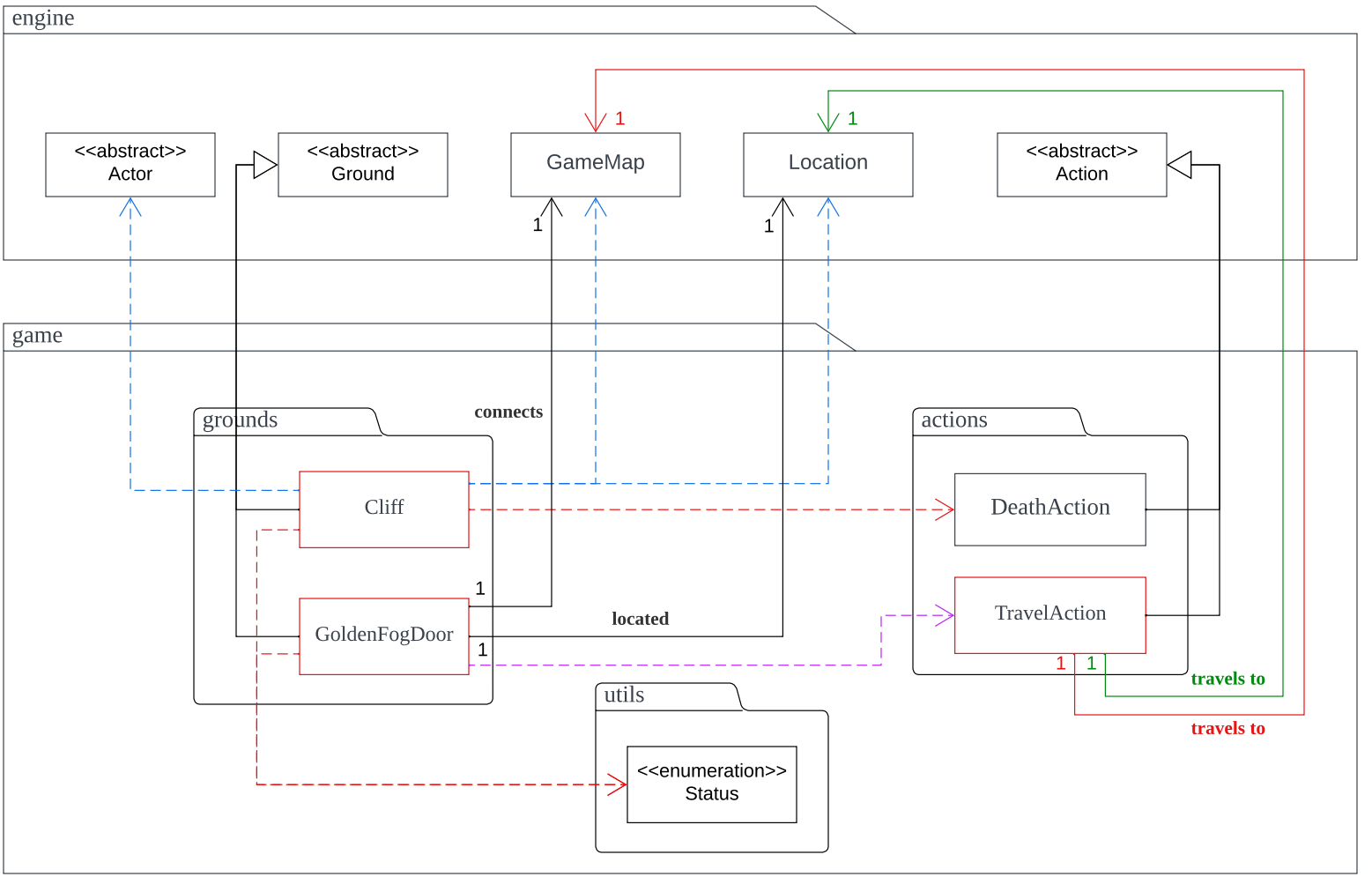
