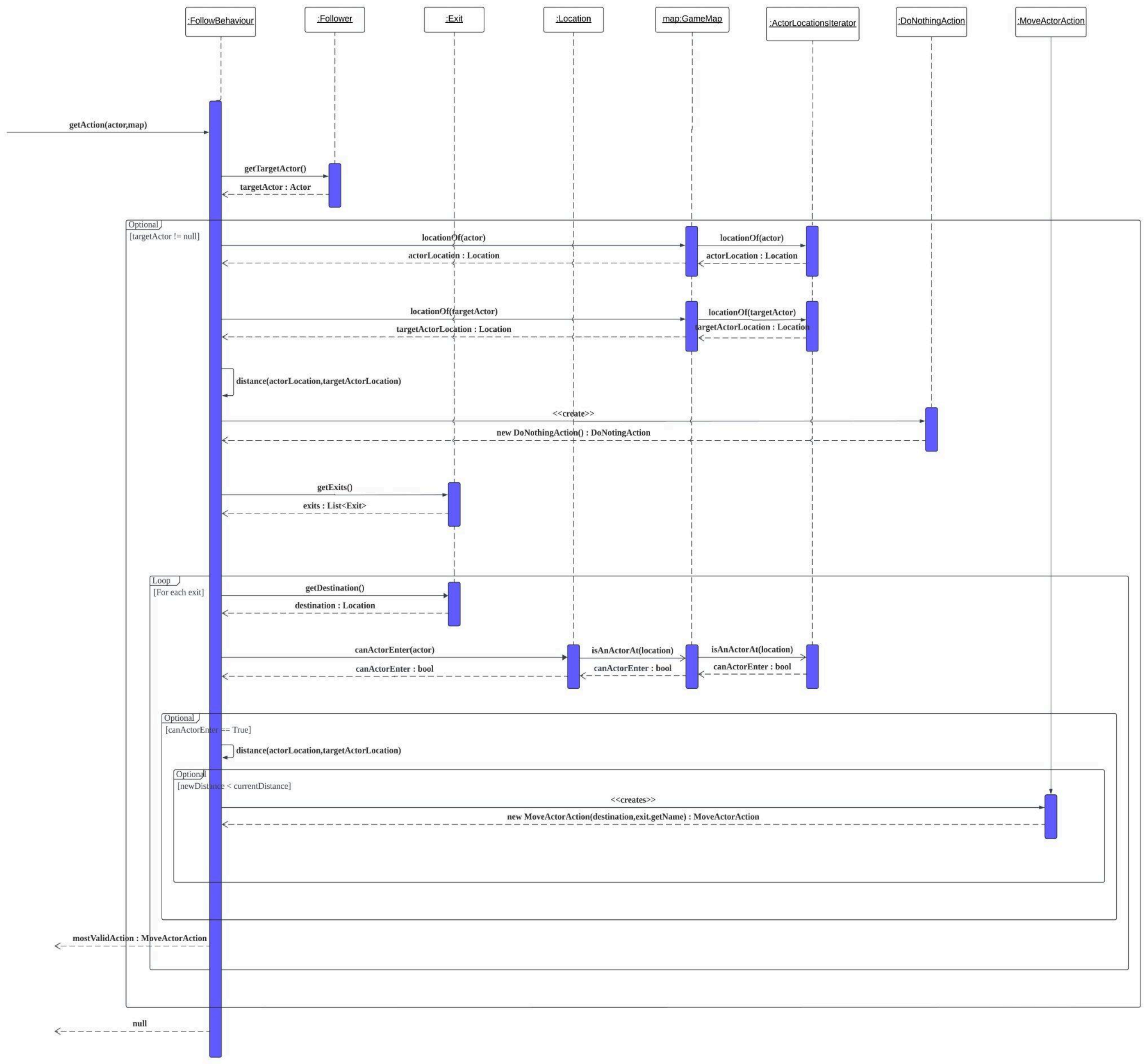


Justification for UML Diagram :

Since all types of behaviours implement Behaviour interface and Enemy abstract class has an association with Behaviour interface, thus we do not need any dependency from AlienBug, SuspiciousAstronaut, HuntsmanSpider to different types of behaviours.

