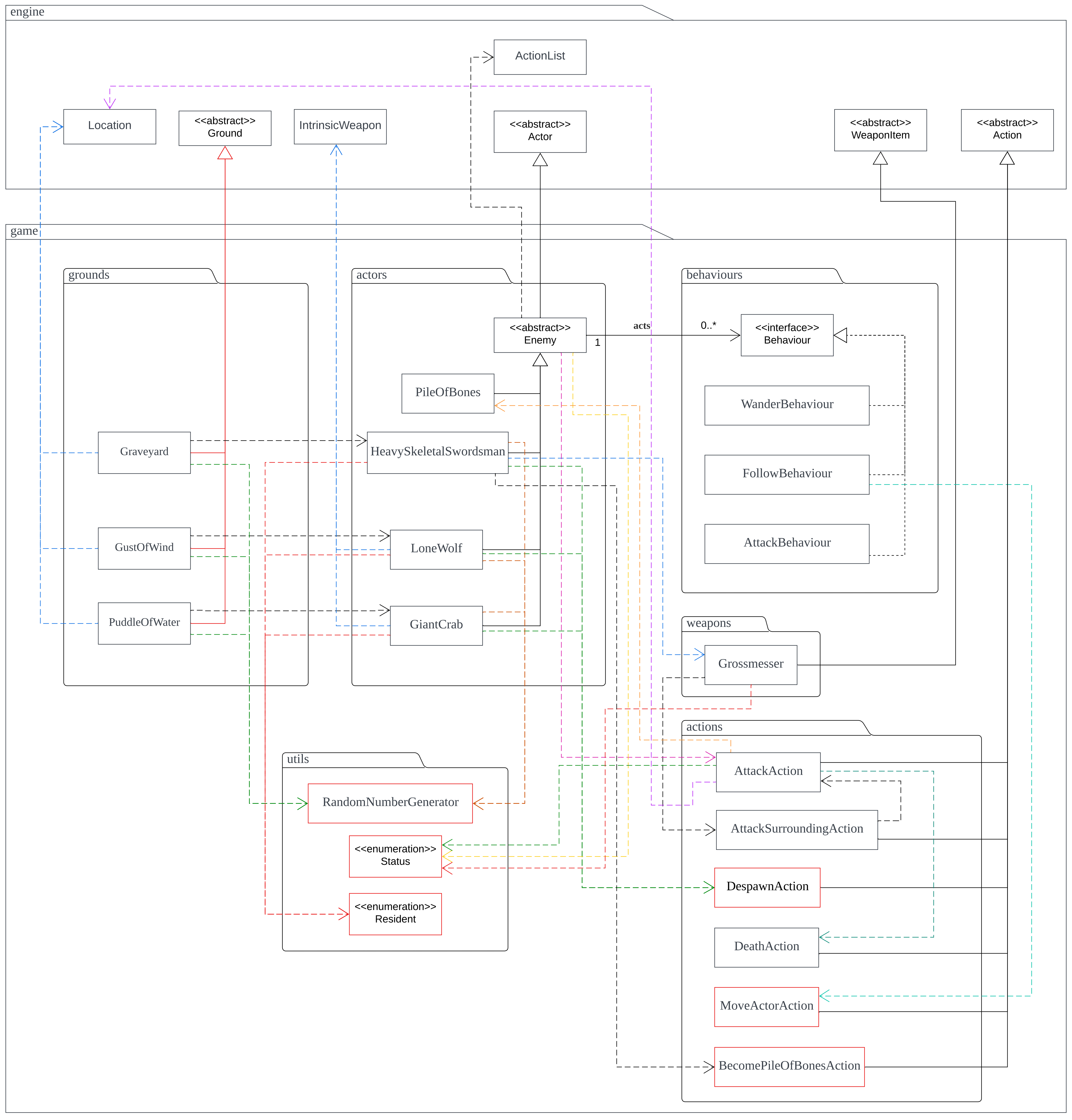
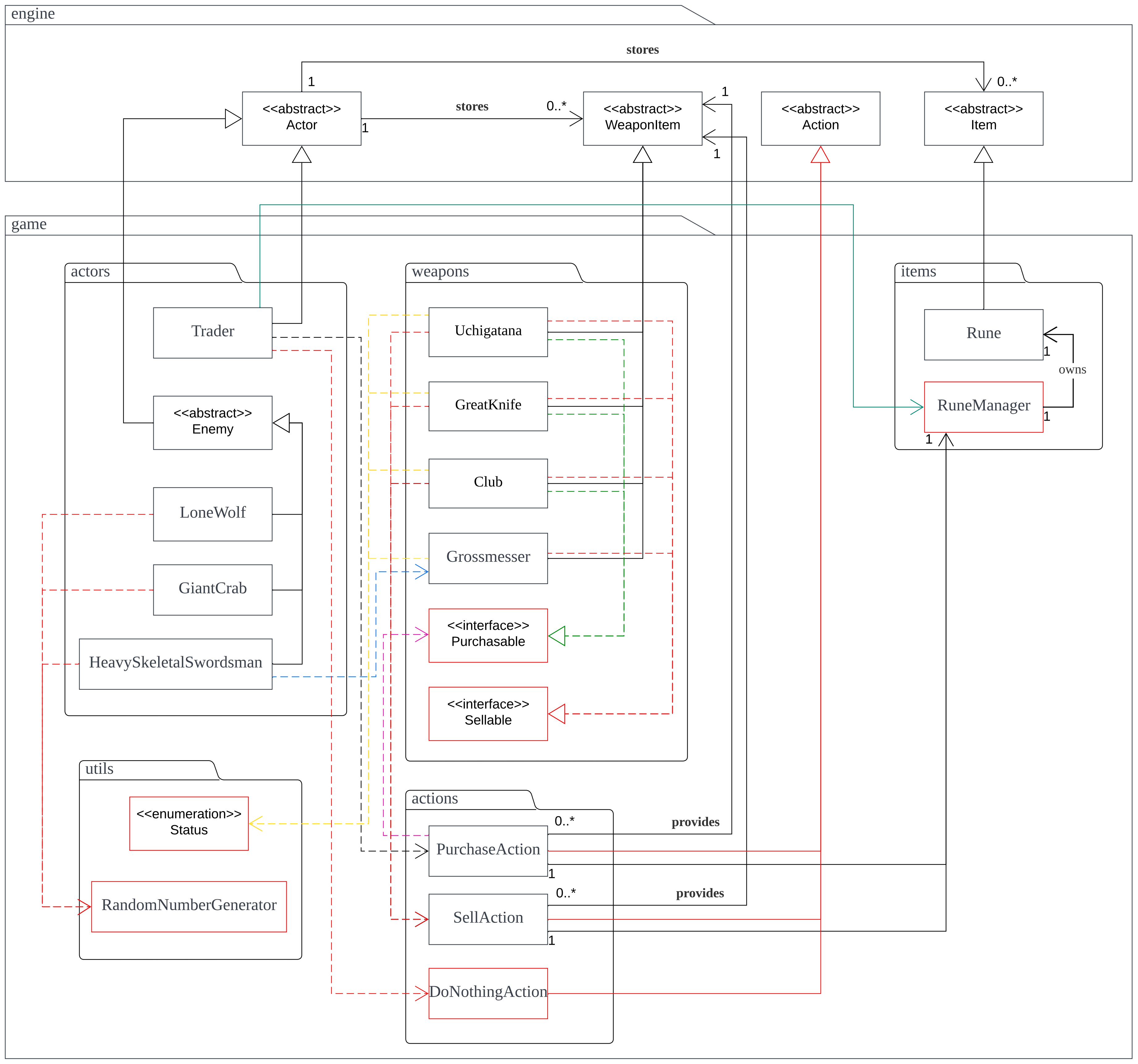
