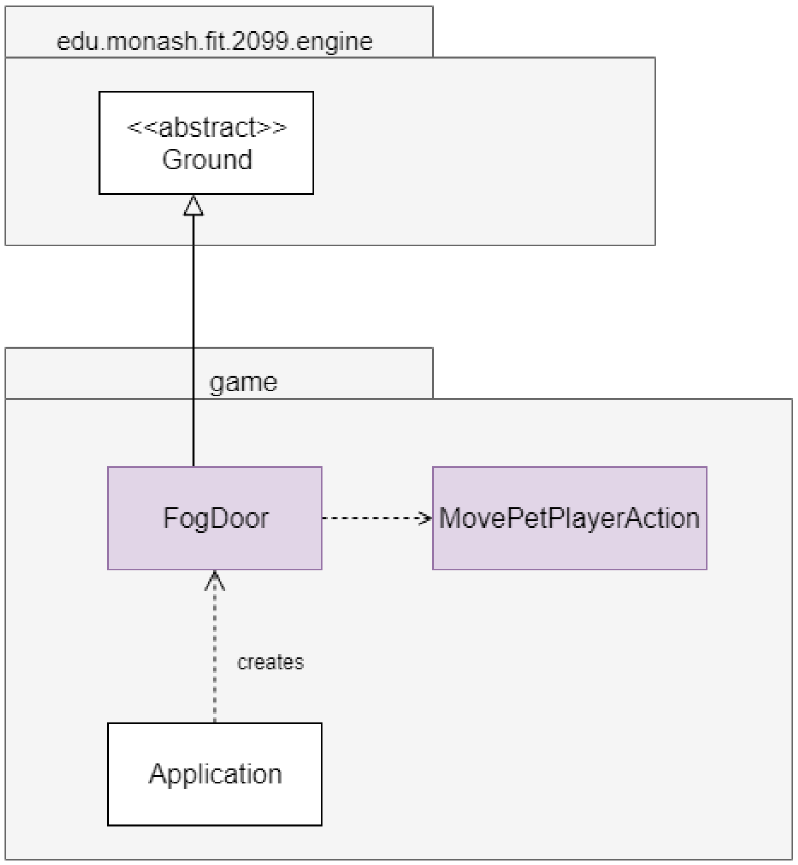
Assignment 3:

Design O' Souls (Extended Edition)

This document is to show the changes in Assignment 2.

- PURPLE: Newly added class in Assignment 3
- YELLOW: Newly added class in Assignment 2
- BLUE: Changes in Classes from Assignment 1
- RED: Newly changes method for Sequence Diagram
- GREEN: Newly added class in Assignment 1
- WHITE: Existing ClassF

Class Diagram 1



This class diagram shows the relationship between the FogDoor class and its allowable action.

New Classes:

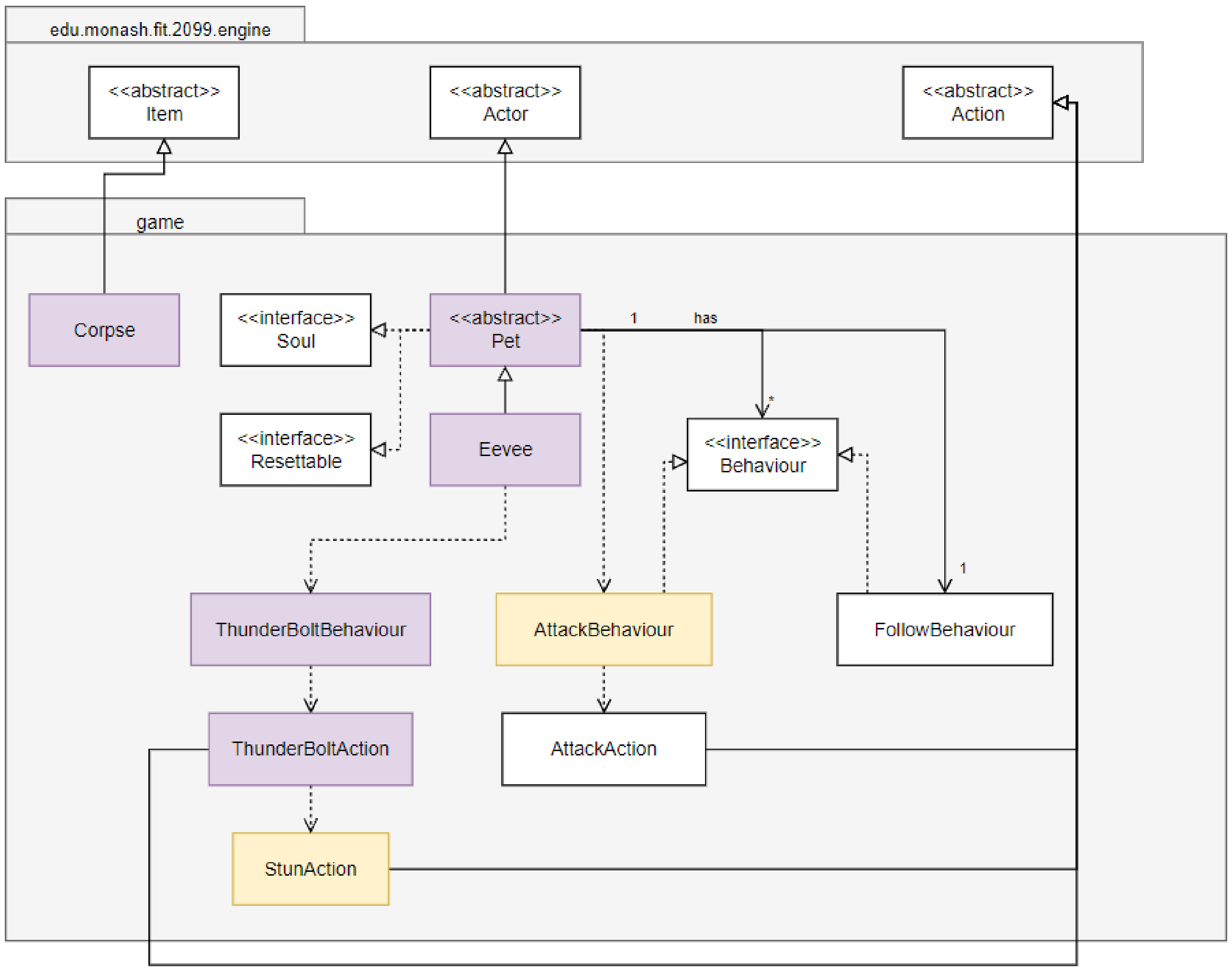
FogDoor

This class contains the logic behind the FogDoor in the game. This class is created for the Player to interact with it and it can move the actor to another map. It extends to the Ground Type as in object-oriented, we treat everything based on the 'abstraction of the real world' which is the logic. The benefits of creating this class are that we can implement and create more features in it in the future. This corresponds to the Open/Closed Principle. For example, we can add more allowable actions for the player to have different options to interact and choose. Besides, it also follows the Single Responsibility Principle as this class is created so that it only contains the functionalities that a fog door can do. Last, we can reuse this class for creating many different fog doors in the game and this corresponds to the DRY principle.

MovePetPlayerAction

This class contains the logic of the actor Pet teleporting to a new map. This class extends to the action class as it is considered a type of action performed by an actor. By doing so, it inherits the necessary methods for the action to execute. Having this class allows us to easily change or add characteristics without affecting the rest of the code. It also allows other actors to inherit this action without having to repeat any code, following the DRY Principle. Additionally, since it only represents the teleportation of the actor Pet to a new map, it has only one responsibility. Hence, this follows the Single Responsibility Principle.

Class Diagram 2



This class diagram shows the relationship between the Pet Class and its functionalities.

New Classes:

<<abstract>> Pet

We create a new abstract class that is named Pet. This class is designed as a parent class for all the pet characters in the game. It extends from the Actor class so it inherits the methods from the Actor class such as the method to add items into inventory and other methods in the Actor Class. It also inherits the Soul and Resettable Class as it is one of the actors on the map. The benefit of creating this abstract Pet class is that we can add features or any methods it needs for all the pet characters in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the pet characters to have the same allowable action that will attack and follow Enemy only, we can implement the getAllowableAction method in the Pet class so that every pet has the allowable action for the AttackAction. This also corresponds to the DRY(Do not Repeat Yourself) principle, which we just need to implement once in the Pet class as all the child classes are extended to the parent Pet class. Dependency Injection has also been implemented for this class so that the pet will always follow the master which is the Player and transfer the soul to its master.

Eevee

This class is used to create an instance of a pet of the type Eevee which is the pet in the game. We created this class as adding or changing the characteristics, status, and ability of this pet can be easily done and it will not affect the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adheres to low coupling. Additionally, we can easily increase the number of the Eevee in the game by just simply re-using this class to create a new Eevee. Having its own class also follows the Single Responsibility Principle as it is only in charge of representing Eevee in the game.

We create a pet as a single new class so that each pet class just has one responsibility which is it contains all the features that the pet needs. This corresponds to the Single Responsibility Principle (SRP) which makes our system easier to maintain and extend in future implementation. This Eevee class will contain all the functionality that the Eevee the pet needed to support its responsibility. This class only contains the attributes that the Eevee has and all attributes are set as private to reduce the connascence. Last, we can easily add features and functionalities that only the Eevee the pet has in its class. For example, Eevee has ThunderBolt Behavior, we can easily add this behavior in the Eevee class. This corresponds to the Open/Closed Principle.

ThunderBoltBehaviour

This class contains the logic behind the behaviour of being able to use ThunderBolt on an actor. This class is considered a type of behaviour hence extending to the Behaviour class as shown in the diagram above. This class generates an action which is the ThunderBoltAction that allows the current actor to stun an actor that is attackable. Creating this class allows for reusability in the future when extending the code which adheres to the principle DRY as it avoids any repeating code from happening. We can easily implement other actors to have this functionality by using this behaviour. Creating this class also follows the Single Responsibility Principle, which will only contain the functionality of the ThunderBolt.

ThunderBoltAction

This class represents the logic behind the ThunderBoltAction. This class is considered a type of action hence extending to the Action class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. Seeing as this class is only responsible for the thunderbolt feature, it follows the Single Responsibility Principle.

Corpse

This class represents the logic behind the Corpse. It extends to the Ground Type as in object-oriented, we treat everything based on the ‘abstraction of the real world’ which is the logic. The benefits of creating this class are that we can implement and create more features in it in the future. This corresponds to the Open/Closed Principle. For example, we can add allowable actions for the player to interact with it. Besides, it also follows the Single Responsibility Principle as this class is created so that it only contains the functionalities that a corpse can do.

Previously Added Classes:

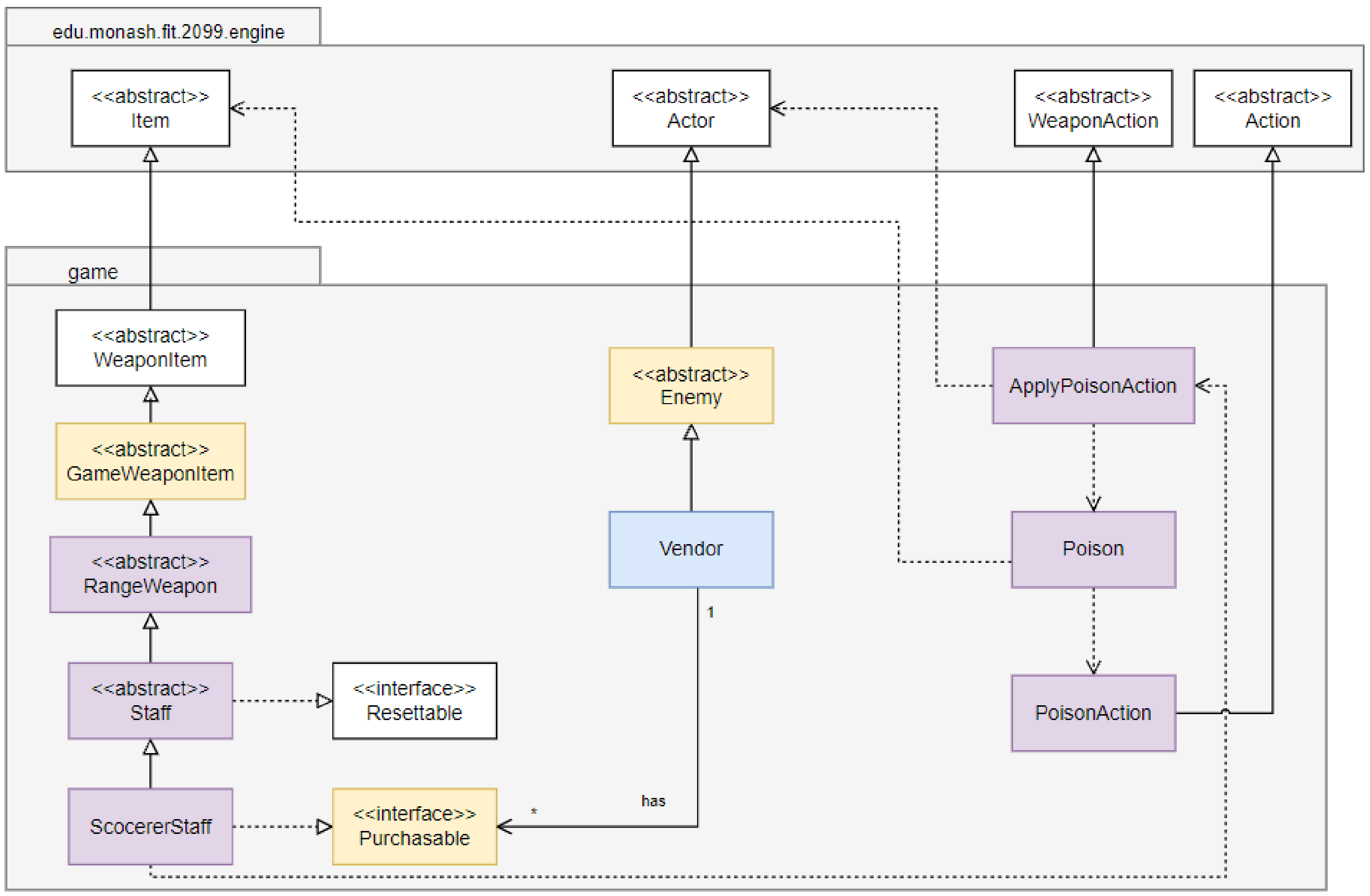
AttackBehaviour

This class contains the logic behind the behaviour of being able to attack an actor. This class generates an action that allows the current actor to attack an actor that is attackable. Creating this class allows for reusability in the future when extending the code which adheres to the principle DRY as it avoids any repeating code from happening. Seeing as it generates an action but is a behaviour, it extends to the Action class and implements the Behaviour interface.

StunAction

This class contains the logic of Yhorm the Giant being stunned after being attacked with the weapon Storm Ruler and the execution of the WindSlashAction. We created this class to stun Yhorm the Giant by adding a capability called ‘STUNNED’ to mimic this action. This class follows the Single Responsibility Principle as it is only responsible for one action that is stunning Yhorm the Giant. Since this is a type of action, this class extends to the Action class.

Class Diagram 3



This class diagram shows the relationship between the Staff Class and its functionalities.

New Classes:

<<abstract>> RangeWeapon

We created a new abstract class that is named RangeWeapon. This class is designed as a parent class for all the ranged weapons in the game. It extends from the GameWeaponItem class so it inherits the methods from the GameWeaponItem which extends from the WeaponItem class such as the method to damage and other methods in the GameWeapon Class. The benefit of creating this abstract RangeWeapon class is that we are able to add features or any methods it needs for all the ranged weapons in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the ranged weapons to have the same features, we can easily implement the relevant method in the RangeWeapon class so that every ranged weapon has the same feature. This also corresponds to the DRY(Do not Repeat Yourself) principle, which we just need to implement once in the RangeWeapon class as all the child classes are extended to the parent RangeWeapon class.

<<abstract>> Staff

This class represents a type of magic weapon that is able to attack from a range. This class is designed as a parent class for all the staves in the game. It extends to the RangeWeapon class to inherit the methods needed in order to execute an attack from a certain range. The benefit of creating this class is that we are able to add new features for all the staves in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the staves to have the same feature which is the sorceries, we can easily implement the relevant method in the Staff class so that every staff has the same feature of sorceries. This also corresponds to the DRY Principle where we just need to implement it once in the Staff class as all the child classes are extended to the parent Staff class.

SorcererStaff

This class is used to create an instance of staff of the type SorcererStaff which is the staff in the game. We created this class as adding or changing the characteristics, status, and ability of this staff can be easily done and it will not affect the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adheres to low coupling. Additionally, we can easily increase the number of the SorcererStaff in the game by just simply re-using this class to create a new SorcererStaff. Having its own class also follows the Single Responsibility Principle as it is only in charge of representing SorcererStaff in the game.

We create a staff as a single new class so that each staff class just has one responsibility which is it contains all the features that the staff needs. This corresponds to the Single Responsibility Principle (SRP) which makes our system easier to maintain and extend in future implementation. This SorcererStaff class will contain all the functionalities that the SorcererStaff needed to support its responsibility. This class only contains the attributes that the SorcererStaff has and all attributes are set as private to reduce the connascence.

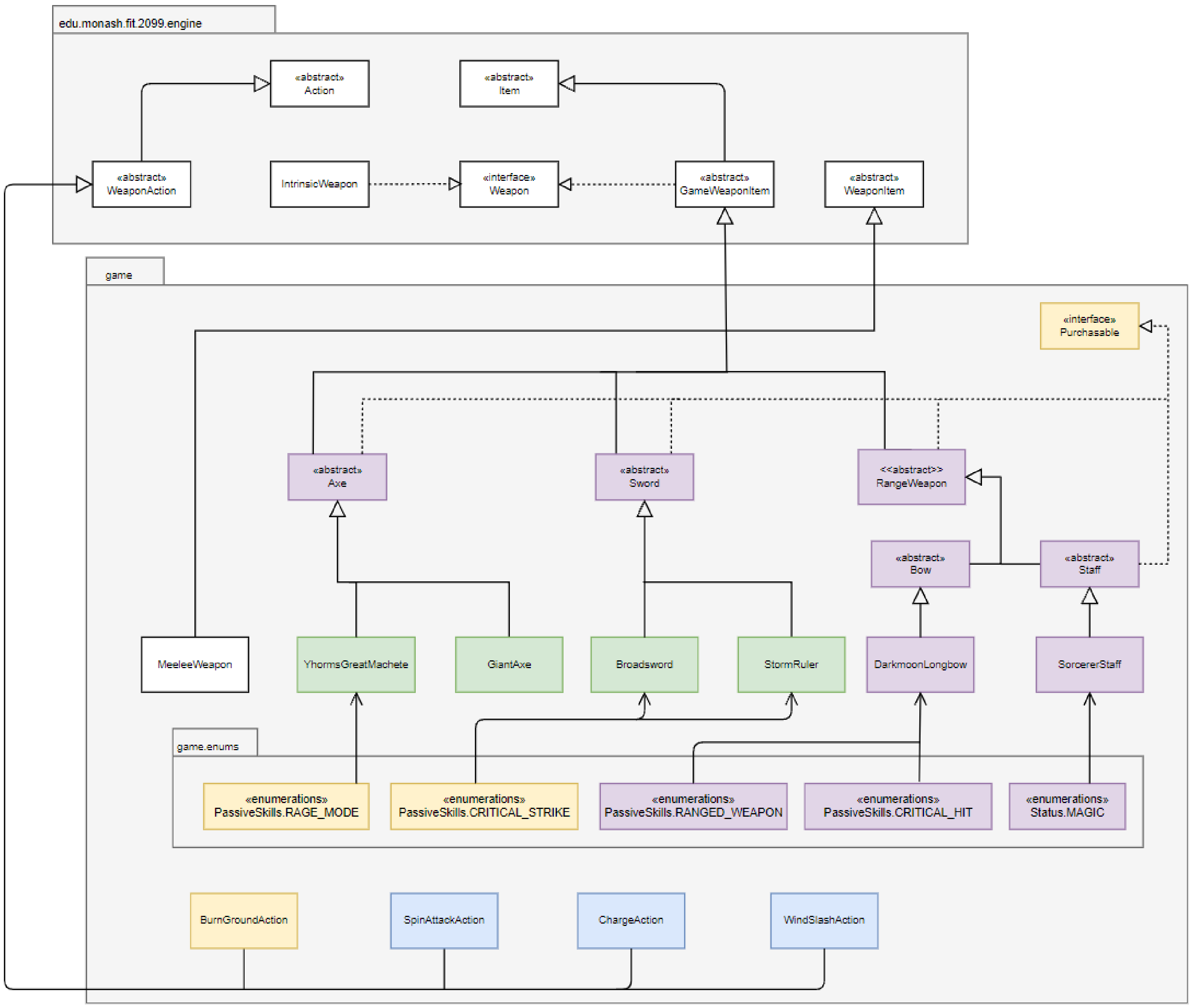
ApplyPoisonAction

This class represents the logic behind the ApplyPoisonAction. This is an allowable action for the SorcererStaff Class. This class extends to the WeaponAction class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. By creating this class, we allow for the possibility of any future weapons to obtain this action which reduces the chances of any repeating code, thus adhering to the DRY Principle and the Dependency Injection Principle. Additionally, this helps with any future extension of the code. We have designed this class to follow the Single Responsibility Principle as it is only responsible for providing the skill of a weapon. Creating this class also follows the Single Responsibility Principle as this class only does the functionality of applying poison features.

Poison
This class represents the logic behind the Poison. It extends to the Item Class as in object-oriented, we treat everything based on the 'abstraction of the real world' which is the logic. The benefits of creating this class are that we can implement and create more features in it in the future. This corresponds to the Open/Closed Principle. For example, we can add allowable actions and tick methods for the actor that has it in their inventory. Besides, it also follows the Single Responsibility Principle as this class is created so that it only contains the functionalities that poison can do.

PoisonAction
This class represents the logic behind the Poison class to do the PoisonAction which is to hurt the actor. This class is considered a type of action hence extending to the Action class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. Seeing as this class is only responsible for the poison and hurt the actor, it follows the Single Responsibility Principle.

Class Diagram 4



This class diagram shows the relationship between the weapons and weapon actions available in the game.

New Classes:

<<abstract>> Axe

This class was created in order to reduce any repeating code seeing as some of the weapons available in the game belong to the same category. By creating this class, we adhere to the DRY Principle as it reduces the possibility of any repeating code. This class will then be the parent class for weapons that are categorized as an ‘Axe’. This is shown with the weapons GiantAxe and YhormsGreatMachete that extend to this class as it is considered a weapon type ‘Axe’. This class extends to the GameWeaponItem class as it is considered a weapon and hence, it is able to inherit all the methods needed for the child classes to implement its unique characteristics. In addition to that, creating this class will allow benefits such as being able to add any additional features in the future easily which makes the process of extending the code much easier.

<<abstract>> Sword

The purpose of creating this class is that it acts as a parent class to the available weapons that are categorized as a ‘sword’. This can be seen with the weapons Broadsword and Storm Ruler that extends to this class as it is considered a weapon type ‘Sword’. By creating this class, we are able to follow the DRY Principle as it reduces the chances of any repeated code. This class extends to the GameWeaponItem class as it is considered a weapon and is able to inherit the necessary methods its child classes require in order to implement its own characteristics. Moreover, having this abstract class allows for easier implementation of new features that can be added in the future without affecting the entire system.

<<abstract>> Bow

This class was created for weapons that are categorized as a weapon type ‘bow’. This can be seen with the weapon Darkmoon Longbow as it extends to this abstract class. By creating this class, not only are we able to reduce any possibility of any repeating code, adhering to the DRY Principle, we are also able to extend the code easily. This includes being able to add new features or characteristics in the future and new weapons that can be considered as a bow. This class extends to the LongRangeWeapon class as it is able to attack actors from a distance. By doing so, it is able to inherit the methods necessary for the child classes to allow for the implementation of its own characteristics as the RangeWeapon further extends to the GameWeaponItem class.

<<abstract>> Staff

We created a new abstract class that is named Staff. This class is designed as a parent class for all the staves in the game. This staff weapon is a magic weapon that can attack enemies from far away. It extends from the RangeWeapon class so it inherits the methods from the RangeWeapon class such as the method to getRange and other methods in the RangeWeapon Class. The benefit of creating this abstract Staff class is that we are able to add features or any methods it needs for all the staves in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the staves to have the same feature which is the sorceries, we can easily implement the relevant method in the Staff class so that every staff has the same feature of sorceries. This also corresponds to the DRY(Do not Repeat Yourself) principle, which we just need to implement once in the Staff class as all the child classes are extended to the parent Staff class.