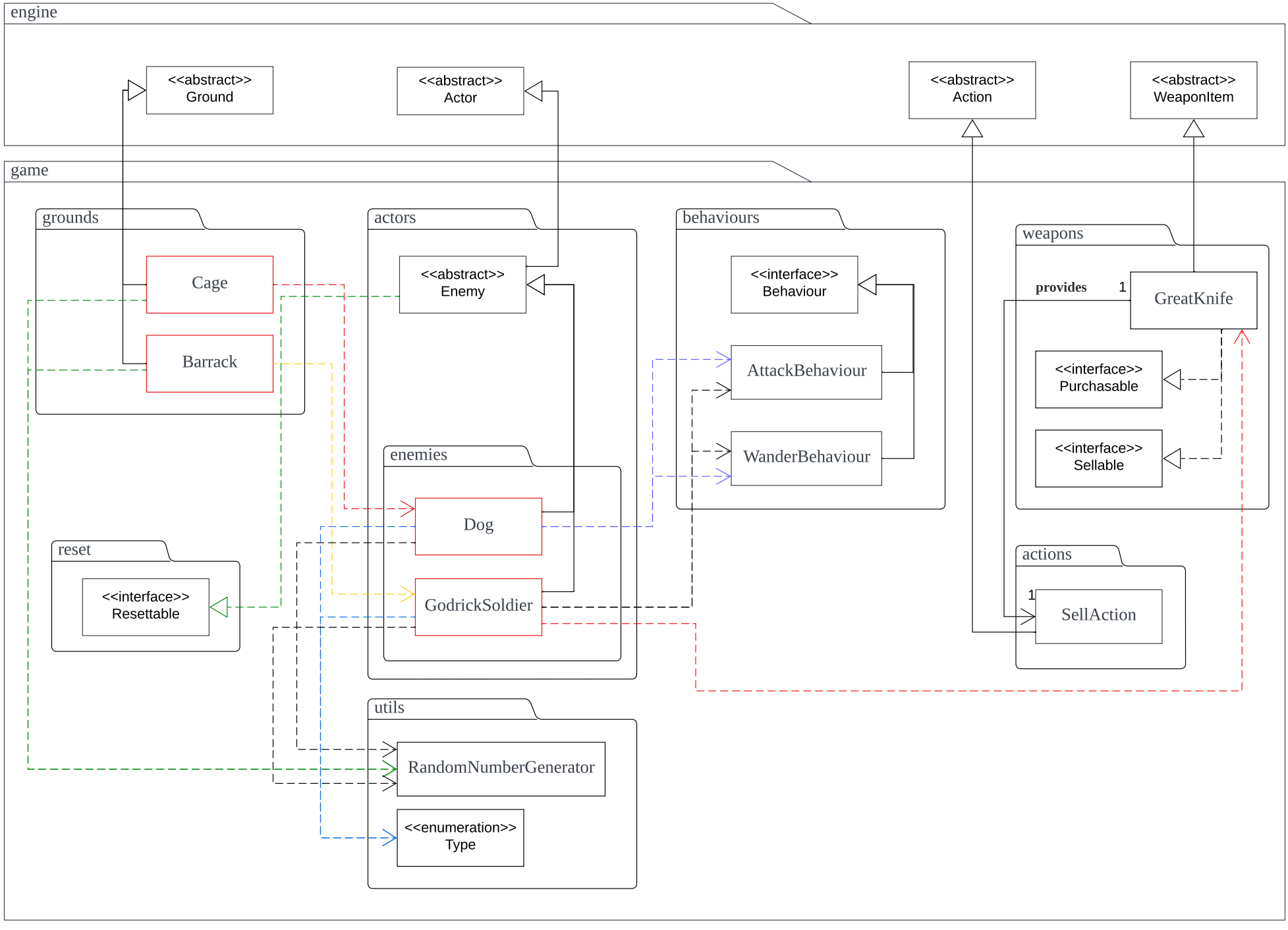