Sequence Diagram

Chosen Scenario : AlienBug implements FollowBehaviour on Player



Design Goal :

For requirement two, we are tasked to allow AlienBug to follow player and steal scraps. Besides, we are tasked to allow Suspicious Astronaut to instantly kill Player once Player stands one exit away from it.

Design Decision :

FollowBehaviour is created to enable AlienBug to follow Player until it dies or Player moves to another moon. PickBehaviour is created to let AlienBug be capable of stealing items on the ground. Tree, SmallInheritTree, LargeInheritTree contributes on spawning LargeFruit and SmallFruit thus they are added to class diagram to provide details how the AlienBug can steal those items.

Final Design :

SOLID principles which have been applied are Single Responsibility Principle (SRP), Open Closed Principle (OCP), Liskov Substitution Principle (LSP), Dependency Inversion Principle (DIP) and Don't Repeat Yourself (DRY).

Follower interface has a dependency with Actor object but not specific Actor object. Thus, it can be used for different actors who want to implement FollowBehaviour on a target actor. As we want to ensure each interface has only a single responsibility and don't have the problem of combining all interfaces in one 'god' interface, we create each interface separately. (SRP, ISP)

From requirement 1, AlienBug is further extended to implement Follower interface and has an association with Actor object instead of having an association with Player object specifically. By having an association with Actor object, AlienBug is allowed to follow other actors by just further extending the getTargetActor() method from Follower interface to follow more actors based on each actor features's. (OCP, LSP, DIP) For the naming of an object of AlienBug, getRandInt() method has been created at Utility class so that when other classes need to use it, we do not need to repeat the same code in different classes. (DRY)

Behaviour interface allows for extension without any modification. Therefore, when a new behaviour is created and it implements Behaviour interface, the new behaviour works well in Enemy abstract class as we just need to call out a specific method (getAction()) to implement the action of behaviour without having to modify Enemy abstract class just to accept the new behaviour. (OCP)

PickBehaviour implements Behaviour interface. FollowBehaviour implements Behaviour interface and its constructor accepts Follower object. With this implementation, we can allow different actors to have FollowBehaviour instead of just AlienBug alone by just substituting them in the constructor without affecting the functionality (LSP). Besides, FollowBehaviour also has a dependency with Actor object instead of specific actor object. Therefore, it can be used to follow different actors instead of just a specific actor. (DIP) As both behaviours have different functionality, we separate them into two classes. (SRP). By doing so, we also do not need to have our behaviours created at each Enemy subclass (DRY).

Advantages of this design is that FollowBehaviour can be used by different actors to follow different actors. This is because FollowBehaviour has an association with Follower object and has a dependency with Actor object, thus it can be widely used by any actor to follow any actor. Besides, by separating behaviour into multiple classes, we can improve our code maintainability as if any modification needed for any behaviours, we can quickly modify it any their respective classes.

Due to AlienBug and SuspiciousAstronaut extends from Enemy abstract class which is a class that extends from Actor abstract class. Since Enemy abstract class has an association with Behaviours interface, it may be a problem when we have some errors found and may affect our implementation on behaviours.

Alternative Design :

An alternate design will cancel out the implementation of the Follower interface and not find our target actor at playTurn() method of AlienBug class. We can find our target actor at the getAction() method of FollowBehaviour. Once we find the target actor is one exit away, we start to follow it.

Analysis on Alternative Design :

The alternate design above is not ideal because it violates various Design and SOLID principles :

1. Open Closed Principle (OCP)
    - If we choose to find our target actor at FollowBehaviour, it cannot be further extended to follow other actors besides the one we are following now. This will severely violate OCP as we will need to modify FollowBehaviour to work on our new target Actor.