<<abstract>> RangeWeapon

This class was created to represent weapons that are able to attack an actor from a distance. To be more specific, 3 squares away from where the holder of the weapons is standing which fulfills the requirement mentioned in the specifications. This is an abstract class as there is more than one weapon in the game that has this feature. This is shown as the weapons Darkmoon Longbow and Staff extend to this class. By doing so, these weapons are able to inherit the necessary methods that allow them to attack from a range. This class further extends to the GameWeaponItem class which inherits the methods the child classes require to implement their own unique characteristics. Additionally, having this class will allow us to add new features in the future easily without affecting the rest of the code. This corresponds to the Open/Closed Principle. For example, if we want all the ranged weapons to have the same features, we can easily implement the relevant method in the RangeWeapon class so that every ranged weapon will inherit it.

DarkmoonLongbow

This class was created to represent the weapon of Aldrich The Devourer, a boss in the game. By having its own class, it is able to have its own set of characteristics that may differ from the rest of the weapons. It also allows us to easily add new features in the future if required to do so. Additionally, it also allows for the different actors to equip it if ever needed which will reduce any repeated code, adhering to the DRY Principle. This class extends to the GameWeaponItem class since it is considered a type of weapon. It is then able to inherit the necessary methods to implement its own characteristics. By having its own class, we follow the Single Responsibility

Principle as it is only in charge of representing a weapon that can be used by Aldrich The Devourer.

Previously Added Classes:

Yhorm's Great Machete

This class is used to create a Yhorm's Great Machete object, a weapon used by Yhorm the Giant (Lord of Cinder). We created this class as setting its characteristics and limitations would be much easier and maintainable when doing so. Limitations include it is only available to be equipped by Yhorm the Giant himself. It is considered a weapon of Yhorm the Giant and hence it extends to the GameWeaponItem Class and because a weapon is a type of item in the game, it also further extends to the Item Class.

Broadsword

This class is used to create a Broadsword object in the game where it can be equipped by an actor and used as a weapon. We created this class as adding or changing characteristics of this weapon can be easily done and it will not affect the rest of the system which helps with the maintainability of the code. Implementing the use of this weapon to other types of actors will also mean adhering to the coding practice DRY. Since this is a type of weapon, it extends to the GameWeaponItem Class to inherit the necessary methods in order to implement its unique characteristics which further extends to the Item Class as it is also considered a type of item in the game.

GiantAxe

This class is used to create a GiantAxe object instead. We created this class to allow the process of changes/additional features to be made easier. It will also permit other types of actors to equip this weapon if new game features are to be made in the future without having any repetition of code, adhering to the DRY Principle. This class extends to the GameWeaponItem Class as it is a weapon that can be used by an actor and inherits the necessary methods in order to allow for any form of interaction. This class then further extends to the Item Class as it is a type of item.

StormRuler

This class is used to create a StormRuler object that can be equipped by a player. By creating this class, it can help with the process of adding/changing characteristics provided by the weapon to be much easier and more maintainable. Having its own class allows other actors to equip this weapon if required in the future without having to repeat any code. This follows the DRY Principle as well as the Single Responsibility Principle as it is responsible for representing the weapon itself. It is also considered a weapon as well as a type of item in the game. Hence, StormRuler extends to the GameWeaponItem Class to inherit the necessary methods and then further extends to the Item Class.

BurnGroundAction

This class represents the logic behind the active skill of Yhorm's Great Machete. This class showcases Yhorm's Great Machete's ability to allow its surroundings to be set on fire when being equipped by Yhorm the Giant. This class extends to the Weapon Action class. By having this class, we are able to easily change and edit different characteristics to our liking and for any possible implementations in the future. This class was also created so that the process of extending the code or having new or existing weapons to implement this skill would be much easier. By doing this, we reduce the possibility of repeating codes that adhere to the practice of the DRY principle. Creating this class also follows the Single Responsibility Principle as this class only does one action and represents only one skill of the weapon Yhorm's Great Machete.

<<interface>> Purchasable

This interface was created in order to distinguish whether an item in the game is purchasable or not. This interface allows us to get the number of souls for each weapon when making a transaction with the Vendor. By creating this interface, we follow the Interface Segregation Principle as only items that are considered purchasable implement this interface while others do not. This also helps us to achieve lower coupling as implementing this interface for any additional items will not affect the rest of the system which helps with the maintainability of the system. In addition to that, this interface helps reduce the chances of repeating code as well as the extension of the system as making future items purchasable in the future would only need to implement this interface. This adheres to the principle of DRY.

SpinAttackAction

Changes that were made to this class involves the changing of the name of the class itself from SpinAttack to SpinAttackAction. The reason is that it is considered a type of action. By adding the word ‘Action’ would help to better distinguish which classes are actions and which are not. By creating this class, we allow for the possibility of any future weapons to obtain this action which reduces the chances of any repeating code, thus adhering to the DRY Principle as well as the Dependency Injection Principle. Additionally, this helps with any future extension of the code. Seeing as this class is only responsible for providing the skill of a weapon, it follows the Single Responsibility Principle.

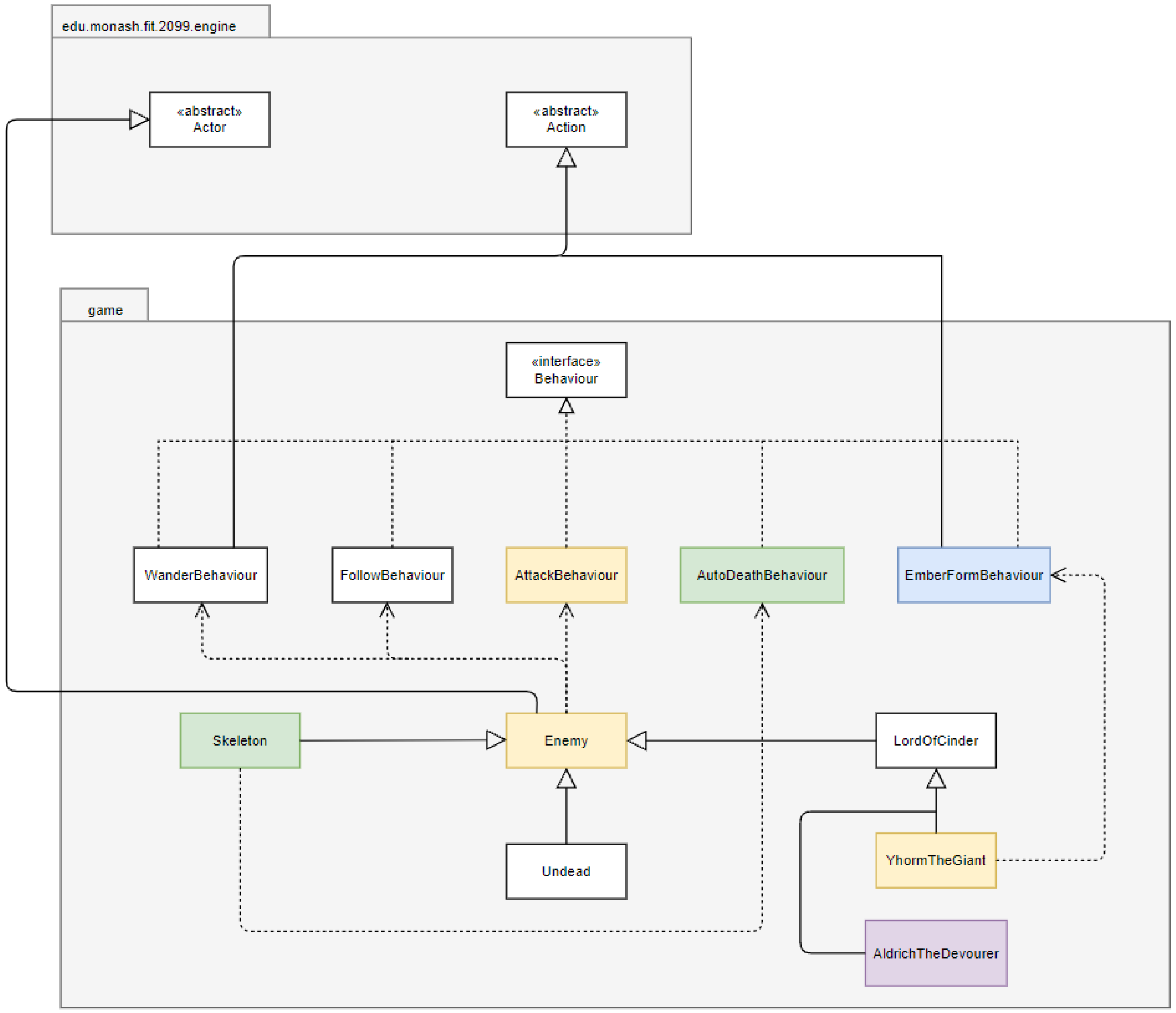
ChargeAction

Changes that were made to this class involves the changing of the name of the class itself from Charge to ChargeAction. The reason is that it is considered a type of action. By adding the word ‘Action’ would help to better distinguish which classes are actions and which are not. Creating this class allows other weapons to obtain this action which reduces the chances of any repeating code from occurring which adheres to the DRY Principle as well as the Dependency Injection Principle. Additionally, helping with any future extension of the system. We have designed this class to follow the Single Responsibility Principle as it is only responsible for providing the skill of a weapon.

WindSlashAction

Changes that were made to this class involves the changing of the name of the class itself from WindSlash to WindSlashAction. The reason is that it is considered a type of action. By adding the word ‘Action’ would help to better distinguish which classes are actions and which are not. By creating this class, it removes any limitation on what weapon can obtain this action, this follows the Dependency Injection Principle. This helps with the reusability of the code and also follows the DRY Principle. We have designed this class to follow the Single Responsibility Principle as it is only responsible for providing the skill of a weapon.

Class Diagram 5



This class diagram shows the relationships between the enemies and their behaviours.

New Classes:

AldrichTheDevourer

This class was created to represent another boss in the game called Aldrich The Devourer. Seeing as this is a boss character, it extends to the LordOfCinder class which further extends to the Enemy class since it is considered an enemy to the Player. By doing so, it is able to inherit the necessary methods and behaviours required by Aldrich The Devourer. Having its own class, allows us to implement this actor's unique characteristic. This adheres to the Single Responsibility Principle as it only represents the character itself. Additionally, if we wanted to have multiple Aldrich The Devourer, we simply needed to create an instance of it instead of having to retype the entire code. This adheres to the DRY Principle as having its own class will result in reducing the chances of having any repeated code.

Previously Added Classes:

Enemy

This class represents all the actors that are categorized as enemies. This class acts as a parent class to the types of enemies and was created as all the different enemies in the game have similar abilities and behaviours. By creating this parent class, we adhere to the Liskov Substitution Principle and the DRY Principle. This class also adheres to the Open/Closed Principle as we are able to add new functionalities to the class without having to modify the existing code. Additionally, new enemies would only have to inherit the parent class in order to receive the necessary behaviours and abilities of an enemy while still being able to have its own characteristics by modifying the child classes. This also reduces the chances of any repeating code from occurring. This class extends to the Actor class and an enemy is a type of actor in the game.

YhormTheGiant

This class represents the enemy Yhorm the Giant that is an extension of the Lord Of Cinder class as it is an enemy boss character in the game. Having its own class would allow it to have different characteristics that may differ from any future enemy bosses. This class has the ability to create the Ember Form Behaviour that only belongs to Yhorm the Giant itself. This class is considered a type of actor that explains the relationships above and ultimately ends with it extending to the Actor class. Having its own class also follows the Single Responsibility Principle as it is only in charge of representing Yhorm the Giant in the game. It also allows for us to create multiple Yhorm The Giants if required without having any repetition of code, adhering to the DRY Principle.

AttackBehaviour

This class contains the logic behind the behaviour of being able to attack an actor. This class generates an action that allows the current actor to attack an actor that is attackable. Creating this class allows for reusability in the future when extending the code which adheres to the principle DRY as it avoids any repeating code from happening. Seeing as it generates an action but is a behaviour, it extends to the Action class and implements the Behaviour interface.

EmberFormBehaviour

The changes in this class include it extending to the Action class while implementing the Behaviour interface. This class contains the logic behind the Ember Form behaviour where the active skill of Yhorm's Great Machete is called and executed. This class was created in order to generate an action where the hit rates of Yhorm the Giant are increased and the surrounding ground around Yhorm the Giant burns. Creating this class promotes reusability which will allow for easy extension of the code in the future. This follows the DRY principle as it reduces the chances of any repeating codes.