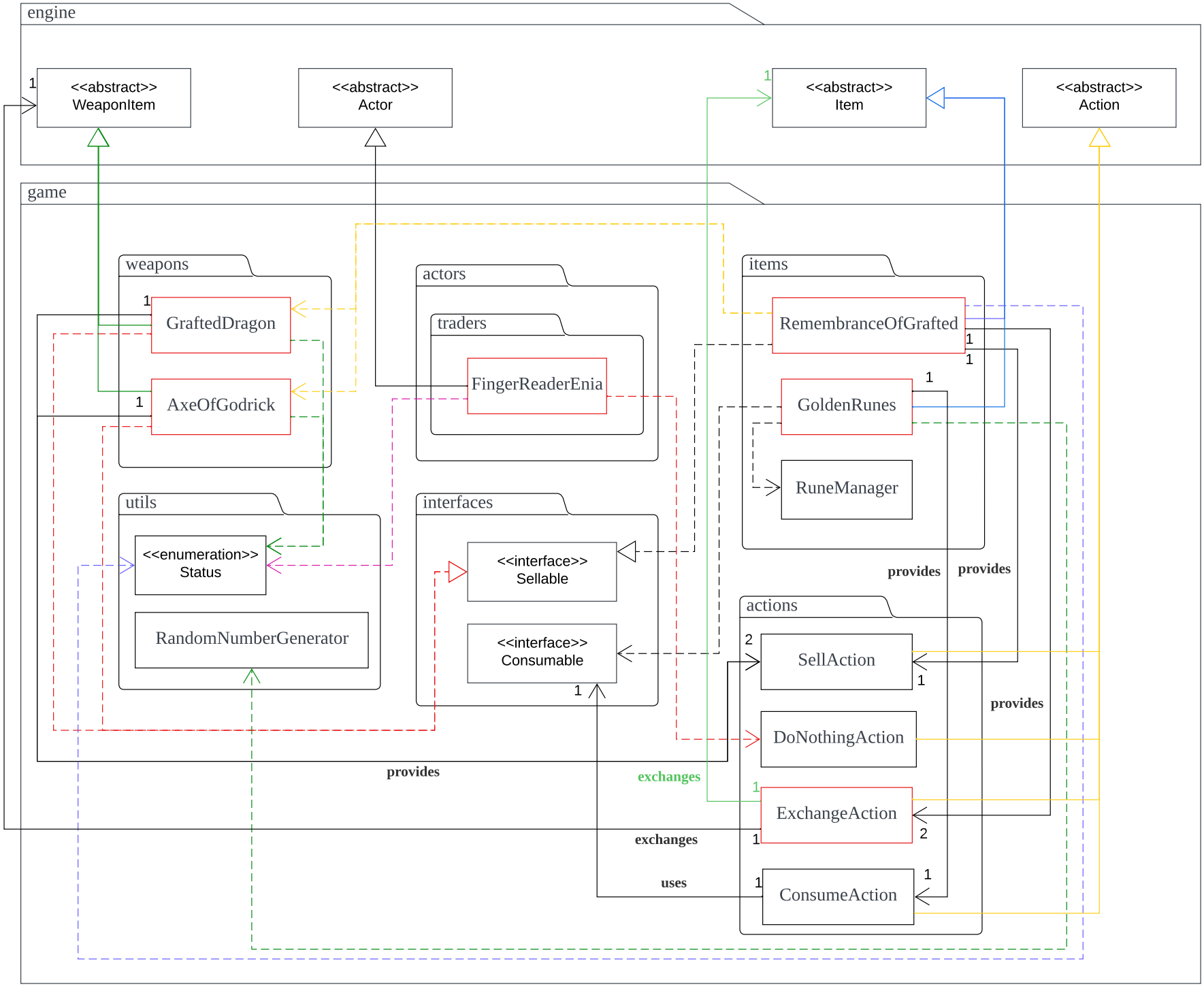
Assignment 3 Requirement 1