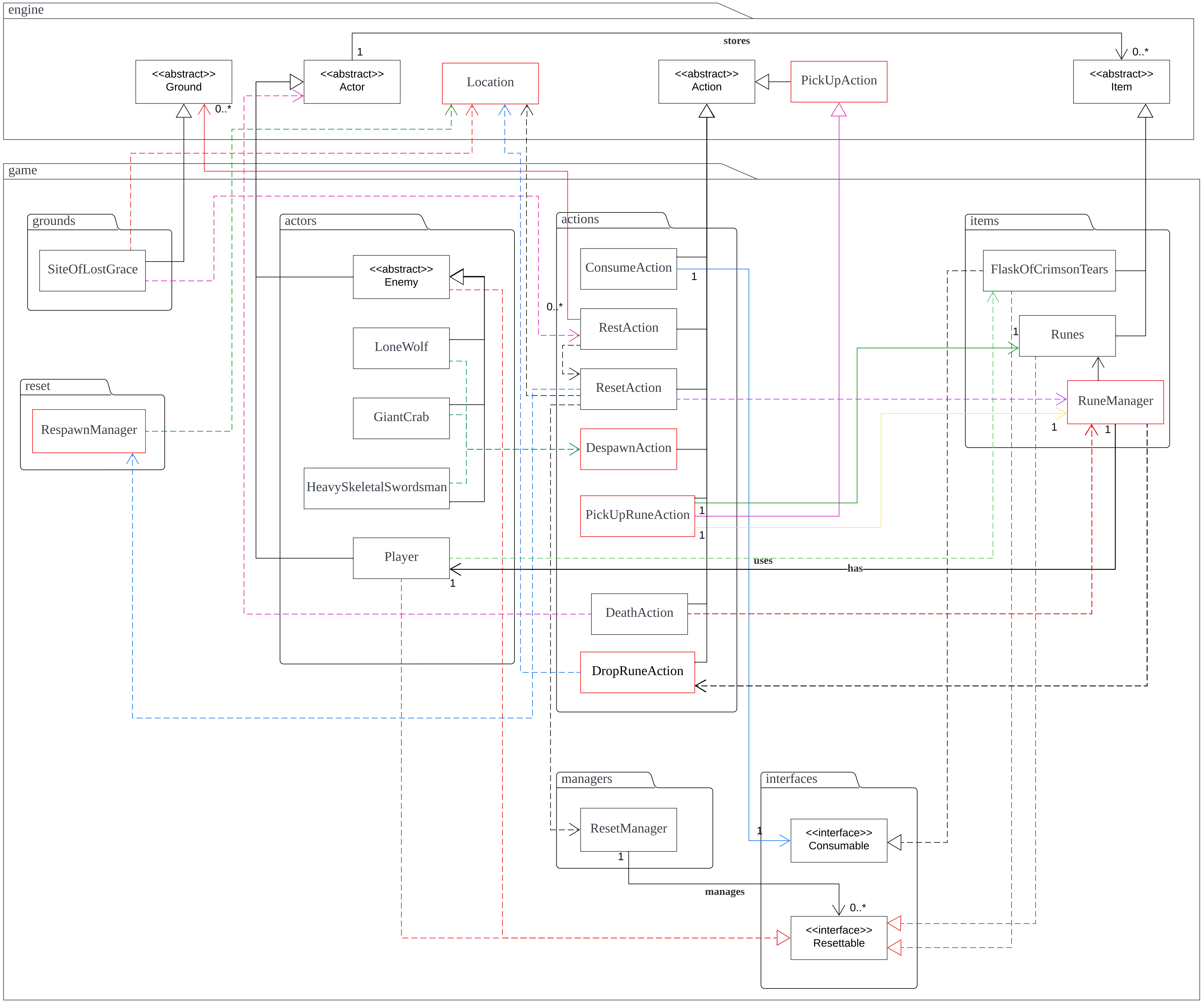
Requirement 1