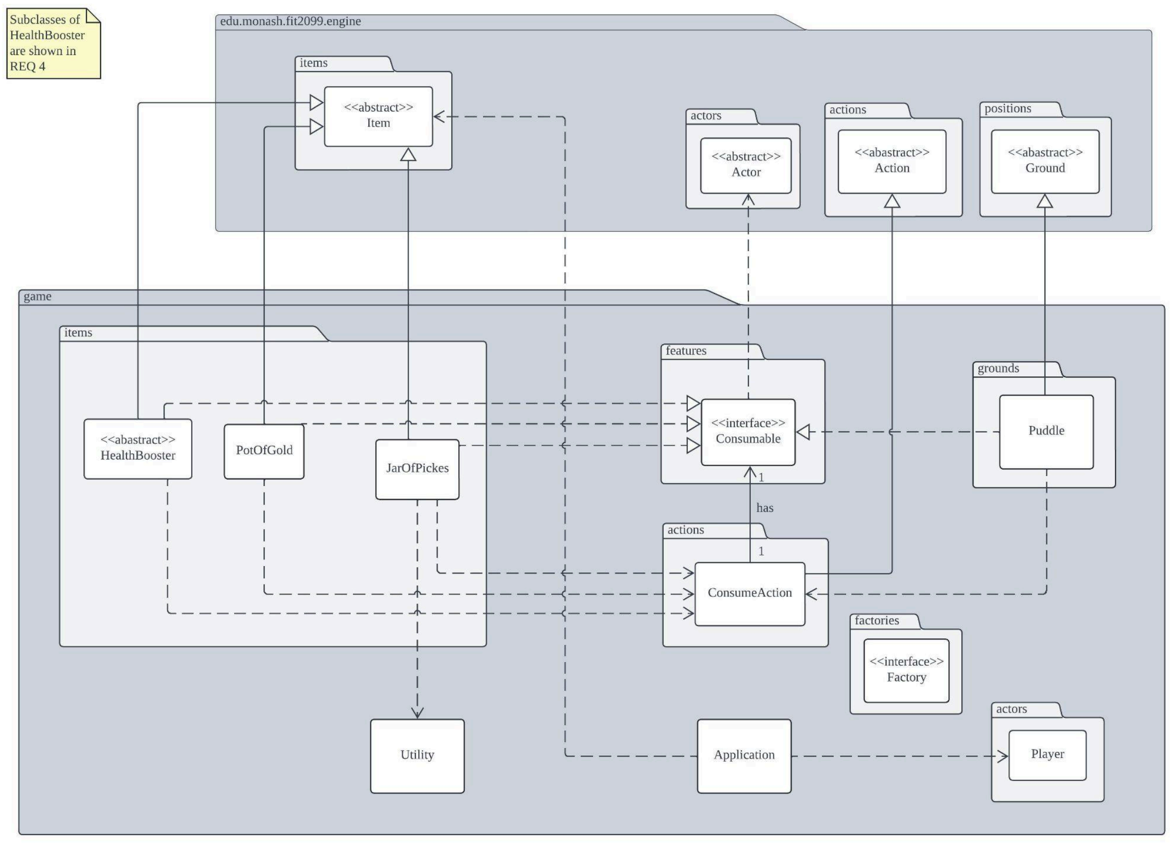
Conclusion :

Reason for choosing this design is that we can add behaviours easily and create behaviours which can be shared by all classes. If we initialise our behaviours in Enemy subclasses, we will need to create all behaviours in respective Enemy subclasses which will then violate DRY as some behaviours may be shared by different Enemy subclasses. Besides, by designing each behaviour class to implement a Behaviour interface, we can add this behaviour easily in Enemy subclasses as there is an association on Enemy and Behaviour.

When we need to add a new behaviour in the future for any subclasses of Enemy, this can be implemented easily with our code. We can create a new behaviour class which implements Behaviour interface and at the corresponding subclass of Enemy can add this behaviour by using putBehaviour() method. Extension discussed earlier clearly does not violate any SOLID principles and can be implemented. Besides, when we need to have a new actor which can also follow another actor, this can be easily achieved by using Follower interface and FollowBehaviour. Besides, when we need to follow another target actor it can be easily done by extending the getTargetActor() method to accept the feature of the new target actor.