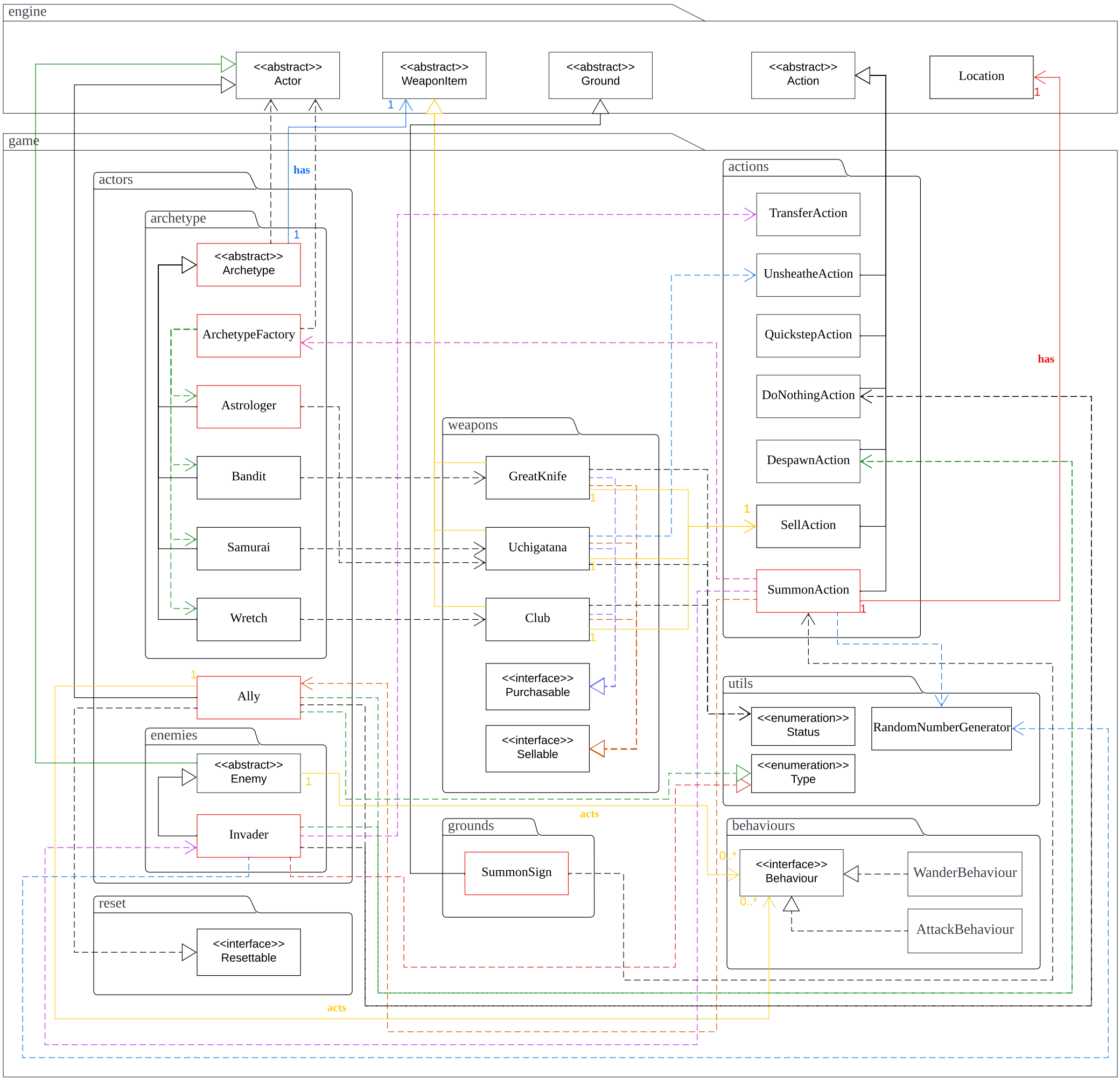
Assignment 3 Requirement 2



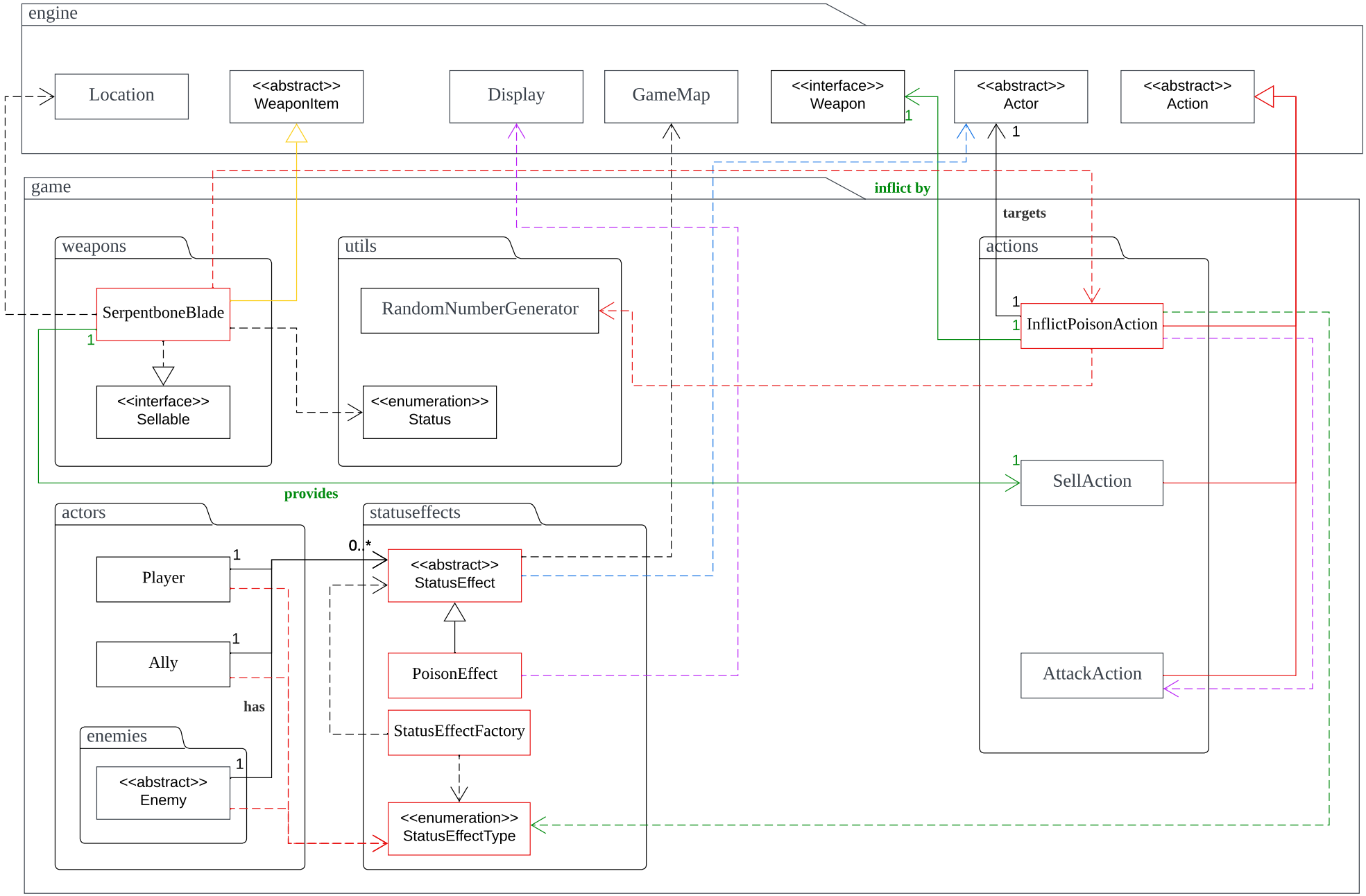
Assignment 3 Requirement 3



Assignment 3 Requirement 4



Assignment 3 Requirement 5



FIT 2099 Assignment 3 Design Rationale

Design Goals:

- Extensible
- DRY Principle
- Maintainability
- Understandability
- SOLID Principle

The diagrams represent an object-oriented system for a text-based rogue-like game that is inspired by the actual game “Elden Ring”. There are a few design goals that we aim to accomplish, including making the design extensible in case new features are added later, ensuring ease of maintenance and understanding for future game developers, avoiding repetition, and finally adhering to the SOLID principle during the implementation phase.

REQ 1

Cliff and GoldenFogDoor are concrete classes that extend the abstract Ground class. It is because they hold characteristics of a Ground. One advantage of this approach is the reduction of code repetition during the development of these classes. One minor disadvantage of this design is that it increases the complexity of the overall system as more classes are introduced into the system. This design approach adheres to the Single Responsibility Principle (SRP) as each class has its own unique characteristics and implementation, also the DRY principle is not breached as the code repetition is avoided by inheriting codes from a parent class. Furthermore, this design approach made the addition of new grounds easy to implement as the new grounds could simply inherit the abstract Ground class and then provide its unique implementation.