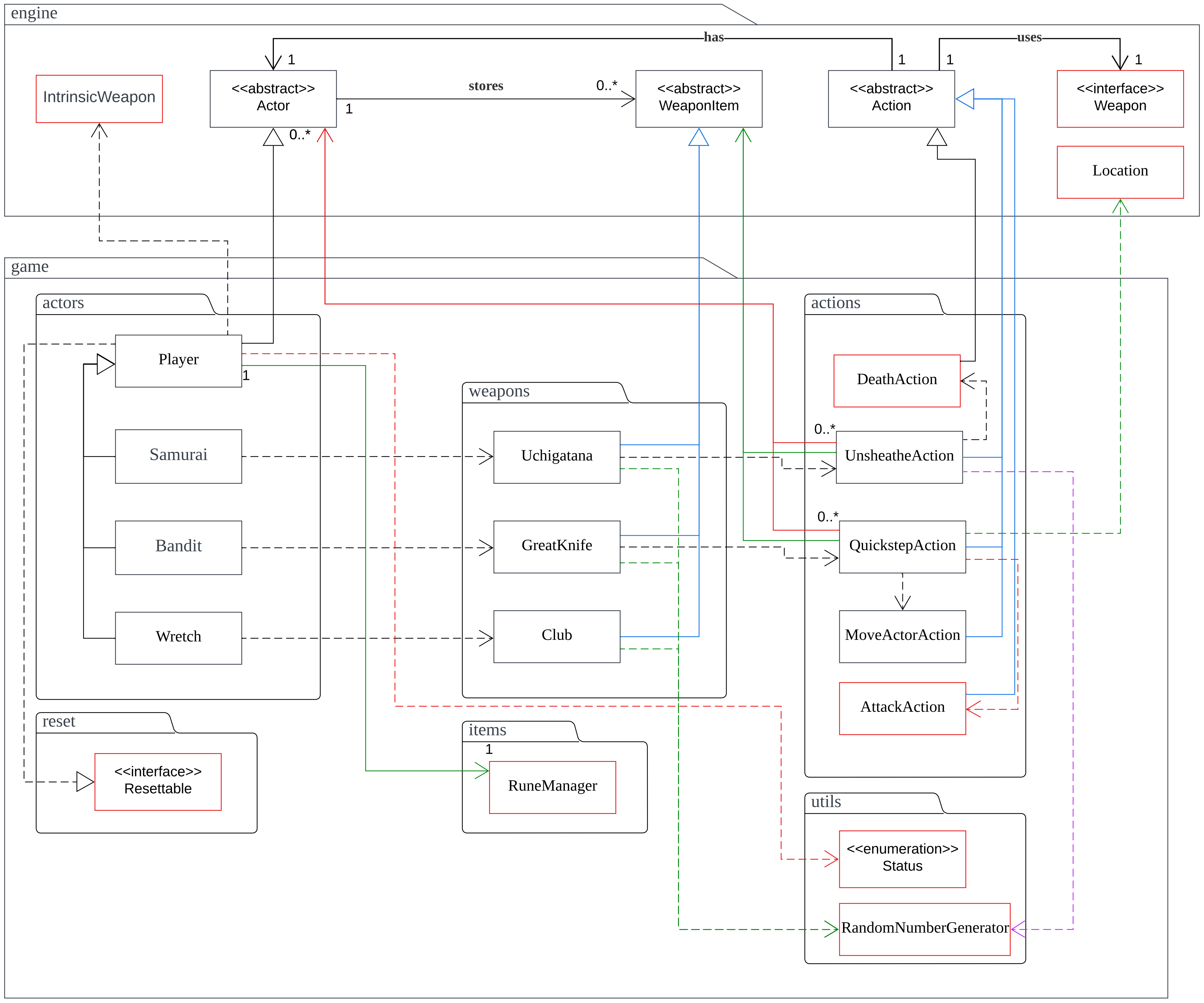
Requirement 2



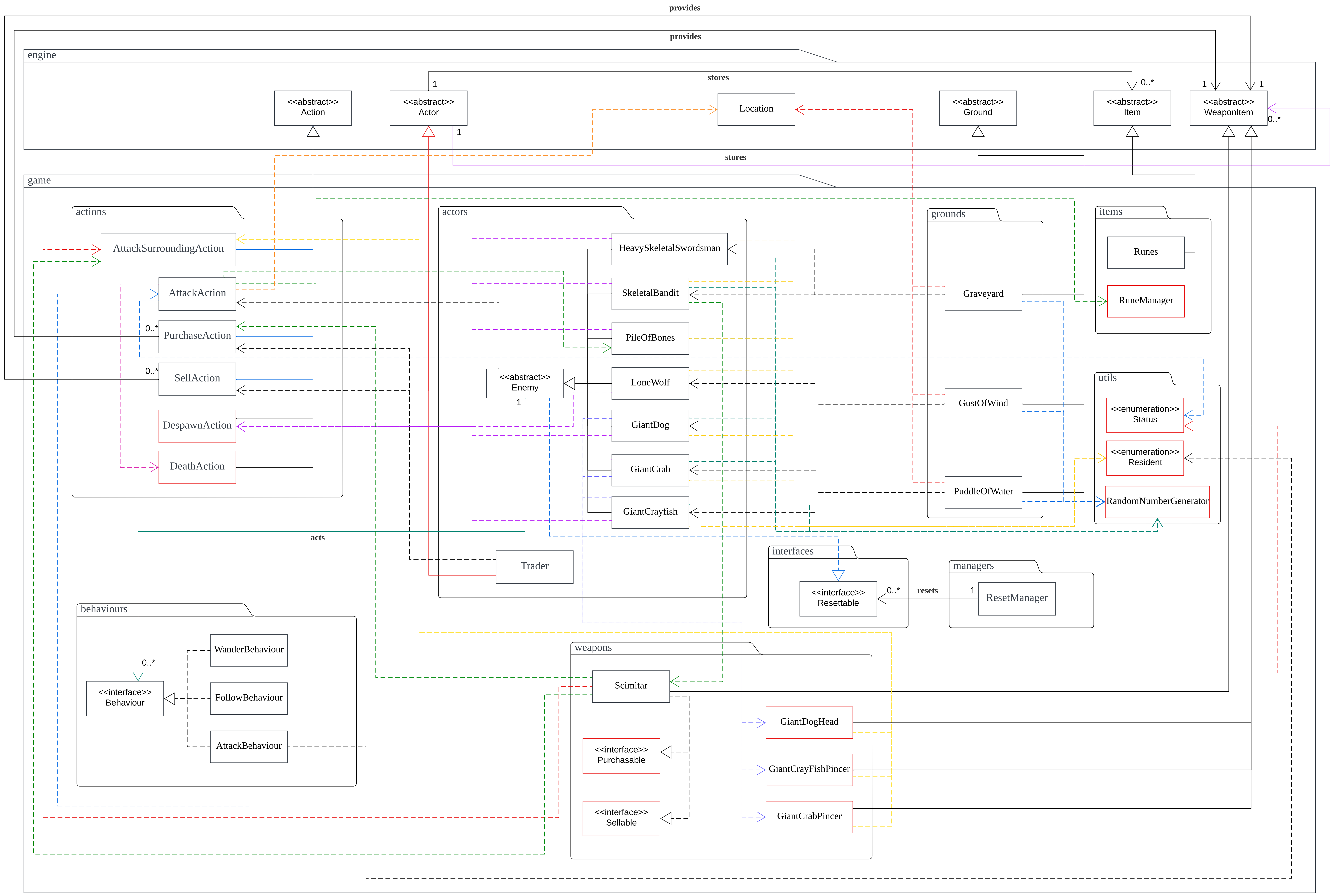
Requirement 3