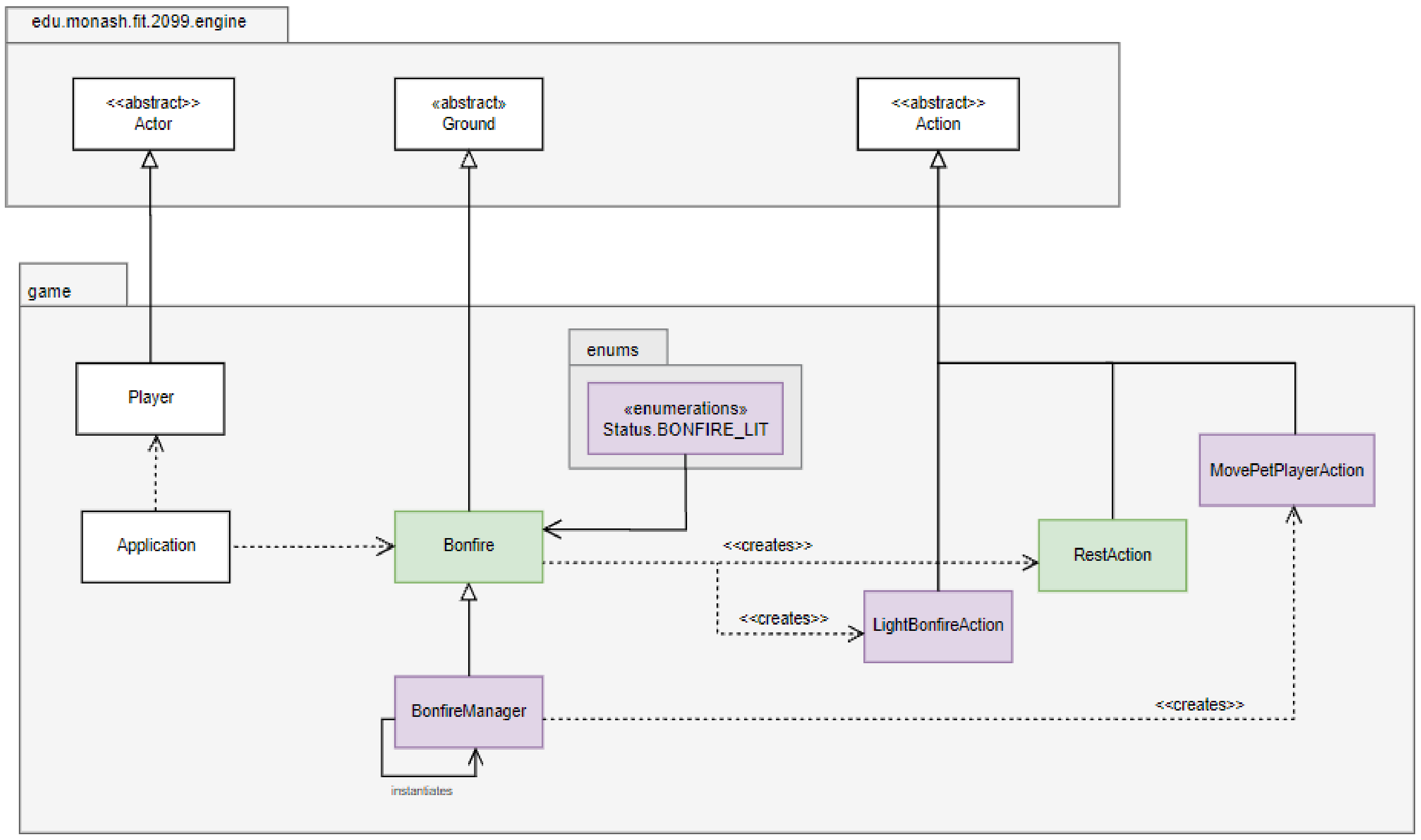
AutoDeathBehaviour

This class was created in order to mimic the auto death behaviour. Creating this class grants us the flexibility to edit and change characteristics of this behaviour without affecting the rest of the system as well as promotes maintainability of the code. It also allows for reusability of the code for other actors or even new ones, if needed in the future which will adhere to the coding practice DRY. This class implements the Behaviour Interface in order to inherit the necessary methods for a behaviour to be executed.

Skeleton

This class is used to create an instance of an enemy of the type Skeleton. We created this class as adding or changing the characteristics can be easily done and it will not affect the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adheres to low coupling. Additionally, we can easily increase the number of the Skeletons in the game by just simply re-using the class to create new Skeletons which adheres to the DRY Principle as there would be no repeated code. It extends the Actor class since it is a type of actor in the game. By doing so, it inherits the methods from the Actor class such as the method to add items into inventory and other methods in the Actor Class.

Class Diagram 6



This class diagram shows the relationships between the Bonfire and all of it's related classes.

New Classes:

BonfireManager

This class contains the logic on how to deal with more than one instance of bonfire in the map(s) and was created to handle the instances of bonfire that exist in the current two maps as well as all future bonfires. This class extends to the Bonfire class in order to inherit all the necessary methods. Since there is always only one instance of BonfireManager throughout the whole game, we use singleton here rather than Dependency Injection as we do not want to overpopulate our constructor since the class itself will handle too many things which will violate the Single Responsibility Principle . The benefit of creating this class is that we can store all of the bonfires in a list within the class and they can be placed anywhere in the map allowing us to implement more features for the Bonfire in the future without having any repeated code. This adheres to the Open/Closed and DRY Principle.

LightBonfireAction

This class represents the logic used to indicate whether a Bonfire is activated or not, and allows a player to respawn back in the campfire after activating it. This class was created in order to relieve the Bonfire class of having too many responsibilities which would violate the Single Responsibility Principle. By creating this class, we are able to add and change any characteristics without affecting the rest of the system adhering to the Open/Closed Principle. This class extends to the action class in order to inherit the necessary methods needed as it is considered a type of action in the game. Additionally, this class follows the Single Responsibility Principle as its only responsibility is to light up bonfires that have been interacted with.

Previously Added Classes:

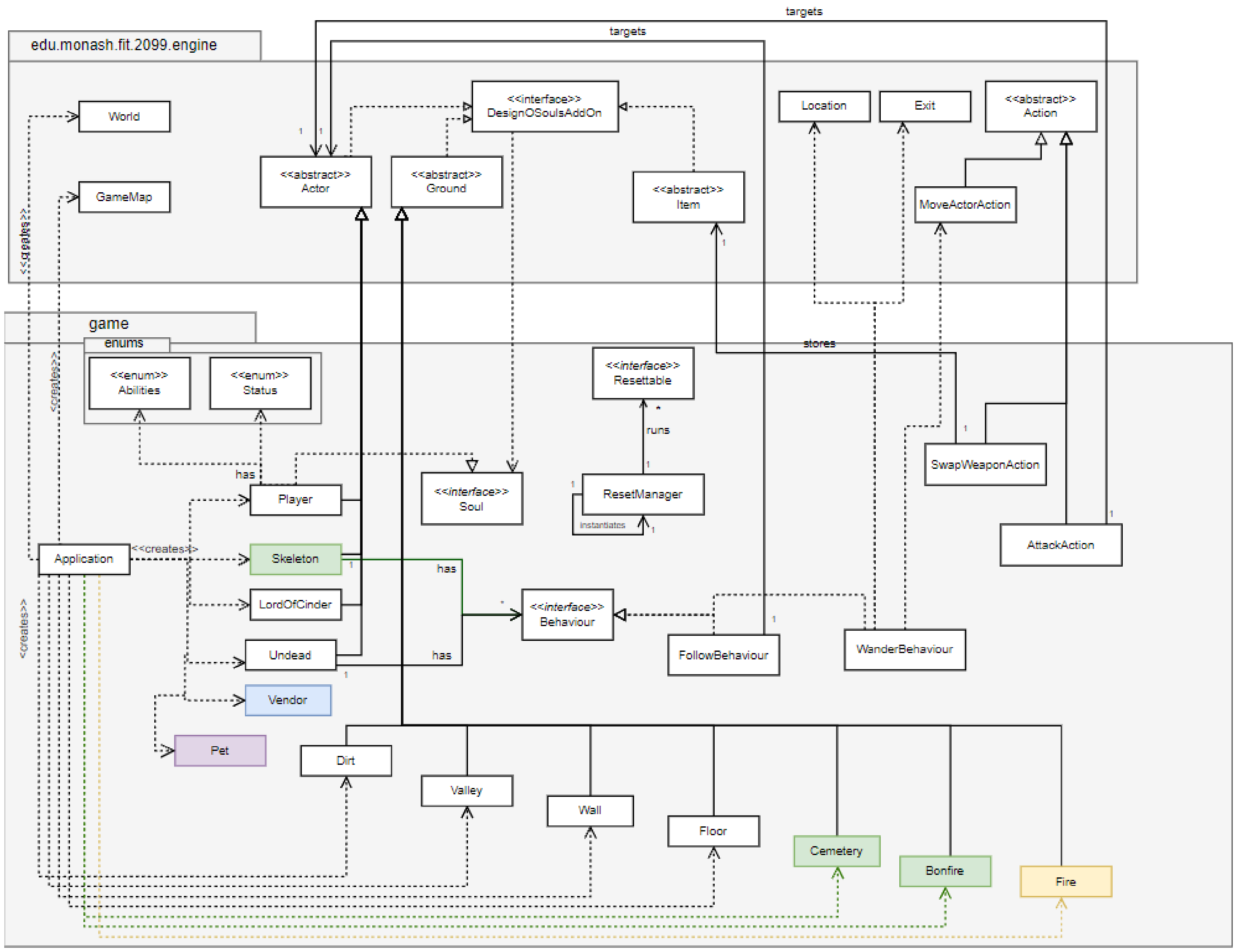
Bonfire

This class is used to create the Bonfire object in the game. We created this class as we can easily change and add characteristics of the Bonfire class while not affecting the rest of the system. We can also create many Bonfire objects with characteristics if needed in the future. Additionally, we can easily increase the number of Bonfire objects on the map by simply re-using this class which follows the DRY Principle since it only uses Application class and BonfireManager, therefore it adheres to low coupling. It extends to the Ground Class as it is considered to be a type of ground and by doing so, it inherits the necessary methods required.

RestAction

This class was created in order to represent the action of resting. This class extends to the Action Class in order to inherit the necessary methods as it is considered a type of action. By creating this class, we are able to extend the code by adding more characteristics as well as allowing other actors to inherit this action in the future without having to repeat any code. By doing so, we adhere to the DRY Principle as well as the Single Responsibility Principle as it is only responsible for the action of resting.

Class Diagram 7



This class diagram shows the relationship between the classes in the engine package and the game package.

Previously Added Classes:

Skeleton

This class is used to create an instance of an enemy of the type Skeleton. We created this class as adding or changing the characteristics can be easily done and it will not affect the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adhere to low coupling. Additionally, we can easily increase the number of the Skeletons in the game by just simply re-using the class to create new Skeletons which adhere to the DRY Principle as there would be no repeated code. It extends the Actor class since it is a type of actor in the game. By doing so, it inherits the methods from the Actor class such as the method to add items into inventory and other methods in the Actor Class.

Cemetery

This class is used to create an instance of the Cemetery class in the game. We created this class so that we can easily add or change the characteristics of the cemetery while not affecting the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adhere to low coupling and the DRY Principle. Additionally, by creating this class we can easily increase the number of the Cemetery objects in the game by re-using this class to create a new one. It extends the Ground Class and by doing so, it inherits the method from the Ground Class as it will stay permanently in the game map.

Bonfire

This class is used to create the Bonfire object in the game. We created this class as we can easily change and add characteristics of the Bonfire class while not affecting the rest of the system. We can also create many Bonfire objects with characteristics if needed in the future. Additionally, we can easily increase the number of Bonfire objects on the map by simply re-using this class which follows the DRY Principle and adheres to low coupling. It extends to the Ground Class as it is considered to be a type of ground and by doing so, it inherits the necessary methods required.