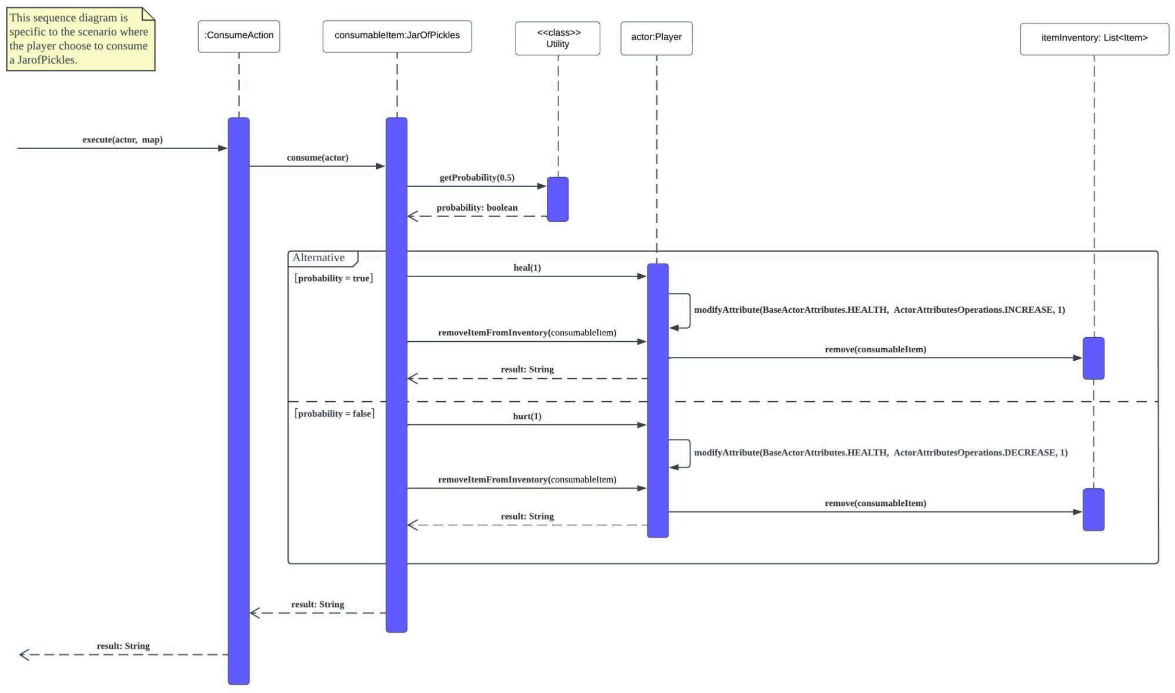
REQ3: More Scraps

UML Diagram



Sequence Diagram

Chosen Scenario : The Player choose to consume a JarofPickles


This sequence diagram is specific to the scenario where the player choose to consume a JarofPickles.
:ConsumeAction
consumableItem:JarOfPickles
<<class>> Utility
actor:Player
itemInventory: List<Item>
execute(actor, map)
consume(actor)
getProbability(0.5)
probability: boolean
Alternative
[probability = true]
heal(1)
modifyAttribute(BaseActorAttributes.HEALTH, ActorAttributesOperations.INCREASE, 1)
removeItemFromInventory(consumableItem)
remove(consumableItem)
result: String
[probability = false]
hurt(1)
modifyAttribute(BaseActorAttributes.HEALTH, ActorAttributesOperations.DECREASE, 1)
removeItemFromInventory(consumableItem)
remove(consumableItem)
result: String
result: String
result: String

Design goal:
The design goal for this requirement is to allow intern to interact with 2 new objects and 1 new modification for existing object in game , which are JarOfPickles, PotOfGold and Puddle. JarOfPickles is a new object that intern can pick up and drop off, it can heal or hurt the intern by 1 hitpoint with a 50% of chance when consumed by intern. PotOfGold is a new object that intern can pick up and drop off, intern gains 10 credits when consume PotOfGold. For the Puddle ground object, intern can now drink water from it and gain 1 maximum health point after consumed it without changing the current hitpoint.

Design Decision:
JarOfPickles extends abstract Item class in the Engine and implements Consumables interface. PotOfGold class will extend abstract Item class in the Engine and implements Consumables interface. Puddle class extends the Consumable interface. Three of the classes will implement Consumable interface to enable intern performing consume action, we will discuss about the detail of implementation below.

Final Design:
The Consumable interface is designed to provide the functionality of an object that can be consumed by an actor. It only focuses on the effect of the object when consuming the object by the actor (ISP). The advantage is that this interface can be reused by objects in the game that can be consumed by an actor. The disadvantage is that we need to provide implementation when using this interface and some implementation will be the same.