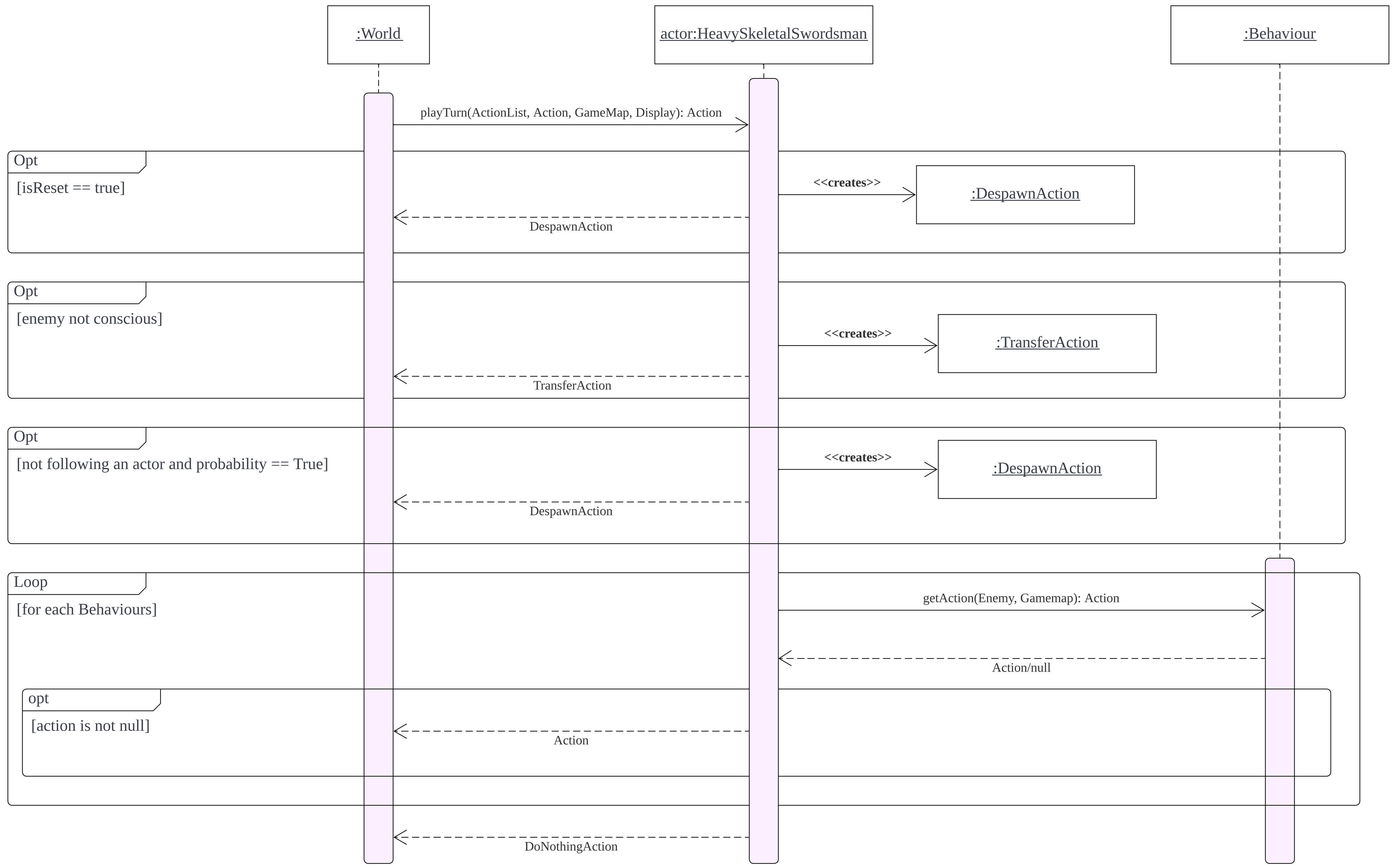
Requirement 4



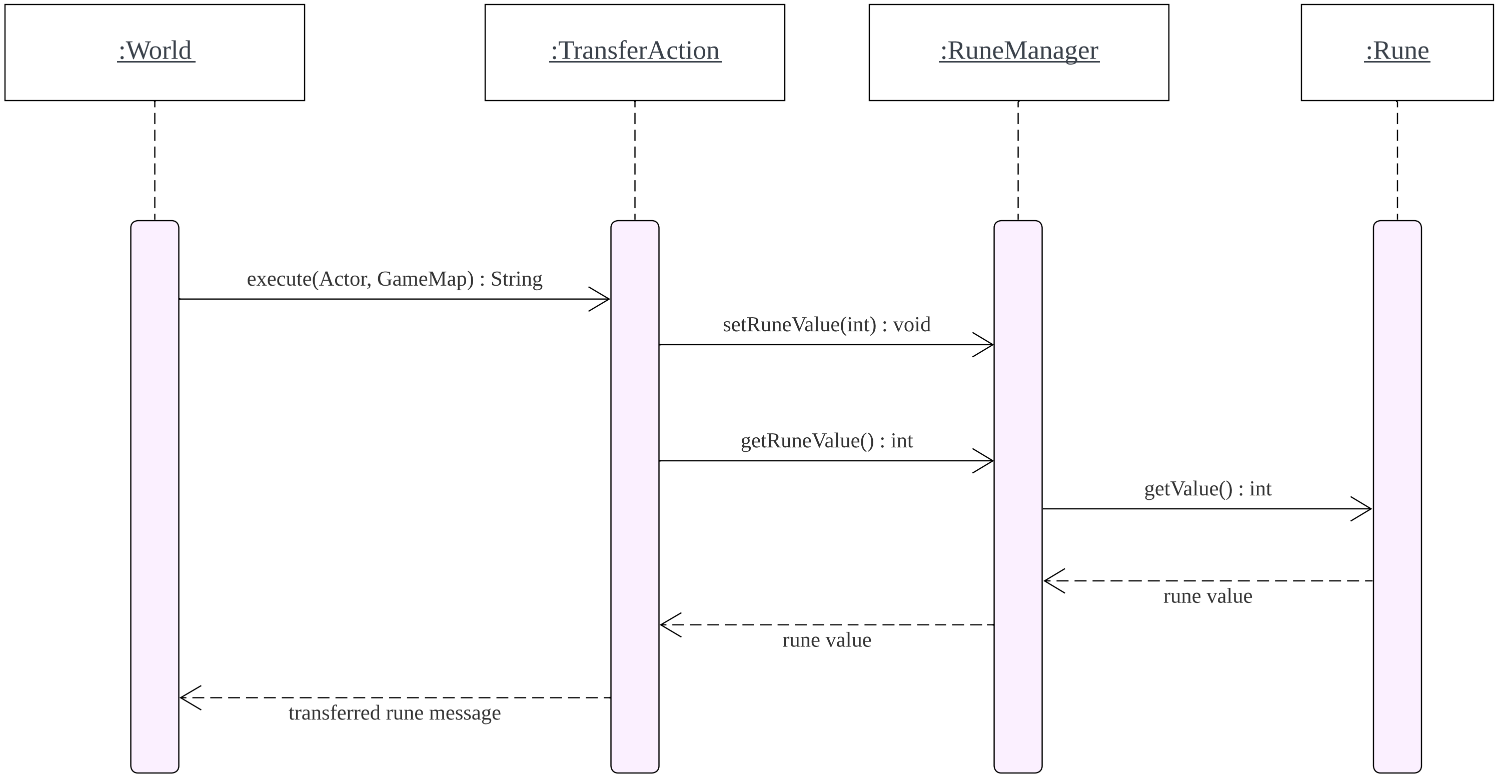
Requirement 5