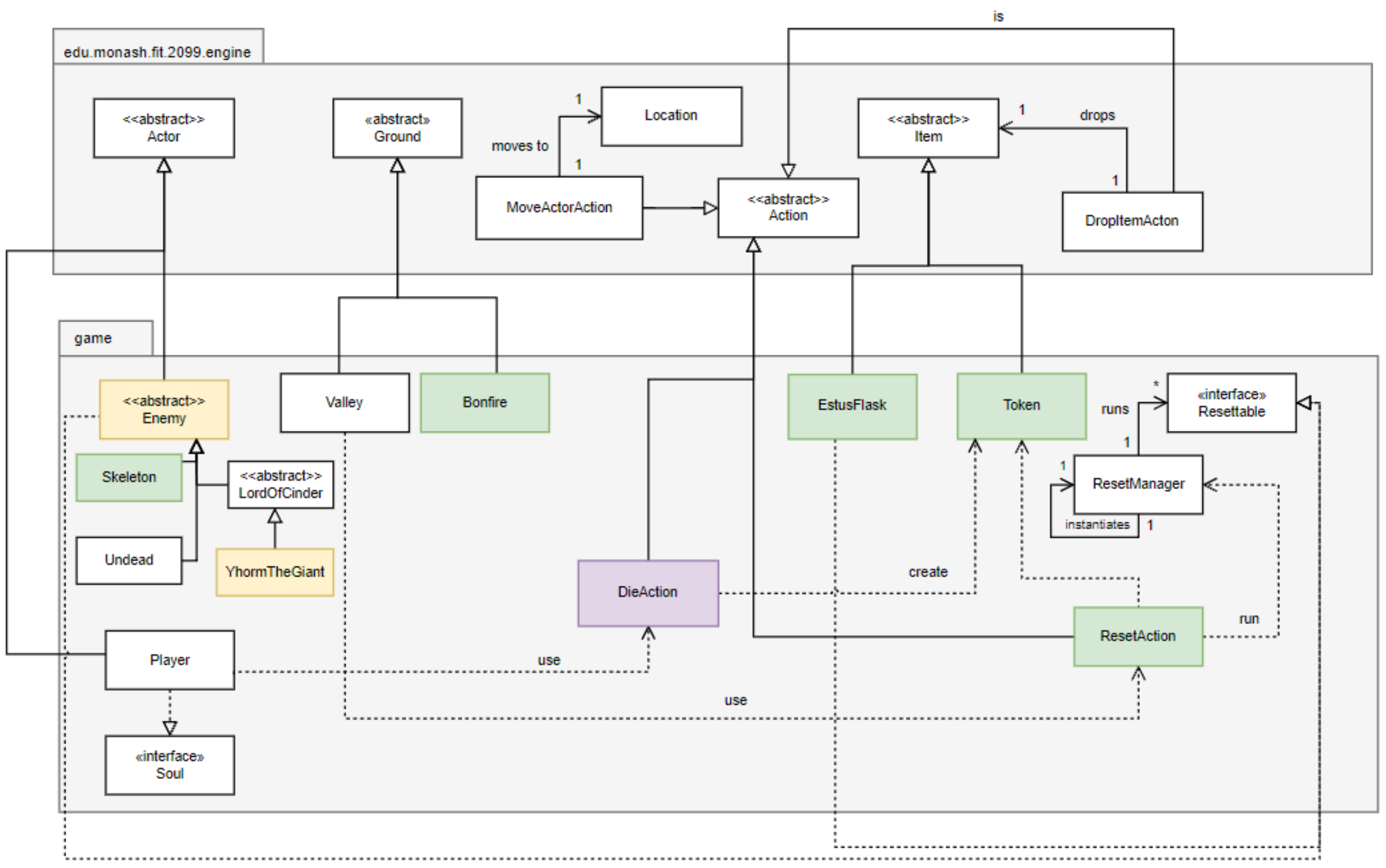
Vendor

This class is used to create the Vendor object in the game. We created this class as we can easily change and add characteristics of the Vendor class while not affecting the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adhere to low coupling. Additionally, we can easily increase the number of Vendor objects in the map by simply re-using this class to create a new Vendor object which follows the DRY Principle. It also follows the Single Responsibility Principle as it is responsible for representing the class itself. It extends to the Ground Class as it is considered to be a type of ground and by doing so, it inherits the necessary methods required.

Fire

This class represents item fire in the game. Creating this class allows us to implement many fire items if ever required so in the future without having to repeat any code. By doing this, we adhere to the DRY Principle as having its own class reduces the chances of any repeating code. It also allows us to add any new characteristics or features in the future without affecting the rest of the code, making the process of extending the code be much more efficient. This class follows the Single Responsibility Principle as it only has one responsibility which is the representation of fire itself.

<u>Class Diagram 8</u>



This class diagram shows the relationship between the Actor class and other classes when an actor dies.

Previously Added Classes:

Skeleton

The Skeleton class is used to create an instance of an enemy of the type Skeleton. We created this class as adding or changing the characteristics, status, and ability of this enemy can be easily done and it will not affect the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adhere to low coupling. Additionally, we can easily increase the number of Skeletons in the game by just simply re-using the class to create new Skeletons.

DieAction

The action death is created by having a DieAction Class. This class is used to focus on the creation of Token and transferring of souls from one actor to Token as well as the move the Player to respawn location. This class extends to the Action Class as it is an action used by the player. By doing so, it will inherit the methods from the Action Class. By creating its own class, we are able to extend the code and allow for other actors in the game to inherit this action without having to repeat any code. This follows the DRY Principle as it reduces the chances of any repeating code from happening.

Bonfire

This class is used to create the Bonfire object in the game. We created this class as we can easily change and add characteristics of the Bonfire class while not affecting the rest of the system. Additionally, we can easily increase the number of Bonfire objects on the map by simply re-using this class which follows the DRY Principle and adheres to low coupling as it reduces any repeated code. It extends to the Ground Class as it is considered to be a type of ground and by doing so, it inherits the necessary methods required.

EstusFlask

This class is used to create an EstusFlask object. The EstusFlask is a health potion that restores the player’s health by 40%. It is an Item that will be used by the Player hence, it extends to the Item Class. By doing so, it inherits the necessary methods from the Item Class. This class was created as it allows for easier implementation of unique characteristics. It follows the Single Responsibility Principle as it is only responsible for representing the item itself. Additionally, we are able to increase the number of Estus flasks if required in the future without having to repeat the entire code which follows the DRY Principle.

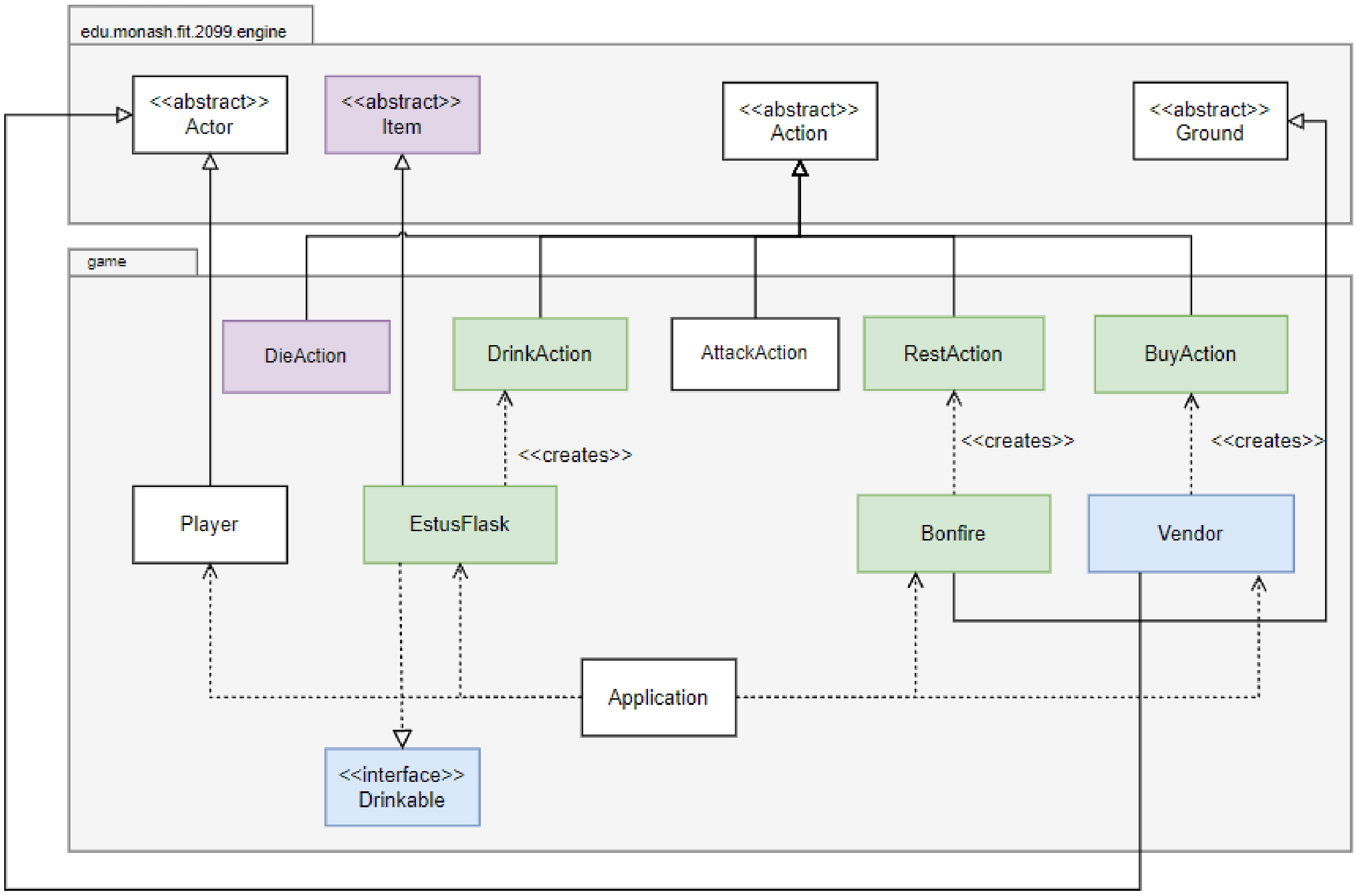
Token

This class is used to create a Token object when the Player dies in the game. We created this class so that we can easily change its characteristics while not affecting the others of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adhere to low coupling and the DRY principle. It extends to Item Class and by doing so, it inherits the methods from the Item class to allow for any form of interaction to happen. This class follows the Single Responsibility Principle as it is in charge of representing the item itself.

ResetAction

This class was created as it is used to execute the ResetManager in the system so that all things considered as Resettables are executed. This class extends to the Action Class as it is considered a type of action. By doing so, it inherits the necessary methods in order for this class to be executed. By creating this class, we are able to allow other actors to inherit this action without having to repeat any code, hence following the DRY Principle. It also follows the Single Responsibility Principle as it is responsible for executing one action.

Class Diagram 9



This class diagram shows the relationships between the Player and its available actions.

Previously Added Classes:

EstusFlask

This class is used to create an EstusFlask object. The EstusFlask is a health potion that restores the player's health by 40%. It is an Item that will be used by the Player hence, it extends to the Item Class. By doing so, it inherits the necessary methods from the Item Class. This class was created as it allows for easier implementation of unique characteristics. It follows the Single Responsibility Principle as it is only responsible for representing the item itself. Additionally, we are able to increase the number of Estus flasks if required in the future without having to repeat the entire code which follows the DRY Principle.

DrinkAction

This class was created to represent the action of drinking. This class extends to the Action Class in order to inherit the necessary methods. By creating this class, we are able to extend the code and allow for other actors to inherit this action without having to repeat any code which follows the DRY Principle. This class also adheres to the Single Responsibility Principle as it is only responsible for one thing that is the action of drinking.

Bonfire

This class is used to create the Bonfire object in the game. We created this class as we can easily change and add characteristics of the Bonfire class while not affecting the rest of the system. Additionally, we can easily increase the number of Bonfire objects on the map by simply re-using this class which follows the DRY Principle and adheres to low coupling as it reduces any repeated code. It extends to the Ground Class as it is considered to be a type of ground and by doing so, it inherits the necessary methods required.

RestAction

This class was created in order to represent the action of resting. This class extends to the Action Class in order to inherit the necessary methods as it is considered a type of action. By creating this class, we are able to extend the code by adding more characteristics as well as allowing other actors to inherit this action in the future without having to repeat any code. By doing so, we adhere to the DRY Principle as well as the Single Responsibility Principle as it is only responsible for the action of resting.

Vendor

This class is used to create the Vendor object in the game. We created this class as we can easily change and add characteristics of the Vendor class while not affecting the rest of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adhere to low coupling. Additionally, we can easily increase the number of Vendor objects in the map by simply re-using this class to create a new Vendor object which follows the DRY Principle. It also follows the Single Responsibility Principle as it is responsible for representing the class itself. It extends to the Ground Class as it is considered to be a type of ground and by doing so, it inherits the necessary methods required.