ConsumeAction is a class that extends the abstract Action class in the engine. It can replace the Action object in the engine when executing the action chosen by the actor without causing an error (LSP). ConsumeAction needs to have the Consumable to execute the function of the object that can be consumed by the actor. It depends on the Consumable interface to complete this action (DIP). The advantage is that ConsumeAction can accept any objects that implement the Consumable interface and apply its effect to the actor who consumes the object.

JarOfPickles is a class that extends abstract Item class in the Engine and implements Consumables interface. It can be picked up and be consumed by the player as the properties in Item class allow JarOfPickles to perform. This has prevented repeating implementation (DRY). JarOfPickles is only representing a jar of pickles in game (SRP). Advantage is we can simply modify one class only which is JarOfPickles if we need to implement extra requirements in the future for JarOfPickles. Disadvantage is we will need to add more classes to represent items if there are more items to be added into the game in the future.

Furthermore, JarOfPickles is using an interface Consumables from A1 to achieve increase or decrease the player's health by 1 if it is consumed (ISP). This plug-in design of interface has given us a flexibility to add extra feature on our current class. getProbability method in Utility class is used to decide whether the JarOfPickles is fresh or expired when consumed. This has prevented JarOfPickles to be a GOD class by not implementing everything inside

JarOfPickles. This has also avoided DRY in JarOfPickles such that we don’t have to implement a similar functionality here as we are using existing interface and methods. We can simply add new features and functionality for JarOfPickles by creating new classes then let JarOfPickles extends the new class without modifying the existing code inside JarOfPickles if there are future extensions. (OCP) The JarOfPickles class depends on abstractions (Item and Consumable) rather than concrete classes. This makes the JarOfPickles class more flexible and easier to maintain. (DIP)

PotOfGold is a class that extends abstract Item class in the Engine and implements Consumables interface. It can be picked up and dropped by the player as the properties in Item class allow it to perform. This has prevented repeating implementation (DRY). PotOfGold is only representing a pot of gold in game(SRP). Advantage is we can simply modify one class only which is PotOfGold if we need to implement extra requirements in the future for PotOfGold. Disadvantage is we will need to add more classes to represent items if there are more items to be added into the game in the future.

Besides, PotOfGold class is using an interface Consumables to achieve to add 10 credits into player’s wallet if user consumed it. (ISP) The PotOfGold class has implemented this small and focused rather than a huge class with extra features and method that it doesn’t need. This has also avoided DRY in JarOfPickles such that we don’t have to implement a similar functionality here as we are using existing interface and methods.We can simply add new features and functionality for PotOfGold by creating new classes then let PotOfGold extends the new class without modifying the existing code inside PotOfGold if there are future extensions. (OCP) The PotOfGold class depends on abstractions (Item and Consumable) rather than concrete classes. This makes the PotOfGold class more flexible and easier to maintain. (DIP)

Puddle class is modified as when player steps on it, the player can consume water from the puddle in game and increases its maximum hitpoint. To achieve this, Puddle class extends the Consumable interface to override the consume method. (ISP) It first check whether current location contains an actor and the actor is the same as the actor provided before returning a ConsumeAction. The benefit when we use interface in this case is that it gives us a flexibility to add in a small and focused feature, rather than a huge bunch of unnecessarily methods if we using a class.

Alternative Design:
An alternative design for this requirement is that we can straight away create JarOfPickles and PotOfGold and implement the logic of consuming inside each classes respectively. For the Puddle class, we also implement the logic inside. For example: add consume method within each classes that has a different implementation for corresponding class.

Analysis on Alternative Design:
However, the alternative design is not suitable and not ideal as it violates many Design and SOLID principles:

1. Don’t Repeat Yourself Principle (DRY)

- Since they have similar functionality, we should not straight away implement the logic individually within each classes. The code will be very repetitive if we do so.

2. Open Closed Principle (OCP)

- If we were asked to add more feature in these classes, we will violate OCP as we have to modify the existing classes. By doing this, the implementation of code will be unstable if we frequently change the code.