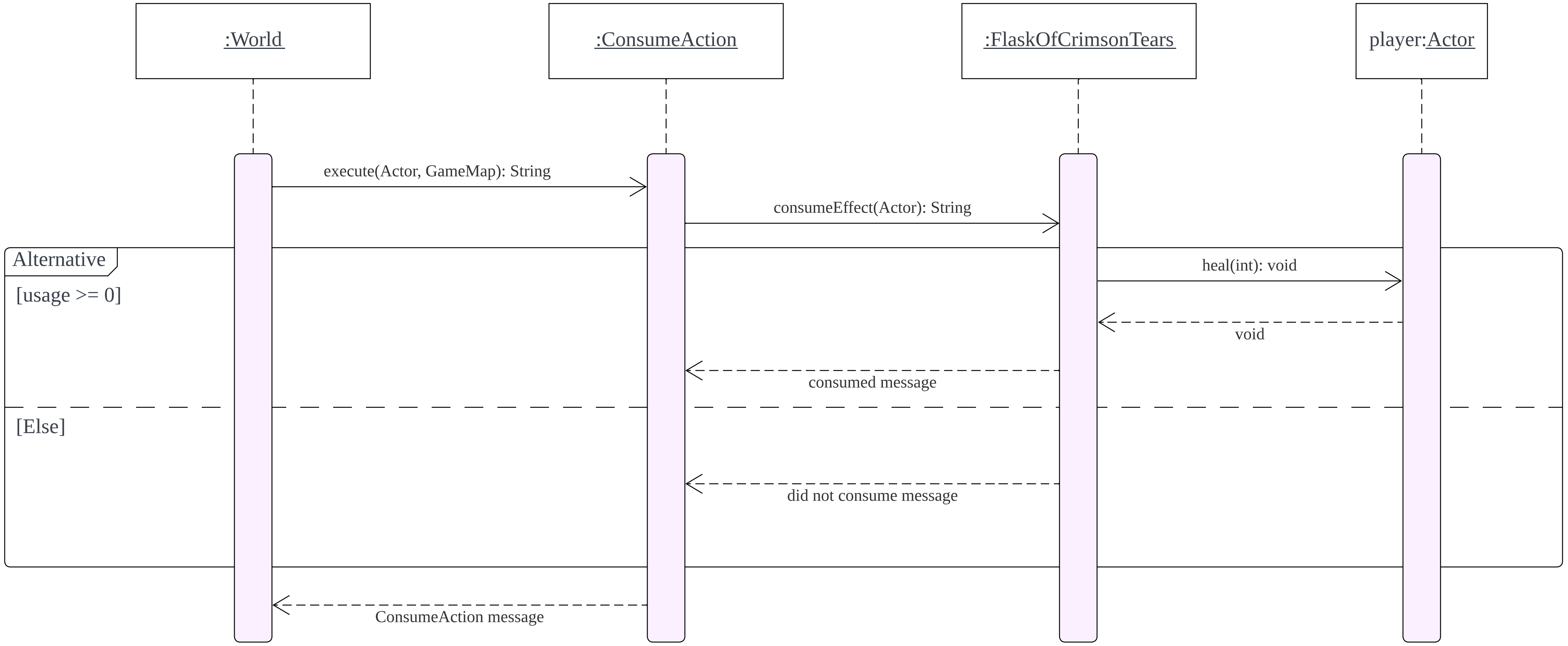
REQ1: Interaction Diagram for World invoking Enemy's playTurn method