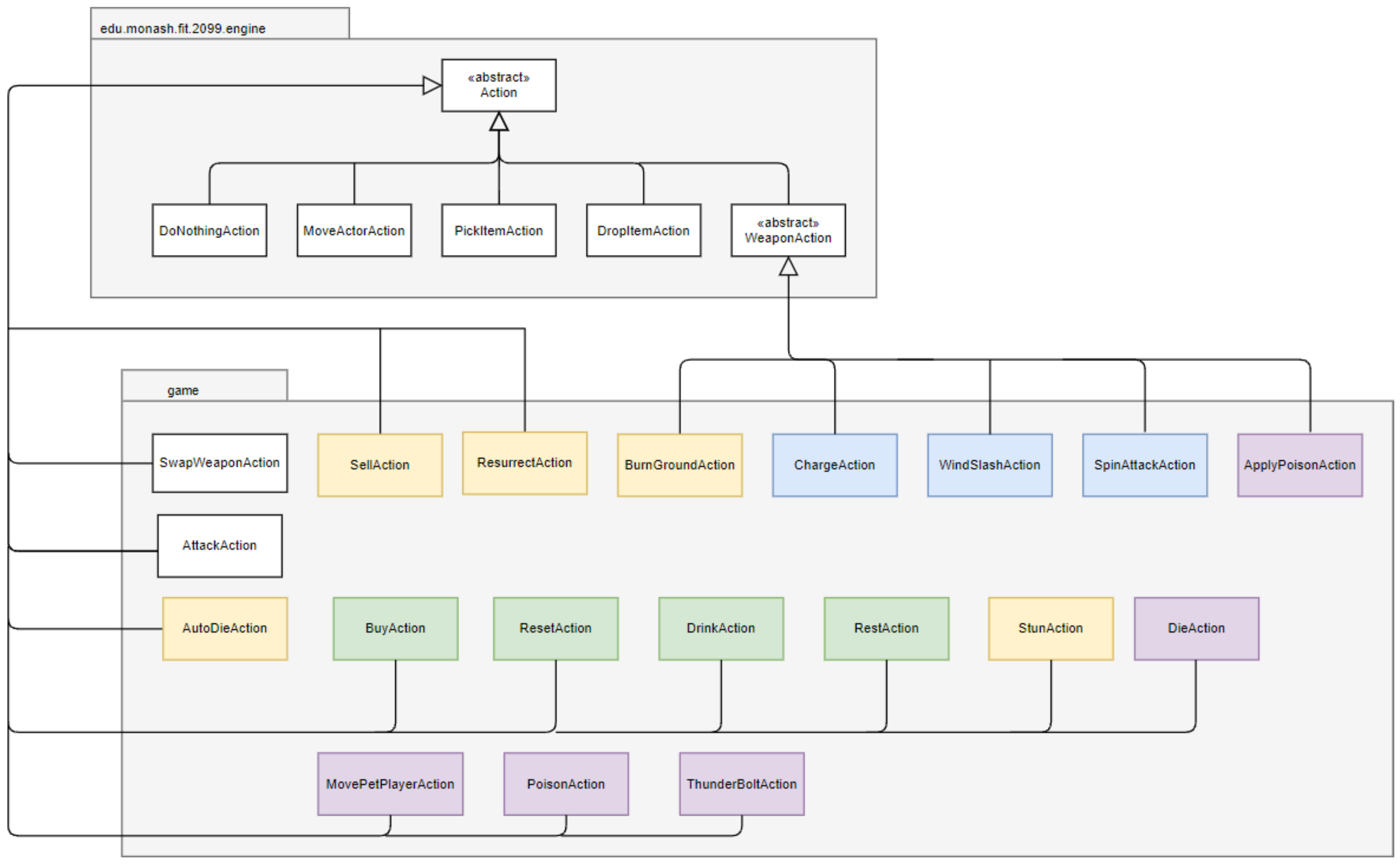
BuyAction

This class was created to represent the action of purchasing an item. This class extends to the Action Class as it is considered a type of action. By doing so, it inherits the Action class and the necessary methods. Having its own class will allow for easier implementation of new characteristics as well as allowing other actors to adopt this action if required in the future. This will then follow the DRY Principle as it reduces the chances of any repeating code. Additionally, this follows the Single Responsibility Principle as it is responsible only for one action.

<<interface>> Drinkable

This interface Drinkable class is created to make the items that implement this interface class a drinkable item. This is to make all the Items that inherit from this class drinkable to an actor. This class was created so that we are able to add new characteristics and features without affecting the rest of the code. By creating this class, we can simply make any item a drinkable item just by implementing this class without having to repeat any code. This follows the DRY Principle as the chances of repeating code will be low.

Class Diagram 10



This class diagram shows the relationships between the different actions available in the game.

New Classes:

ApplyPoisonAction

This class represents the logic behind the ApplyPoisonAction. This is an allowable action for the SorcererStaff Class. This class extends to the WeaponAction class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. By creating this class, we allow for the possibility of any future weapons to obtain this action which reduces the chances of any repeating code, thus adhering to the DRY Principle and the Dependency Injection Principle. Additionally, this helps with any future extension of the code. We have designed this class to follow the Single Responsibility Principle as it is only responsible for providing the skill of a weapon. Creating this class also follows the Single Responsibility Principle as this class only does the functionality of applying poison features.

PoisonAction

This class represents the logic behind the Poison class to do the PoisonAction which is to hurt the actor. This class is considered a type of action hence extending to the Action class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. Seeing as this class is only responsible for the poison and hurt the actor, it follows the Single Responsibility Principle.

DieAction

The action death is created by having a DieAction Class. This class is used to focus on the creation of Token and transferring of souls from one actor to Token as well as the move the Player to respawn location. This class extends to the Action Class as it is an action used by the player. By doing so, it will inherit the methods from the Action Class. By creating its own class, we are able to extend the code and allow for other actors in the game to inherit this action without having to repeat any code. This follows the DRY Principle as it reduces the chances of any repeating code from happening.

MovePetPlayerAction

This class contains the logic of the actor Pet teleporting to a new map. This class extends to the action class as it is considered a type of action performed by an actor. By doing so, it inherits the necessary methods in order for the action to execute. Having this class allows us to easily change or add characteristics without affecting the rest of the code. It also allows other actors to inherit this action without having to repeat any code, following the DRY Principle. Additionally, since it only represents the teleportation of the actor Pet to a new map, it has only one responsibility. Hence, this follows the Single Responsibility Principle.

ThunderBoltAction

This class represents the logic behind the ThunderBoltAction. This class is considered a type of action hence extending to the Action class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. Seeing as this class is only responsible for the thunderbolt feature, it follows the Single Responsibility Principle.

Previously Added Classes:

AutoDieAction

This class represents the logic behind the auto death characteristic of the enemy type Undead. This class fulfills the characteristic of the Undead by having a 10% chance of dying each turn. This class extends to the Action class as it is considered a type of action for the Undead. We created this class as it will allow for an easier process of extending the code in the future as well as allowing other actors to inherit such characteristics. By doing this, we follow the DRY principle as it will avoid any possible repeating code as well as the Single Responsibility Principle as this class is only responsible for the possibility of the Undead dying at each turn.

SellAction

This class represents the logic behind the selling of items performed by the Vendor. This class displays a description to allow the user to know when it is making a transaction with the Vendor. This class extends to the Action class as it is an action performed by the Vendor. By creating this class, it adheres to the Single Responsibility Principle as it only has one job in the system.

StunAction

This class contains the logic of Yhorm the Giant being stunned after being attacked with the weapon Storm Ruler and the execution of the WindSlashAction. We created this class in order to stun Yhorm the Giant by adding a capability called ‘STUNNED’ in order to mimic this action. This class follows the Single Responsibility Principle as it is only responsible for one action that is stunning Yhorm the Giant. Since this is a type of action, this class extends to the Action class.

BurnGroundAction

This class represents the logic behind the ground burning whenever Yhorm The Giant goes into its ember mode as well as showcase Yhorm’s Great Machete’s ability to allow its surroundings to be set on fire when being equipped by Yhorm the Giant This class makes sure to check it’s surrounding ground and continuously burn the ground that is surrounding Yhorm The Giant. This class extends to the Weapon Action class. We created this class as it will allow for an easier process of extending the code in the future as well as allowing other actors to inherit such characteristics such as the other Lord of Cinders that wish to use this class. By doing this, we follow the DRY principle as it will avoid any possible repeating code as well as the Single Responsibility Principle as this class is only responsible for burning the ground surrounding Yhorm The Giant each turn.

ResurrectAction

This class represents the logic behind the ability of the enemy Undead to be able to respawn from the Cemeteries. This class is considered a type of action hence extending to the Action class as shown in the diagram above. Having its own class enables us to make the process of extending the code much easier which can help minimize the chances of bugs from occurring. Seeing as this class is only responsible for the resurrection of the Undead, it follows the Single Responsibility Principle.

SpinAttackAction

We decided to change the name of the class from SpinAttack to SpinAttackAction. We decided that this was best because it is considered a type of action after all. By adding the word ‘Action’, it would help to better distinguish which classes are actions and which are not. By creating this class, we allow for the possibility of any future weapons to obtain this action which reduces the chances of any repeating code, thus adhering to the DRY Principle and the Dependency Injection Principle. Additionally, this helps with any future extension of the code. We have designed this class to follow the Single Responsibility Principle as it is only responsible for providing the skill of a weapon.

ChargeAction

We decided to change the name of the class from Charge to ChargeAction. We decided that this was best because it is considered a type of action after all. By adding the word ‘Action’ would help to better distinguish which classes are actions and which are not. Creating this class allows other weapons to obtain this action, this follows the Dependency Injection Principle. It also reduces the chances of any repeating code from occurring hence, adhering to the DRY Principle. Additionally, helping with any future extension of the system. This class is designed in such a way that it adheres to the Single Responsibility Principle as it is only in charge of providing the skill of a weapon.

WindSlashAction

We decided to change the name of the class from WindSlash to WindSlashAction.We decided that this was best because it is considered a type of action after all. By adding the word ‘Action’ would help to better distinguish which classes are actions and which are not. By creating this class, it removes any limitation on what weapon can obtain this action hence, following the Dependency Injection Principle. This helps with the reusability of the code and also follows the DRY Principle. We have designed this class to follow the Single Responsibility Principle as it is only responsible for providing the skill of a weapon.

BuyAction

This class was created to represent the action of purchasing an item. This class extends to the Action Class as it is considered a type of action. By doing so, it inherits the Action class and the necessary methods. Having its own class will allow for easier implementation of new characteristics as well as allowing other actors to adopt this action if required in the future. This will then follow the DRY Principle as it reduces the chances of any repeating code. Additionally, this follows the Single Responsibility Principle as it is responsible only for one action.

ResetAction

This class was created as it is used to execute the ResetManager in the system so that all things considered as Resettables are executed. This class extends to the Action Class as it is considered a type of action. By doing so, it inherits the necessary methods in order for this class to be executed. By creating this class, we are able to allow other actors to inherit this action without having to repeat any code, hence following the DRY Principle. It also follows the Single Responsibility Principle as it is responsible for executing one action.

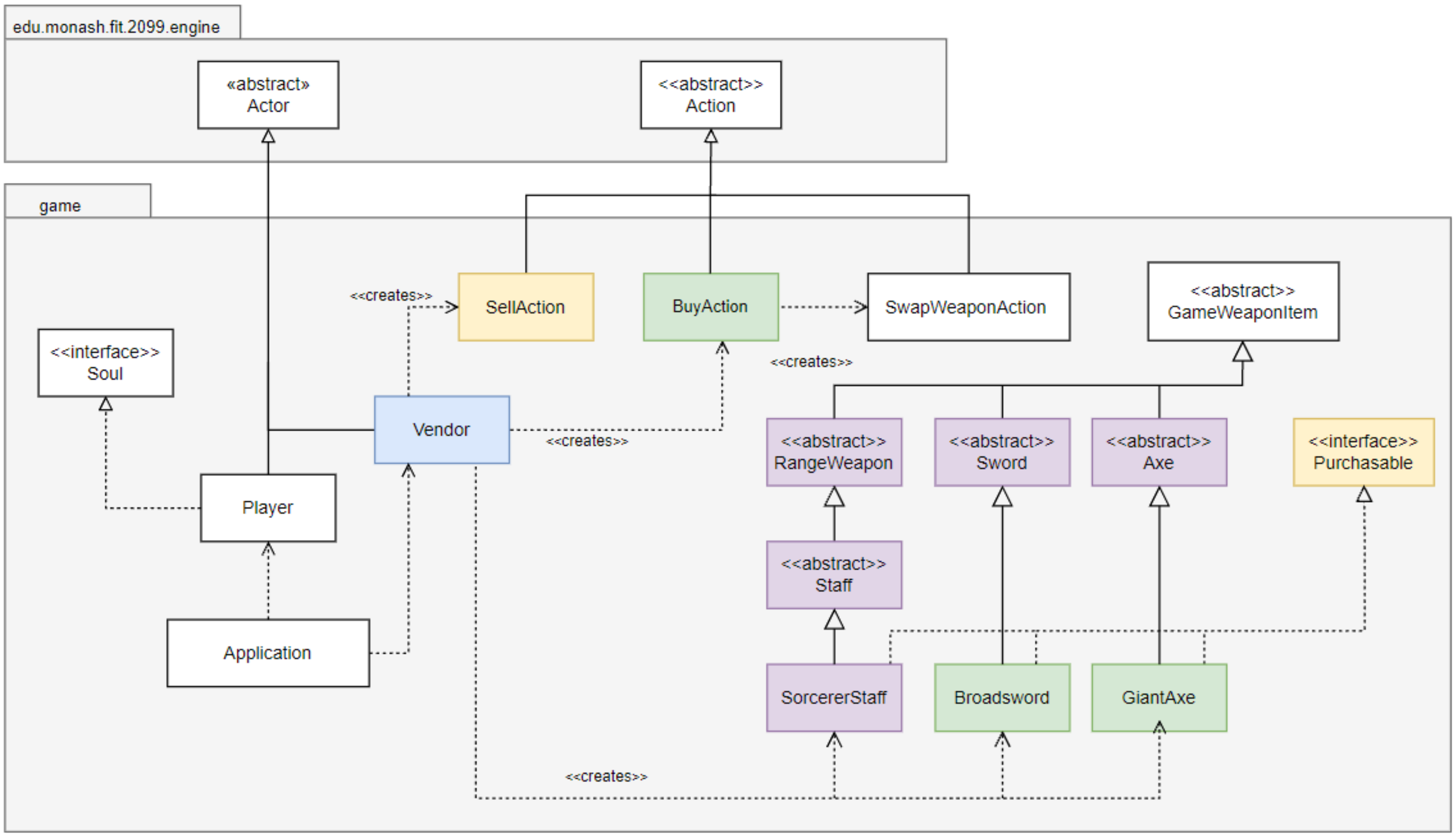
RestAction

This class was created in order to represent the action of resting. This class extends to the Action Class in order to inherit the necessary methods as it is considered a type of action. By creating this class, we are able to extend the code by adding more characteristics as well as allowing other actors to inherit this action in the future without having to repeat any code. By doing so, we adhere to the DRY Principle as well as the Single Responsibility Principle as it is only responsible for the action of resting.