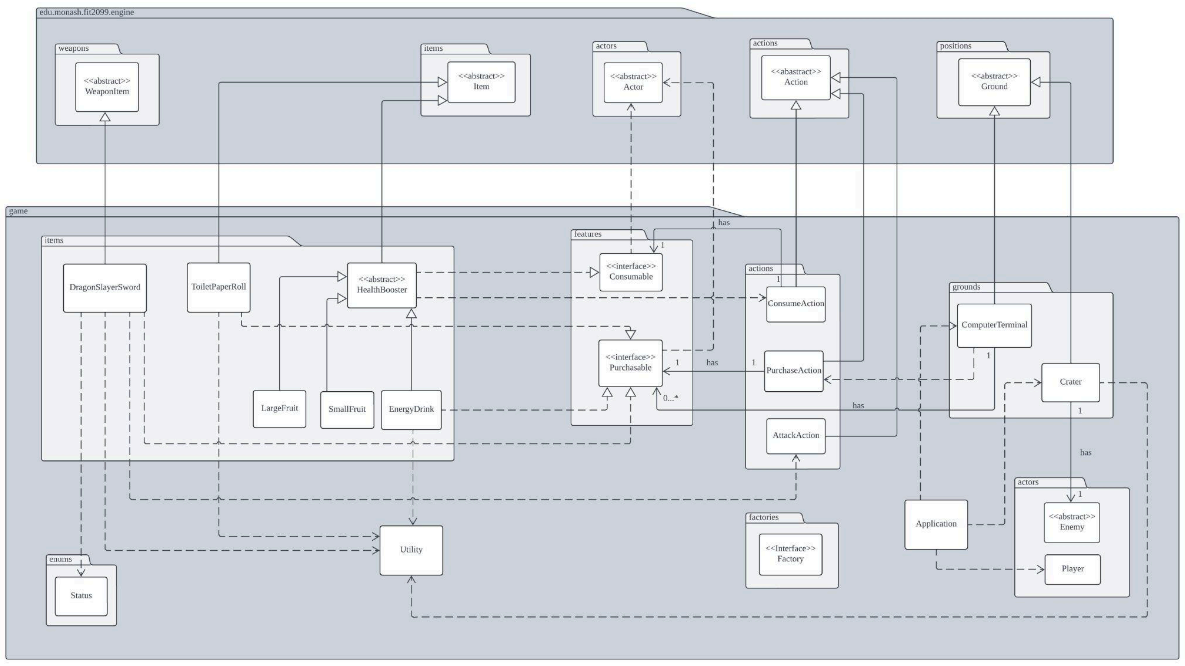
Also each class can only extends from one abstract class, in order to achieve potential future extension, we has decided choose the final design and use interface for more flexibility.

Conclusion:

Overall, our chosen design provides a robust framework for intern interacts with new in game items with their functionality: Jar of Pickles that has a 50% of chance of being pasts its expiry date. It will hurt intern by 1 point if it expired, heal intern by 1 otherwise. A Pot of Gold that an intern can take out the gold from the pot and gain 10 credits. Intern can also consume water from the Puddle, gaining 1 maximum health. We ensured that these classes reusing the Consumables interface, such that if we need to add features in the future, we don’t have to implement changes in theses classes, instead we can simply just modify the Consumables interface. (OCP) This has brought flexibility and space for future extension. In the future, if there is an extra feature for Puddle class such that the water is poisoned and will hurt intern when consumed, we can simply override the consume method in Puddle class and achieve this easily without modifying other classes.