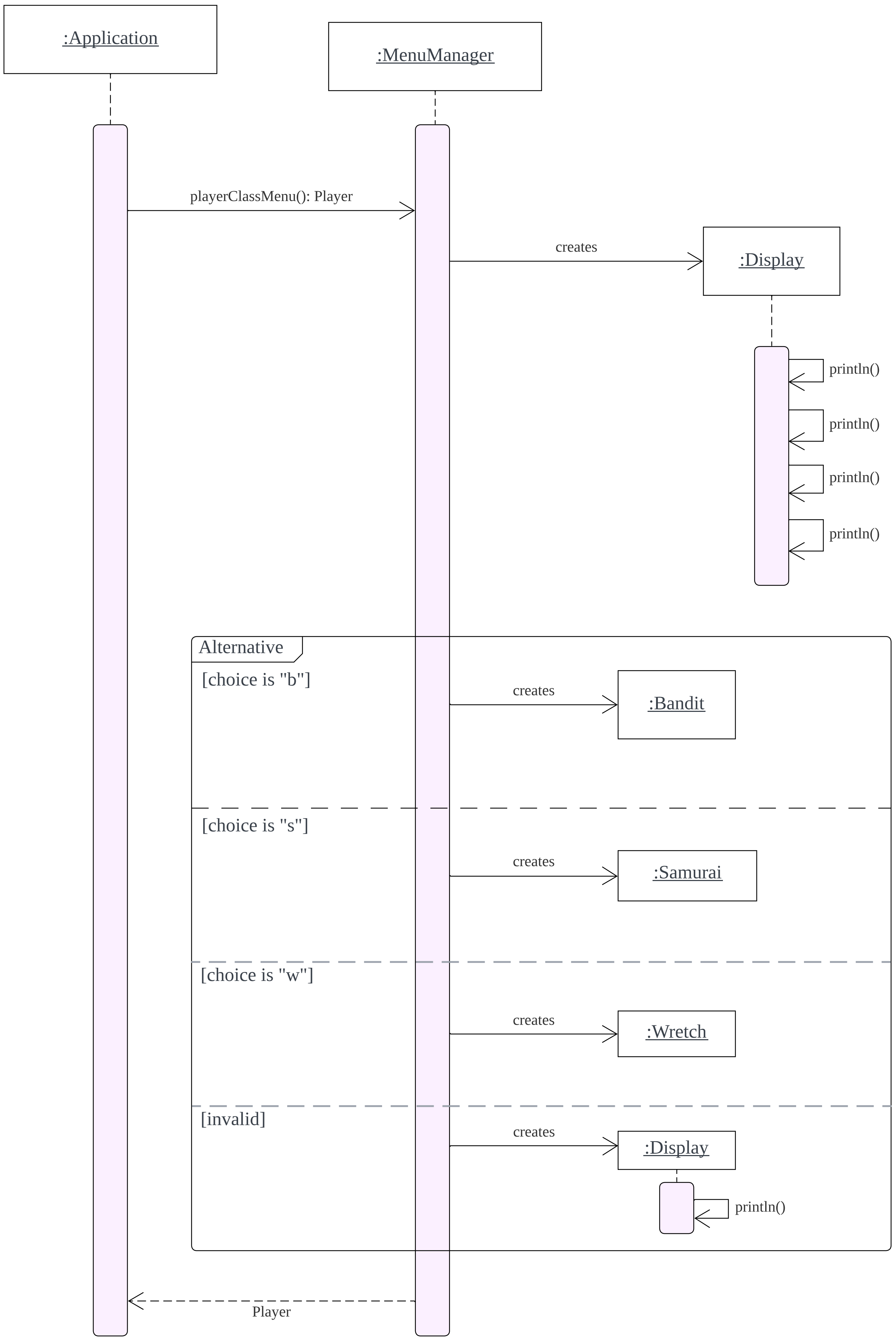
REQ2: Interaction DiagramTransferAction's execute method



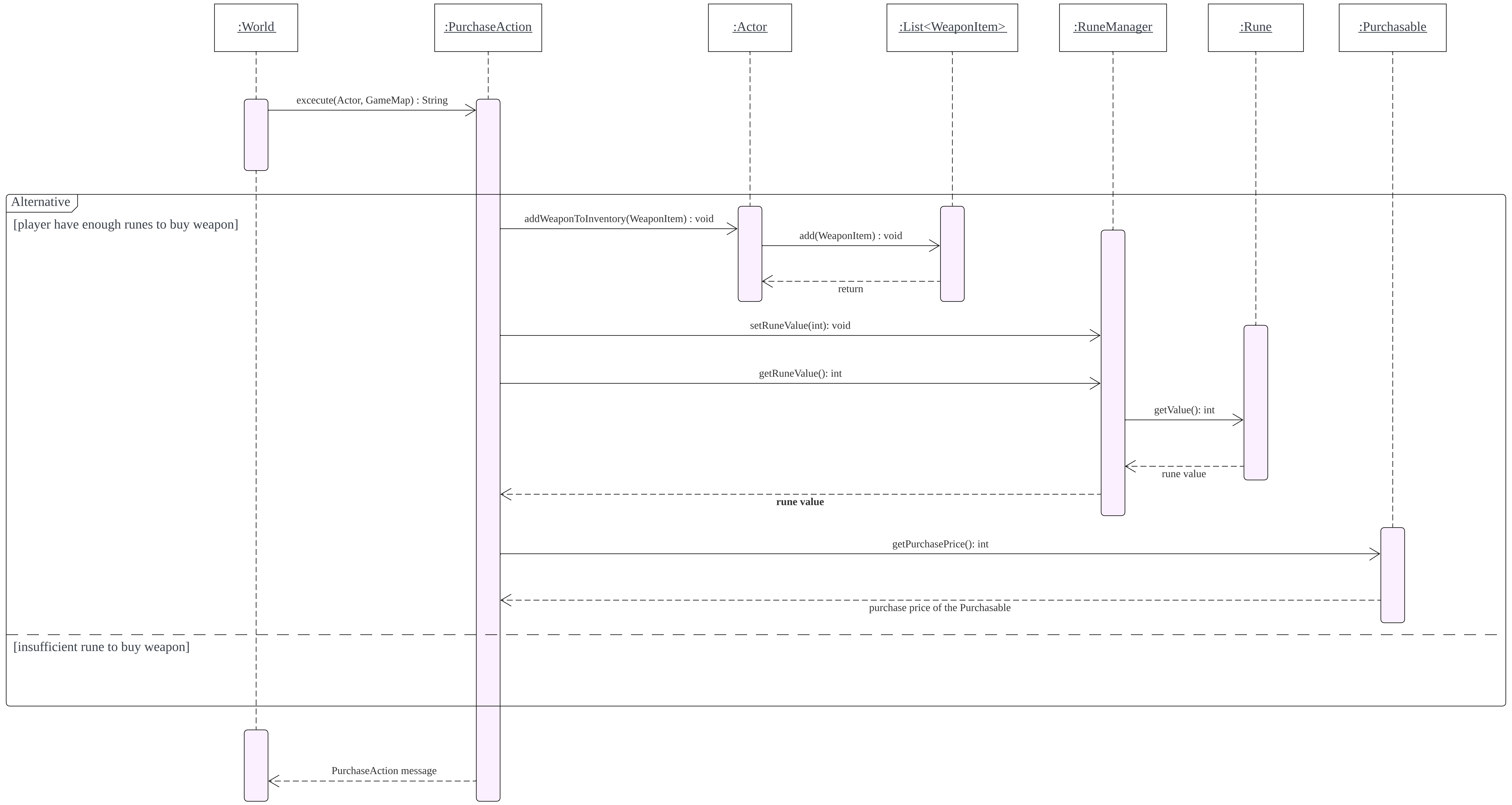
REQ3: Sequence Diagram for ConsumeAction of FlaskOfCrimsonTears