DrinkAction

This class was created to represent the action of drinking. This class extends to the Action Class in order to inherit the necessary methods. By creating this class, we are able to extend the code and allow for other actors to inherit this action without having to repeat any code which follows the DRY Principle. This class also adheres to the Single Responsibility Principle as it is only responsible for one thing that is the action of drinking.

Class Diagram 11



This class diagram shows the relationships between Vendor and its actions.

New Classes:

<<abstract>> Axe

This class was created in order to reduce any repeating code seeing as some of the weapons available in the game belong to the same category. By creating this class, we adhere to the DRY Principle as it reduces the possibility of any repeating code. This class will then be the parent class for weapons that are categorized as an ‘axe’. This is shown with the weapons GiantAxe and YhormsGreatMachete that extend to this class as it is considered a weapon type ‘Axe’. This class extends to the GameWeaponItem class as it is considered a weapon and hence, it is able to inherit all the methods needed for the child classes to implement its unique characteristics. In addition to that, creating this class will allow benefits such as being able to add any additional features in the future easily which makes the process of extending the code much easier.

<<abstract>> Sword

The purpose of creating this class is that it acts as a parent class to the available weapons that are categorized as a ‘sword’. This can be seen with the weapons Broadsword and Storm Ruler that extends to this class as it is considered a weapon type ‘Sword’. By creating this class, we are able to follow the DRY Principle as it reduces the chances of any repeated code. This class extends to the GameWeaponItem class as it is considered a weapon and is able to inherit the necessary methods its child classes require in order to implement its own characteristics. Moreover, having this abstract class allows for easier implementation of new features that can be added in the future without affecting the entire system.

<<abstract>> Staff

This class represents a type of magic weapon that is able to attack from a range. This class is designed as a parent class for all the staves in the game. It extends to the RangeWeapon class to inherit the methods needed in order to execute an attack from a certain range. The benefit of creating this class is that we are able to add new features for all the staves in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the staves to have the same feature which is the sorceries, we can easily implement the relevant method in the Staff class so that every staff has the same feature of sorceries. This also corresponds to the DRY Principle where we just need to implement it once in the Staff class as all the child classes are extended to the parent Staff class.

<<abstract>> RangeWeapon

We created a new abstract class that is named RangeWeapon. This class is designed as a parent class for all the ranged weapons in the game. It extends from the GameWeaponItem class so it inherits the methods from the GameWeaponItem which extends from the WeaponItem class such as the method to damage and other methods in the GameWeapon Class. The benefit of creating this abstract RangeWeapon class is that we are able to add features or any methods it needs for all the ranged weapons in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the ranged weapons to have the same features, we can easily implement the relevant method in the RangeWeapon class so that every ranged weapon has the same feature. This also corresponds to the DRY(Do not Repeat Yourself) principle, which we just need to implement once in the RangeWeapon class as all the child classes are extended to the parent RangeWeapon class.

Previously Added Classes:

SellAction

This class is used to represent the logic of selling items to the Vendor from the players. This class will then display a short description in order to let the user know whenever an item has already been sold. We decided that it was wise to extend this class to the Action class as it is considered a type of action that would be performed by the Vendor. Therefore, by creating this class it follows the Single Responsibility Principle since it has only one responsibility throughout the system.

<<interface>> Purchasable

This interface was created in order to distinguish whether an item in the game is purchasable or not. This interface allows us to get the number of souls for each weapon when making a transaction with the Vendor. By creating this interface, we follow the Interface Segregation Principle as only items that are considered purchasable implement this interface while others do not. This also helps us to achieve lower coupling as implementing this interface for any additional items will not affect the rest of the system which helps with the maintainability of the system. In addition to that, this interface helps reduce the chances of repeating code as well as the extension of the system as making future items purchasable in the future would only need to implement this interface. This adheres to the principle of DRY.

Vendor

This class contains the logic behind the Vendor in the game. The changes made to this class is that it now extends to the Actor class where before it was extended to the Ground class. The reason for this change is that the Vendor class creates an action that can be performed by the Player as well as an action that can be performed by itself. By creating this class, we are able to set different characteristics or add new items that can be sold without affecting the rest of the system. This helps to keep coupling low throughout the system and makes the system more maintainable. In addition to that, we are able to add more items that can be sold by the Vendor only if the items implement the Purchasable interface, this follows the Dependency Inversion

Principle. Additionally, we can easily increase the number of Vendors if needed in the future while simply re-using the class to create new instances of itself. By doing this, we adhere to the DRY principle as it reduces the chances of any repeating code.

BuyAction

This class was created to represent the action of purchasing an item. This class extends to the Action Class as it is considered a type of action. By doing so, it inherits the Action class and the necessary methods. Having its own class will allow for easier implementation of new characteristics as well as allowing other actors to adopt this action if required in the future. This will then follow the DRY Principle as it reduces the chances of any repeating code. Additionally, this follows the Single Responsibility Principle as it is responsible only for one action.

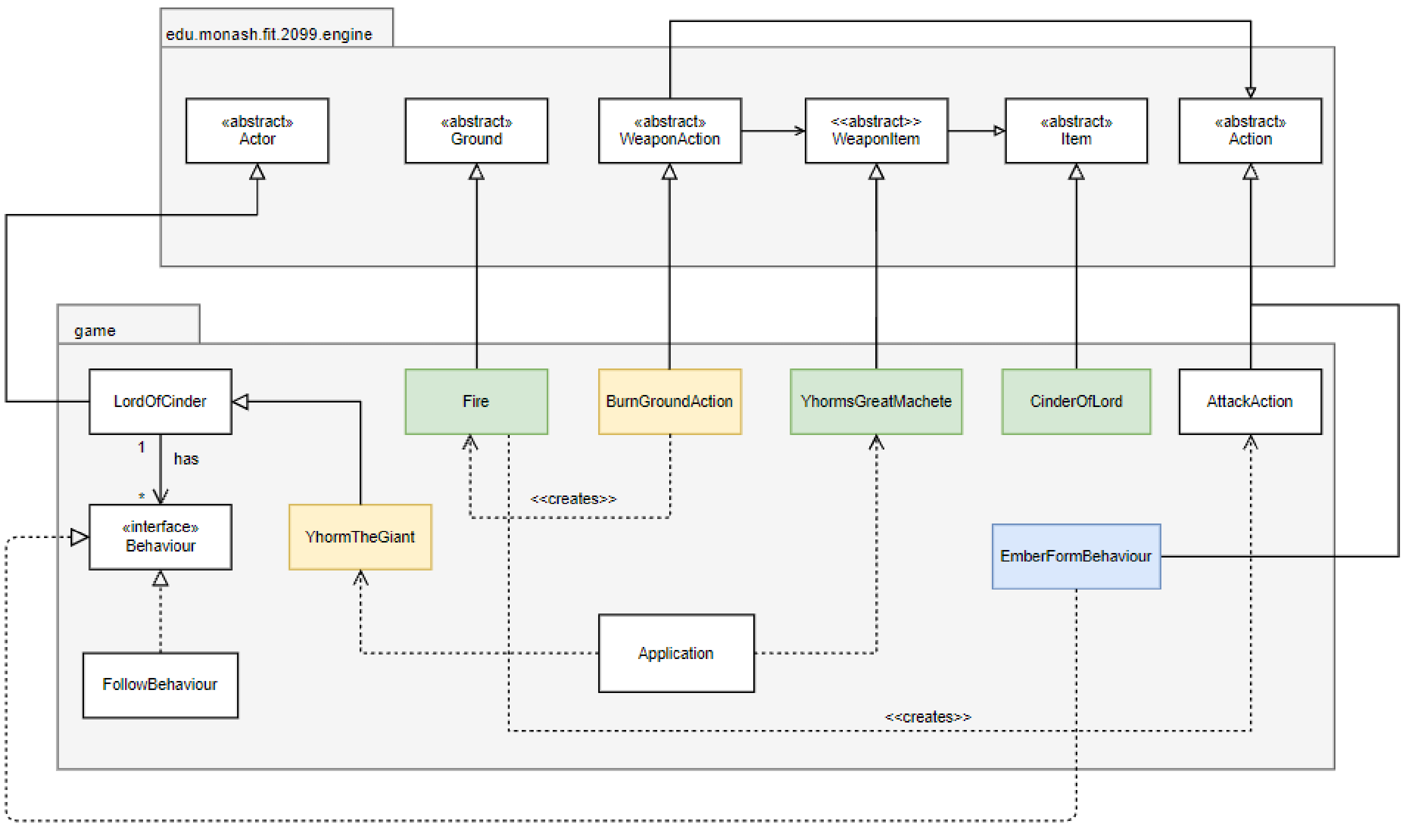
Broadsword

This class is used to create a Broadsword object in the game where it can be equipped by an actor and used as a weapon. We created this class as adding or changing characteristics of this weapon can be easily done and it will not affect the rest of the system which helps with the maintainability of the code. Implementing the use of this weapon to other types of actors will also mean adhering to the coding practice DRY. Since this is a type of weapon, it extends to the GameWeaponItem Class to inherit the necessary methods in order to implement its unique characteristics which further extends to the Item Class as it is also considered a type of item in the game.

GiantAxe

This class is used to create a GiantAxe object instead. We created this class to allow the process of changes/additional features to be made easier. It will also permit other types of actors to equip this weapon if new game features are to be made in the future without having any repetition of code, adhering to the DRY Principle. This class extends to the GameWeaponItem Class as it is a weapon that can be used by an actor and inherits the necessary methods in order to allow for any form of interaction. This class then further extends to the Item Class as it is a type of item.

Class Diagram 12



This class diagram shows the relationship between Yhorm The Giant and its related classes.

Previously Added Classes:

Fire

This class represents item fire in the game. Creating this class allows us to implement many fire items if ever required so in the future without having to repeat any code. By doing this, we adhere to the DRY Principle as having its own class reduces the chances of any repeating code. It also allows us to add any new characteristics or features in the future without affecting the rest of the code, making the process of extending the code be much more efficient. This class follows the Single Responsibility Principle as it only has one responsibility which is the representation of fire itself.

YhormTheGiant

This class represents the enemy Yhorm the Giant that is an extension of the Lord Of Cinder class as it is an enemy boss character in the game. Having its own class would allow it to have different characteristics that may differ from any future enemy bosses. This class has the ability to create the Ember Form Behaviour that only belongs to Yhorm the Giant itself. This class is considered a type of actor that explains the relationships above and ultimately ends with it extending to the Actor class. Having its own class also follows the Single Responsibility Principle as it is only in charge of representing Yhorm the Giant in the game. It also allows for us to create multiple Yhorm The Giants if required without having any repetition of code, adhering to the DRY Principle.