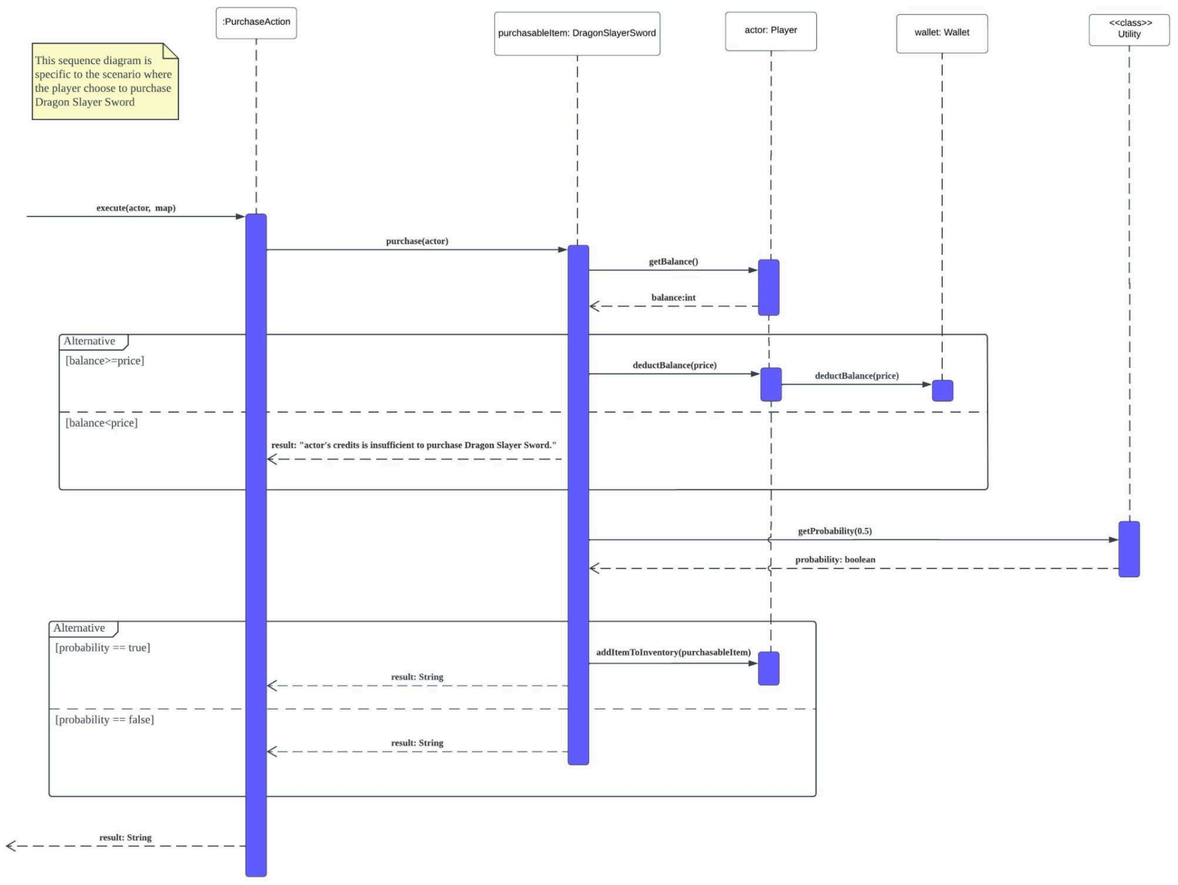
REQ4: Static factory's staff benefits

UML Diagram



Sequence Diagram

Chosen Scenario : The Player choose to purchase Dragon Slayer Sword



Design Goal :

The design goal for this assignment is to allow the intern to purchase items from Computer Terminal with his credits. The Computer Terminal can print items including Energy Drink, Dragon Slayer Sword and Toilet Paper Roll. Energy Drink can heal the intern. Dragon Slayer Sword can be used to attack Enemy by intern. Toilet Paper Roll is just a normal item. There are some special scenarios for each item when the intern purchases them.

Design Decision :

DragonSlayerSword extends the abstract WeaponItem class in the engine. The ToiletPaperRoll class extends the abstract Item class in the engine. EnergyDrink class extends the HealthBooster abstract class. HealthBooster is an abstract class from A1, the design rationale will be discussed below. The Purchasable interface is created to enable the functionality where an object can be bought by an actor. PurchaseAction is a class that extends the abstract Action class and it is an action for an actor to purchase a purchasable object. ComputerTerminal extends the abstract Ground in the engine and is modified to provide the PurchaseAction in the tick method.

Final Design :

DragonSlayerSword is a weapon that the actor can pick up and attack another actor in the game. WeaponItem in the engine has these properties. Extending the WeaponItem class can prevent the repeating implementation (DRY). This DragonSlayerSword represents the dragon slayer sword(SRP). The advantage is that we only need to provide the detail of this weapon by passing it to its parent's constructor and overwriting the allowableActions to generate an AttackAction using this weapon based on the status of actors at the exits of the owner. The disadvantage is that as the variety of weapons grows, the number of classes will increase due to our adherence to the Single Responsibility Principle.