REQ4: Sequence Diagram for PlayerClassMenu of MenuManager



REQ5: Interaction Diagram of PurchaseAction



FIT 2099 Assignment 2 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

The four different enemy types, which are concrete classes, extend the abstract Enemy class, which in turn extends the abstract Actor class. Since they possess certain shared characteristics and methods, it is necessary to abstract these identities 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. Therefore, it complies with the Single Responsibility Principle (SRP) and the 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 the base class. However, this approach has some disadvantages, such as 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 disadvantage, an alternative such as an interface can be used instead for certain methods.

Furthermore, the Enemy class has an association with the Behaviour interface, while all Behaviour classes implement the Behaviour interface; as a result, any modifications made to the implementation of the respective Behaviour classes would not affect how Enemy uses the Behaviour interface. The benefit of this design is that the system would reduce the dependencies between the Enemy class and all of the Behaviours. In addition, any new Behaviour needed can simply implement the Behaviour interface, and the code of the Enemy class can remain unchanged, therefore fulfilling the Dependency Inversion Principle (DIP). However, a minor disadvantage of this approach is that it reduces the readability of the design.

Moreover, AttackAction and AttackSurroundingAction are concrete classes that extend the Action abstract class, as these classes share similar characteristics with an Action. Also, making them an individual class adheres to the Single Responsibility Principle (SRP), as different Actions will have different implementations of the “execute” method.

Grossmesser extended the abstract WeaponItem class. As Grossmesser shares common attributes and methods with WeaponItem, just that it is a specific weapon, Hence, inheriting these identities can avoid repetition (DRY).

Graveyard, GustOfWind, and PuddleOfWater, which are concrete classes, extend the abstract Ground class because they share the characteristics of a Ground. Consequently, this approach results in less repeated code (DRY) when these classes are implemented.

<u>REQ 2</u>

Trader is a concrete class that extends the abstract Actor class because it shares similar characteristics with the Actor. This decreased code repetition occurs during the creation of the class, which fulfils the DRY principle. One disadvantage of this method is that the Trader might inherit methods from Actor that it doesn't need, hence making it bloated with unnecessary methods.

Uchigatana, GreatKnife, Club, and Grossmesser are concrete classes that extend the abstract WeaponItem class in the game engine. Since they share common attributes and characteristics of a WeaponItem, it is logical to inherit these identities to avoid repetitions (DRY). Doing so reduces the repetition of code when these weapons are added to the game, which adheres to the DRY principle. This approach also made the addition or modification of the system easier, as any new weapon that may need to be added can simply inherit the WeaponItem abstract class while having its own implementation, which adheres to one of our design goals, which is the extensibility of the system.

// New from A2

Meanwhile, weapons that can be purchased from the Trader such as Uchigatana, GreatKnife and Club implements the Purchasable interface. This approach creates a layer of abstraction between the PurchaseAction and Uchigatana, GreatKnife, Club. Doing so reduces the dependency between the PurchaseAction and these purchasable weapons. One advantage of this design is that the code for the PurchaseAction requires little to none modification whenever a new Purchasable weapon is added in the future as the new weapon could simply implement the Purchasable interface. One disadvantage of this design is that it reduces the readability and complexity of the codes. This design adheres to one of the SOLID principles which is the dependency inversion principle.

On the other hand, PurchaseAction and SellAction are concrete classes that inherit the abstract Action class since they represent an action. Also, making them an individual class adheres to the Single Responsibility Principle (SRP), as different Actions will have their own implementations of the "execute" method.

Additionally, Rune is a concrete class that extends the abstract Item class in the game engine. This design implementation minimises code repetition which fulfils the DRY principle since it shares similar characteristics with the Item class. Conversely, to prevent the Player class becoming a god class, we abstract the implementations of rune related matters by having a RuneManager class. This creates a layer between the Rune and the Player therefore reducing dependencies as Player would not be affected by the changes in the implementation of runes related matters.

<u>REQ 3</u>

FlaskOfCrimsonTears is a concrete class that extends from the Item abstract class as it holds similar characteristics to an Item. Hence, reducing code repetition by following the principle (DRY). Since it is an item that can be consumed by the Player, one approach is to create a ConsumeAction class, and the class would have a dependency on the FlaskOfCrimsonTears. However, this violates the Open Closed Principle (OCP) when a new Item that can be consumed is needed in the system, as the implementation of Consume Action would have to change accordingly; thus, an interface Consumable is made. The reason for the existence of the Consumable interface is because not all items can be consumed; therefore, items that can be consumed would implement this interface. Additionally, this approach reduces the dependencies between ConsumeAction and all items that can be consumed, as it would only depend on the interface's type rather than a specific Item class that can be consumed. This approach also made the addition of new consumable Items in the future easier as the ConsumeAction's code needs no changes to its implementation and the new Item can simply implement the Consumable interface and provide the implementation, which follows the Dependency Inversion Principle (DIP) and the Open Closed Principle (OCP). However, a minor drawback of this approach is that it decreases the readability of the code.