To fulfil the requirement of player travelling between maps, a TravelAction concrete class is created where it inherits the abstract Action class. The reason for this is that TravelAction holds the characteristics of an Action. The advantage of this approach is that it reduces code repetition (DRY) in the implementation phase as most codes can be inherited from Action class, and only the unique implementations are required to be coded. One minor disadvantage of this approach is that more classes would be introduced to the systems in the future if more game actions are required, hence increasing the complexity of the system. This design approach however, adheres to several design principles, such as Single Responsibility Principle (SRP) since the TravelAction has its own unique implementation and has only one responsibility. Also, it adheres to the Liskov Substitution Principle (LSP) because in any part of the system where it expects an object of Action type, TravelAction is capable of substituting Action as TravelAction is a subtype of Action. This design also allows the system to have new actions to be easily added and implemented as the new actions can inherit from the Action class and provide a unique implementation.

REQ 2

The two different enemies, Dog and GodrickSoldier which are concrete classes, extends the abstract Enemy class, which in turn extends the abstract Actor class. Since they share similar characteristics and methods as Enemy, it is necessary to extend from Enemy to avoid repetition, which follows the DRY principle. Instead of turning the Enemy class into a god class, we have distinct dedicated classes with their own responsibilities. Hence, it complies

with the Single Responsibility Principle (SRP) and Open Closed Principle (OCP), since newly added types of enemies can easily inherit the Enemy class while adding new features, but it does not change any implementation of base class. However, this approach has some disadvantages, which includes the Enemy class may have abstract methods that may not be suitable for child classes to implement, so the child class will have to implement an empty method or throw an Exception. To combat this issue, an alternative such as an interface can be used instead for certain methods.

Optional Requirements in REQ 2
Since HeavyCrossbow is an optional requirement, we replace this weapon with an existing weapon which is GreatKnife in the system. For instance, the weapon of a GodrickSoldier would be a GreatKnife instead of a HeavyCrossbow.

Cage and Barrack, which are concrete classes, extend the abstract Ground class because they share the characteristics of Ground. Consequently, this approach results in less repeated code, which adheres to DRY when these classes are implemented. On top of that, it fulfils the Single Responsibility Principle (SRP) as each of the Ground has their own details to store and method to implement. Nonetheless, the overall complexity of object oriented systems will be an issue to raise, since the increment of classes will indirectly increase the complexity.

REQ 3

AxeOfGodrick and GraftedDragon are concrete classes that extend from the WeaponItem abstract class. It is because these classes share similar characteristics and attributes of a WeaponItem. Doing so avoids the repetition of code during the implementation phase of these weapons therefore adhering to the DRY principle. On the other hand, the one disadvantage of this approach is that the complexity of the overall systems increases whenever more weapons are introduced. This design approach however adheres to the Single Responsibility Principle (SRP) as each weapon class created would have its own unique implementation but only has a single responsibility which is to represent a weapon. On the other hand, this approach simplifies the process of implementing new weapon classes in the future as new weapons can simply provide the unique implementation and also extends the WeaponItem class. Meanwhile, to fulfil the requirements of these weapons being sellable to the traders in the game, they also implement the Sellable interface which was previously created in assignment 2. The advantage of his approach is that it is easy to implement weapons that can be sold to the traders without making changes to the SellAction or the trader's code. This design therefore also adheres to the Dependency Inversion Principle (DIP). In the future, the new weapons that can be sold can simply implement the Sellable interface.

RemembranceOfGrafted and Golden Runes are concrete classes that extend from the abstract Item class since they have behaviours and characteristics of an item in the game. The advantage of this design is that repetition of code would be reduced, which adheres to the DRY principle. Not doing so would turn the Item class into a god class which violates the single responsibility principle. Furthermore this approach also made the addition of new items easy to implement as the new item classes can inherit from the abstract Item class and then provide the respective implementation. On the other hand, since RemembranceOfGrafted is an item that can be sold, it implements the Sellable interface.

This reduces the dependencies between traders and the items that can be sold, as the addition of new items that can be sold did not affect the higher level modules, therefore adhering to the Dependency Inversion Principle (DIP). Meanwhile, the Golden Runes would implement the Consumable interface as it is an item that can be consumed. This design approach reduces the dependencies between ConsumeAction and the Golden Runes as a layer of abstraction was created which indicates that the code of ConsumeAction needs no modifications to its implementation. This approach adheres to the Dependency Inversion Principle (DIP). This approach is extensible as in the future, any items that can be consumed can simply implement the Consumable interface and provide the implementation details.