ToiletPaperRoll can be picked and dropped off by the actor. Item class has this basic property so inheritance can prevent this repeating implementation (DRY). ToiletPaperRoll represents toilet paper roll in the game(SRP). The benefit is it is easy to add more features for ToiletPaperRoll by just adding methods to this class. The disadvantage is the number of classes will grow if we add more items to the game in the future.

The HealthBooster class provides the implementation of the consumable interface that allows an actor to consume and heal their hit point, therefore inheritance can prevent a similar implementation (DRY). The advantage is only the details of EnergyDrink are needed to pass into the constructor of the HealthBooster. EnergyDrink represents the energy drink in the game (SRP). The advantage is we can make EnergyDrink with more specific features by implementing other interfaces or adding methods. The disadvantage is the number of classes will be increased if more item that can heal player is added in the game.

Purchasable interface only focuses on the event of an object being bought by the actor (ISP). DragonSlayerSword, ToiletPaperRoll and EnergyDrink implement this interface to provide the logic when an actor purchases them. The advantage is that any class can implement this interface to enable it to be purchased by an actor. The disadvantage is that we must provide implementation when using this interface and the implementation may be similar occasionally.

PurchaseAction can replace the Action object in the engine when executing the action chosen by the actor without causing an error (LSP). PurchaseAction has an association with Purchasable to allow this Purchasable object to be bought by an actor. It depends on the Purchasable interface to complete this action (DIP). The advantage is that PurchaseAction can accept any objects that implement the Purchasable interface and allow the actor to purchase it using their credits.

ComputerTerminal acts like a Ground so we can avoid repeating implementation by inheriting the Ground class (DRY). The advantage is that ComputerTerminal can be added into the game as a Ground object. It can replace the Ground in the game to execute the code without causing error (LSP). The disadvantage is that we need to instantiate it in the Application when adding it to the game.

ComputerTerminal has a dependency relation with the Purchasable. Since the Factory is a generic factory, we provide the Purchasable as the type for the factory to allow the ComputerTerminal to only hold the factories that can generate Purchasable objects. A ComputerTerminal may have zero or many Purchasable factories to generate the PurchaseAction. Any type of Purchasable factory can be held by the ComputerTerminal and replace the Purchasable factory to execute the method that generates Purchasable without causing an error (LSP). ComputerTerminal depends on the Purchasable factory without any low-level module (DIP). The advantage of using a generic factory is we can use syntactical sugar as the factory and we do not need to create classes when an object must be generated by a factory. The disadvantage is that we need to provide the type of class when declaring the factory, so it will make one more dependency line between the class using the factory and the class to be generated by the factory.

Computer Terminal is added to the game through the Application class. When the intern is on or beside the Computer Terminal, the ComputerTerminal will provide the option for the intern to buy the item. When the intern holds the item, the action allowed by the item will be shown as an option to the intern, such as DragonSlayerSword can be used to attack Enemy if they are at the exit of the intern. EnergyDrink can be consumed by the intern.

Alternative Design :

An alternative way is an abstract class can be created to represent the Item class that can be purchased by the player. Assume the abstract class is called PurchaseItem and it must extend the Item class and provide an abstract method. Now, it should be accepted by the PurchaseAction to ensure only Item can be purchased can cause a PurchaseAction. We need to repeat the implementation of WeaponItem and Booster for the features of Dragon Slayer Sword and Energy Drink since PurchaseItem does not provide their features. This design violates the DRY principle. Interface is chosen as different classes can have this same feature and we will not repeat the implementation stated above.

Conclusion :

Overall, our final design ensures that the Computer Terminal will only generate Purchasable objects and only Purchasable objects can cause a PurchaseAction. Interface helps us to provide different implementations and any subclass of the Item can implement it to adhere to the DRY principle in this requirement. We also reused the abstract class created before since they follow the OCP principle. In the future, if an actor can be purchased by the intern, it is easy to achieve this requirement by implementing the Purchasable interface and adding it to the factory of ComputerTerminal. In addition, if the player can sell a Purchasable object to another actor, the purchaseAction can be reused by returning it in the allowableAction method of the player.