SiteOfLostGrace is a concrete class that inherits from the abstract Ground class as it shares similarities and responsibilities with a Ground. The advantage of this approach is that it reduces code repetition as similar codes to implement a Ground object are not needed. In addition, this also follows the principle of SRP as SiteOfLostGrace is a unique ground and has its own implementation and characteristics, hence creating a class of SiteOfLostGrace adheres to the SRP.

To handle the game reset requirement, the Resettable interface is used, and all classes that are resettable would implement this interface. Next, the ResetManager is created, which has an association with the Resettable rather than an association with all classes that are resettable if the Resettable interface is not used. This approach made our design more flexible for future extensions whenever a new class that is resettable is needed. Meanwhile, the ResetManager can simply call the reset method of each Resettable class without knowing the exact implementation of the reset method in the classes that are resettable. This design approach adheres to the Dependency Inversion Principle.

PickUpRuneAction and DropRuneAction are concrete classes that extend PickUpAction and DropAction class, which in turn inherits with Action class. Since the classes share similar characteristics and methods, it is necessary to abstract these identities to avoid repetition, which adheres to the DRY principle. Instead of turning PickUpAction and DropAction to a god class, we abstract the implementation for pick up and drop rune into PickUpRuneAction and DropRuneAction. Meanwhile, the design implementation of PickUpRuneAction and DropRuneAction fulfils the Single Responsibilities Principle (SRP), since both classes only have specific features to deal with.

REQ 4

The three Combat Archetypes extended the concrete Player class. Since they share common attributes and methods, with differences in weapons and HP that are specific to each archetype. Therefore, the decision to inherit the Player class is to avoid repetitions (DRY) as the implementations of each Archetypes share similar characteristics with the Player. Another advantage of this approach is that any new addition of the Combat Archetypes in the future would be easy to implement as only very little code needs to be made to the new class. However, one disadvantage is that these classes are tightly coupled with the Player class because as the Player class is a concrete class; thus, any modifications of the implementations made to the Player class may require modifications in these three Combat Archetypes. This approach not only adheres to the DRY principles but also adheres to the Liskov Substitution Principle as any part of the code that expects a Player class, these three classes are able to substitute themselves.

UnsheatheAction and QuickstepAction are concrete classes that extend the abstract Action class. The reason for this approach is that these 2 classes share and hold similar characteristics of an Action, therefore doing so reduces the repetition of code in the implementation of these classes which adheres to the DRY principle. In addition, creating these 2 classes adheres to the Single Responsibility Principle as these two actions have their own unique implementation, implementing these classes in the Action class would violate the SRP. The disadvantage of this approach is that the complexity and the readability of the overall code reduces. Meanwhile this design approach makes it easy to account for addition of new Action classes as only the unique implementations would need to be coded.

REQ 5

The newly added enemies in requirement 5, GiantDog, GiantCrab, and GiantCrayfish are concrete classes that extend the Enemy abstract class just like the previous enemies in requirement 1. The design is similar because they share common identities, attributes, and methods, which are abstracted in the Enemy class, hence reducing repetitions once again (DRY). This also simplifies the extension of new enemies classes in the future as new enemy classes could simply inherit the Enemy class. This approach shares the same disadvantage as REQ1 where some abstract methods declared in the Enemy class may not be suitable for the child class to implement if the new enemies have different characteristics. Hence, the alternative is to use an interface to account for these different characteristics.

Scimitar is a concrete class that extends the abstract WeaponItem class. Since Scimitar shares common attributes and methods with WeaponItem, it abstracts these identities in accordance with the DRY principle, which avoids code repetition in the Scimitar implementation phase. This design approach was previously used in REQ 2 for the addition of the new weapons, and the reason for the similarity is because Scimitar has the identity of a weapon. In the future, new weapons that need to be added can simply extend the WeaponItem class and do not need to repeat the codes to implement a weapon in the system. Furthermore, since the REQ5 mentioned that Scimitar can be bought from and sold to the Trader, the design approach of Dependency Inversion Principle allows us to implement this feature as stated in REQ2, this class can simply implement the Purchasable and Sellable interface and the code for PurchaseAction and SellAction requires no modification to account for this Scimitar class. One disadvantage of this approach is that the complexity of the overall code may be increased and the number of classes increases to account for new features.

GiantDogHead, GiantCrayFishPincer and GiantCrabPincer are weapons of the GiantDog, GiantCrab, and GiantCrayfish. These classes would extend the WeaponItem class as they share similar characteristics of a WeaponItem. This approach reduces the code repetition during the implementation of the code as the similar code is already provided in the

WeaponItem class and only little code would be needed to implement these classes, therefore adhering to the DRY principle. The disadvantage of this approach WeaponItem may contain abstract methods that these classes do not need to implement hence violating the liskov substitution principle.

For spawning the enemies in the East or West of the map, we decided not to use EnemyFactory classes to spawn the enemies but instead we spawn the enemies directly using the respective Ground. While this design approach violates the Single Responsibility Principle, the tradeoff is that this approach is easy to implement the spawning of the respective actors. The readability of the code is also better and easier to understand. Furthermore, using the EnemyFactory classes or creating new classes that extend the EnemyFactory abstract class decreases the simplicity of the code and increases the complexity of the overall design of the code. The disadvantage of this approach is that it violates the Single Responsibility Principle, since the classes have codes that include the spawning mechanism of the actors. Furthermore, any new actors that could be spawned at the respective grounds would require modifications of the implementation of the ground. One alternative is to have an EastEnemyFactory and WestEnemyFactory extending the EnemyFactory abstract class, then create another abstract class such as SpawnableGround where grounds that are capable of spawning would extend the SpawnableGround class. This approach would adhere to the Single Responsibility Principle as the ground classes would not be involved to spawn.