FingerReaderEnia is a concrete class that inherits the Actor abstract class because it holds similar behaviours and characteristics with the Actor. One benefit of this approach is that during the implementation phase of the FingerReaderEnia, repetition of codes would be reduced which fulfils DRY principle. One minor disadvantage of this approach is that the FingerReaderEnia may need to implement the abstract methods defined in the Actor class which it does not need or throw an Exception. To combat this issue, an alternative such as an interface can be used instead for certain methods.

ExchangeAction is a concrete class that extends from the abstract Action class as it holds characteristics of an action. One benefit of this design is that it reduces the code repetition when implementing the class, which therefore fulfils the DRY principle. The disadvantage of this method is that the complexity of the overall system becomes complex as more action classes would be added whenever new action is required in the game. This design approach adheres to the Liskov Substitution Principle (LSP) as ExchangeAction is a subtype of Action class, hence it can substitute Action to any part of the system where an object of type Action is expected. In the future, if any new actions are needed it is easy to add as the action can simply extend from the Action abstract class and then provide the unique implementations.

Optional Requirements in REQ 3
Since GodrickTheGrafted is an optional requirement, it is therefore not implemented. Therefore, the RemembranceOfGrafted is added to the player's inventory for the purpose of testing the exchange feature with FingerReaderEnia.

REQ 4

For assignment 3, we added a new Archetype abstract class which is to be extended by the various archetypes (Bandit, Samurai, and Wretch) from assignment 2 instead of the archetypes extending from the Player class. This is because in assignment 3, not only the player can choose an archetype as allies and invaders can as well. Implementing Archetype in this approach, it minimises code repetition in various archetypes which will be introduced in the future. Hence, this design approach adheres to the DRY principle while making the system extensible. Furthermore, this design is easy to extend as new archetypes introduced in the future can simply just extend from this Archetype abstract class such as the one introduced in assignment 3.

Astrologer is a concrete class that extends the Archetype abstract class, since it holds similar characteristics to an Archetype. Therefore, reducing code repetition by following the DRY principle. Additionally, it fulfils the Single Responsibility Principle (SRP), as each of the archetypes has their own identities and methods to be implemented. Apart from that, this

approach is extensible when there's a newly Archetype added to the system, as it can inherit the Archetype abstract class, but in spite of that the complexity of the object-oriented system will be increased when increment in the number of classes being created.

ArchetypeFactory is a concrete class, responsible for creating an Archetype that is chosen by the player before starting the game or randomly selected by allies or invaders. Meanwhile, this design approach adheres to the Single Responsibility Principle (SRP), since the class has its own responsibility which is creating Archetypes. Additionally, it adheres to the Open-Closed Principle (OCP) while making the system extensible since this approach allows the developer to add newly Archetype that is being introduced in the future without modifying any previous codes that were implemented.

Ally, a concrete class which extends from the Actor abstract class. This design implementation adheres to DRY principle by minimising code repetition since it shares similar characteristics with Actor class. Additionally, making Ally an individual class fulfils the Single Responsibility Principle (SRP), as different Actors will have their own implementation of the method and details. Nonetheless, one of the disadvantages is the overall complexity of object-oriented systems will increase due to increment in the number of newly added classes.

Invader is a concrete class that extends the Enemy abstract class as it shares common characteristics and methods of an enemy. Therefore, inheriting these identities can avoid repetition (DRY) while also adhering to Single Responsibility Principle (SRP), since Invader has its own identities and methods to be implemented. Consequently, these approaches result in less repeated code (DRY), but increase the overall complexity of object-oriented systems when these classes are implemented. Nevertheless, an issue in implementing this approach is increasing the complexity or system when newly added Enemy classes.

Optional Requirements in REQ 4
Since AstrologerStaff is an optional requirement, we replace this weapon with an existing weapon which is Uchigatana in the system. For instance, the weapon of an AstrologerStaff would be an Uchigatana instead of an AstrologerStaff.

SummonSign is a concrete class that inherits from the abstract Ground class as it shares similarities and responsibilities as a Ground. The advantage of this approach is that it decreases the repetition of code (DRY) since similar codes to implement a Ground object are not needed. Moreover, this also follows the principle of Single Responsibilities Principle (SRP) as SummonSign is an unique ground and has its own implementation and characteristic. Following this design approach, it makes the system extensible easily since the newly added ground class can be inherited from the Ground abstract class. While the system is being further implemented, the complexity will be higher since there’s more new classes added to the system.

SummonAction, a concrete class that extends the abstract Action class since it shares common attributes as well as methods. Hence, it is necessary to abstract these identities for avoiding repetition, indirectly adhering to the DRY principle. Additionally, the design implementation of SummonAction fulfils the Single Responsibilities Principle (SRP), as it has its own specific features to deal with. Undeniably, this design implementation allows the

system to be extensible easily, as the new action to be introduced in the future can simply inherit the Action abstract class. However, the complexity of the system could be a risk, since the more classes being implemented the higher complexity of the system will be taken into account.

REQ 5

SerpentboneBlade is a concrete class that extends from the abstract WeaponItem class as it shares similar characteristics and behaviours of a weapon item. One benefit of this approach is that during the implementation stage of the class, the repetition of code is reduced as most of the codes can be inherited from the WeaponItem class, therefore this adheres to the DRY principle. However, one minor disadvantage of this design is the overall complexity of the system. Meanwhile, to fulfil the requirement of the weapon being sellable to the traders in the game, SerpentboneBlade also implements the Sellable interface. Overall, there are several design principles that have been adhered during the implementation of this weapon. First of all, Single Responsibility Principle (SRP) as the requirement of the new weapon is fulfilled through the creation of a new weapon class, and not doing so would turn the WeaponItem class into a god class, which violates the Single Responsibility Principle (SRP). Additionally, the Liskov Substitution principle is also adhered as the SerpentboneBlade class can substitute in place of the WeaponItem class in any part of the system where it expects an object of type WeaponItem as it is a subtype of the WeaponItem class. The extensibility of this design approach is fairly good as any new additions of weapons in the future can simply extend the WeaponItem class and then provide unique implementations.

PoisonEffect is a concrete class that extends from the abstract StatusEffect class since it has the characteristics of a StatusEffect. The unique implementation of a poison effect would be implemented in the PoisonEffect class. One advantage of this approach is that it avoids the use of “instanceof” where the type of the object has to be checked before use, as polymorphism can be used. One of the design principles that this approach adheres to is the Liskov Substitution principle, as PoisonEffect can substitute StatusEffect in any part of the system where an object of type StatusEffect is expected. In the future, if any new status effects need to be added to the system, it can be implemented easily by creating a new class that inherits the abstract StatusEffect class and provides a unique implementation, while other code that depend on the StatusEffect class requires no modifications to account for this addition of the new status effect.

StatusEffectFactory is a class acting like a factory class that is responsible for the creation of a StatusEffect object. This design approach reduces the coupling between the client code with all of the existing StatusEffect since the client code can simply use the factory method provided by the StatusEffectFactory class. This design approach adheres to the single responsibility principle as the only responsibility that the StatusEffectFactory holds is to handle the creation of StatusEffect objects. Furthermore, it also adheres to the open close principle as any new status effects can simply be added in the StatusEffectFactory class and other parts of the client code needs no modification, which makes the system extensible in the future. However one minor disadvantage is that the complexity of the overall system increases as more classes would be added, which in turn also reduces the readability of the code.

InflictPoisonAction is a concrete class that inherits the abstract Action class because it shares similar behaviours and attributes of an Action. One advantage of this approach is that it reduces the repetition of code in the development stage of the class as most attributes or methods can be inherited from the parent class, hence this adheres to the DRY principle. The disadvantage of this design is that more classes would be introduced to the system which increases the scale and complexity of the overall system, which reduces the readability of the code. Meanwhile this design approach adheres to the Liskov Substitution principle as it is a subtype of Action, therefore it can substitute Action in any part of the code where an object of type Action is expected. In the future, any new features or action introduced can be easily added and implemented into the system as the new action class can inherit the Action class and the provided unique implementation.