EmberFormBehaviour

The changes in this class include it extending to the Action class while implementing the Behaviour interface. This class contains the logic behind the Ember Form behaviour where the active skill of Yhorm's Great Machete is called and executed. This class was created in order to generate an action where the hit rates of Yhorm the Giant are increased and the surrounding ground around Yhorm the Giant burns. Creating this class promotes reusability which will allow for easy extension of the code in the future. This follows the DRY principle as it reduces the chances of any repeating codes.

BurnGroundAction

This class represents the logic behind the ground burning whenever Yhorm The Giant goes into its ember mode as well as showcase Yhorm's Great Machete's ability to allow its surroundings to be set on fire when being equipped by Yhorm the Giant This class makes sure to check it's surrounding ground and continuously burn the ground that is surrounding Yhorm The Giant. This class extends to the Weapon Action class. We created this class as it will allow for an easier process of extending the code in the future as well as allowing other actors to inherit such characteristics such as the other Lord of Cinders that wish to use this class. By doing this, we follow the DRY principle as it will avoid any possible repeating code as well as the Single Responsibility Principle as this class is only responsible for burning the ground surrounding Yhorm The Giant each turn.

Yhorm's Great Machete

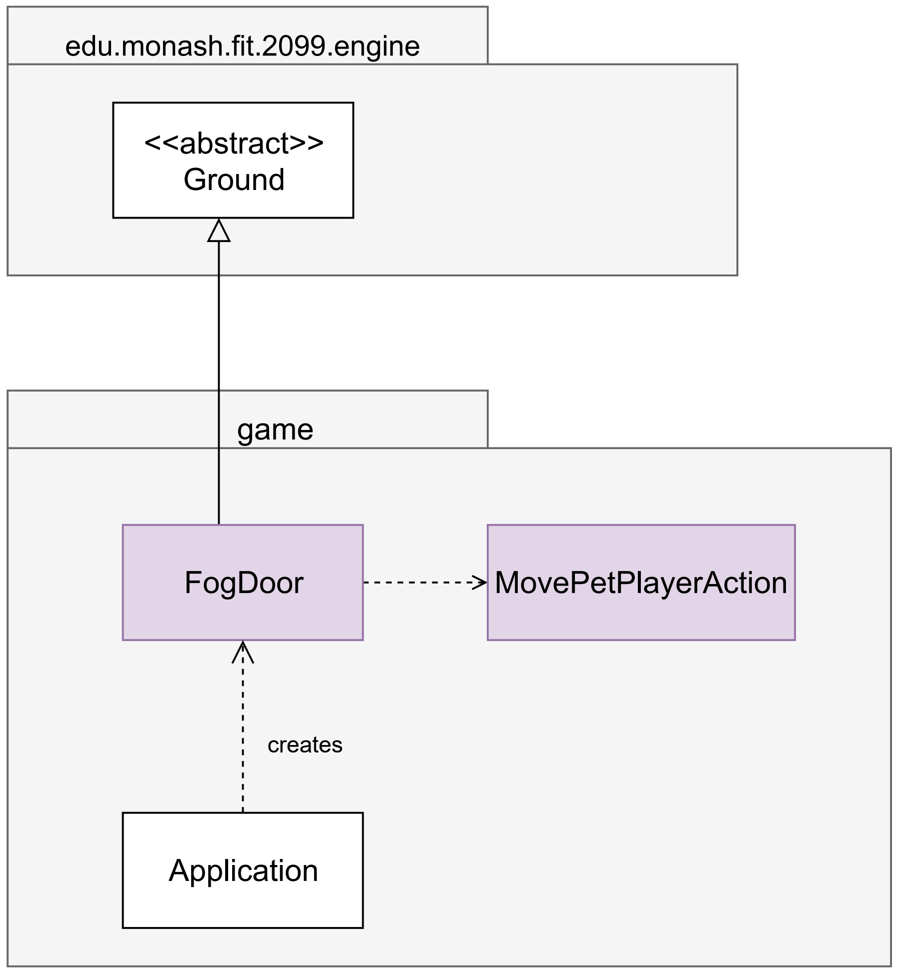
This class is used to create a Yhorm's Great Machete object, a weapon used by Yhorm the Giant (Lord of Cinder). We created this class as setting its characteristics and limitations would be much easier and maintainable when doing so. Limitations include it being only available to be equipped by Yhorm the Giant himself. Creating this class allows for not only Yhorm The Giant itself to equip it but for other actors as well without having any repeated codes It is considered a weapon of Yhorm the Giant and hence it extends to the WeaponItem Class to inherit the necessary methods and because a weapon is a type of item in the game, it also further extends to the Item Class.

CinderOfLord

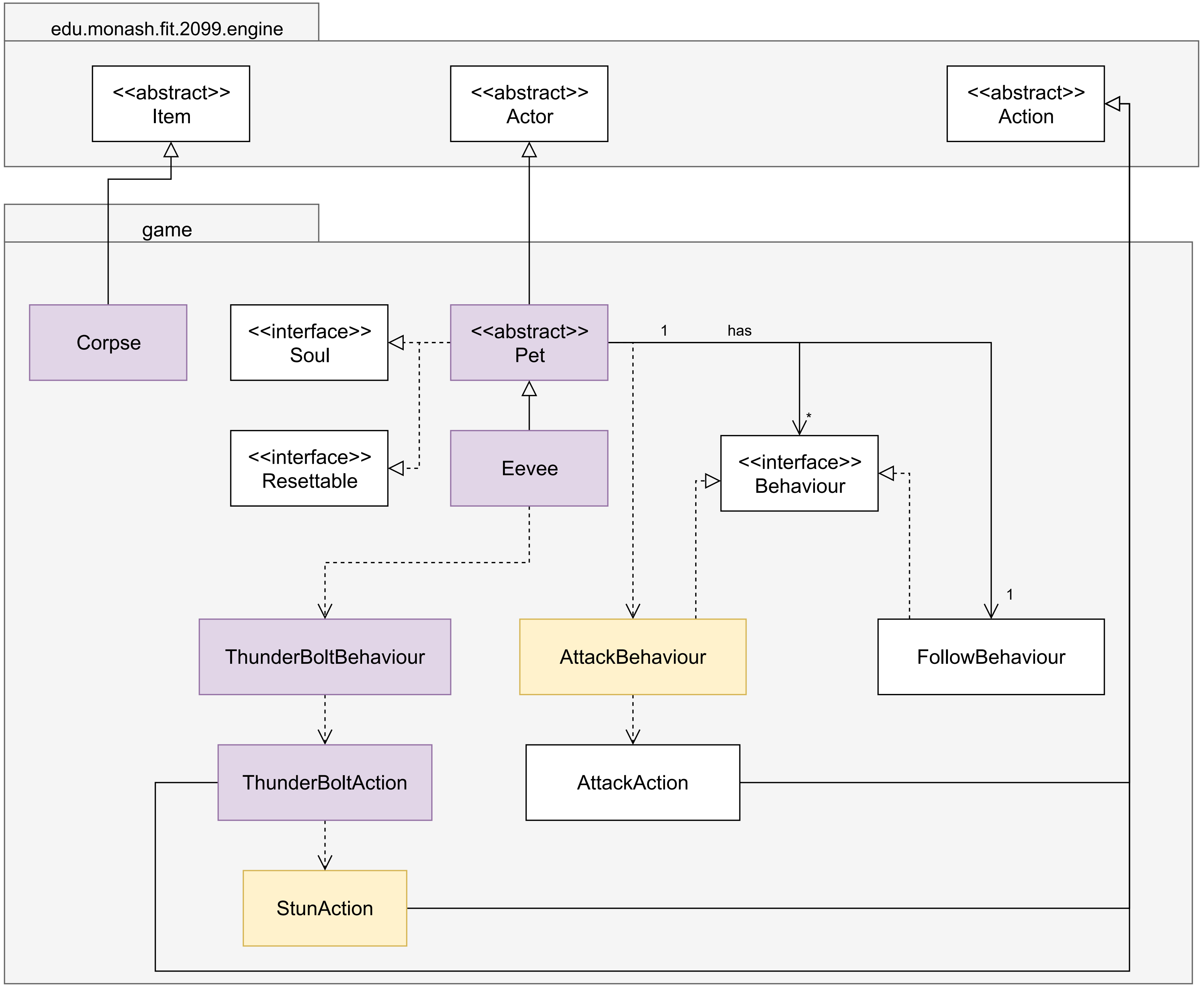
This class was created to represent an item dropped by a Lord Of Cinder after its death. In the event that we have a new Lord Of Cinder character, we are able to allow this new actor to inherit this item without any repetition of code. This also applies to any actors in the future hence, this follows the DRY Principle. Additionally, we are able to add new features and characteristics to this class without affecting the rest of the system. Since it is considered a type of item, it extends to the Item class in order to inherit the necessary methods to represent an item in the game. This class also follows the Single Responsibility Principle as it has only one responsibility which is the representation of this object itself.

UML Diagrams

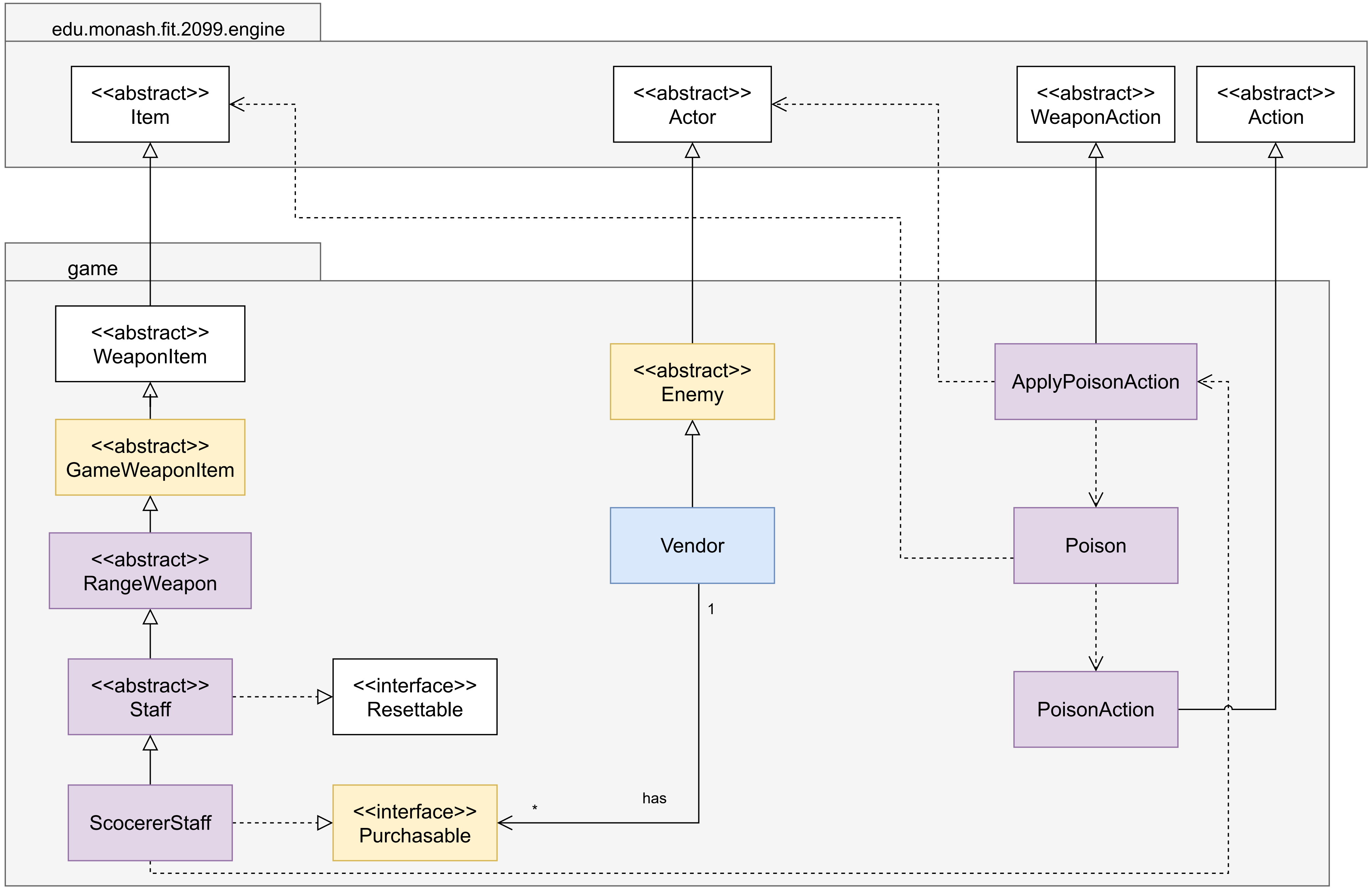
This class diagram shows the relationship between the FogDoor class and its allowable action.



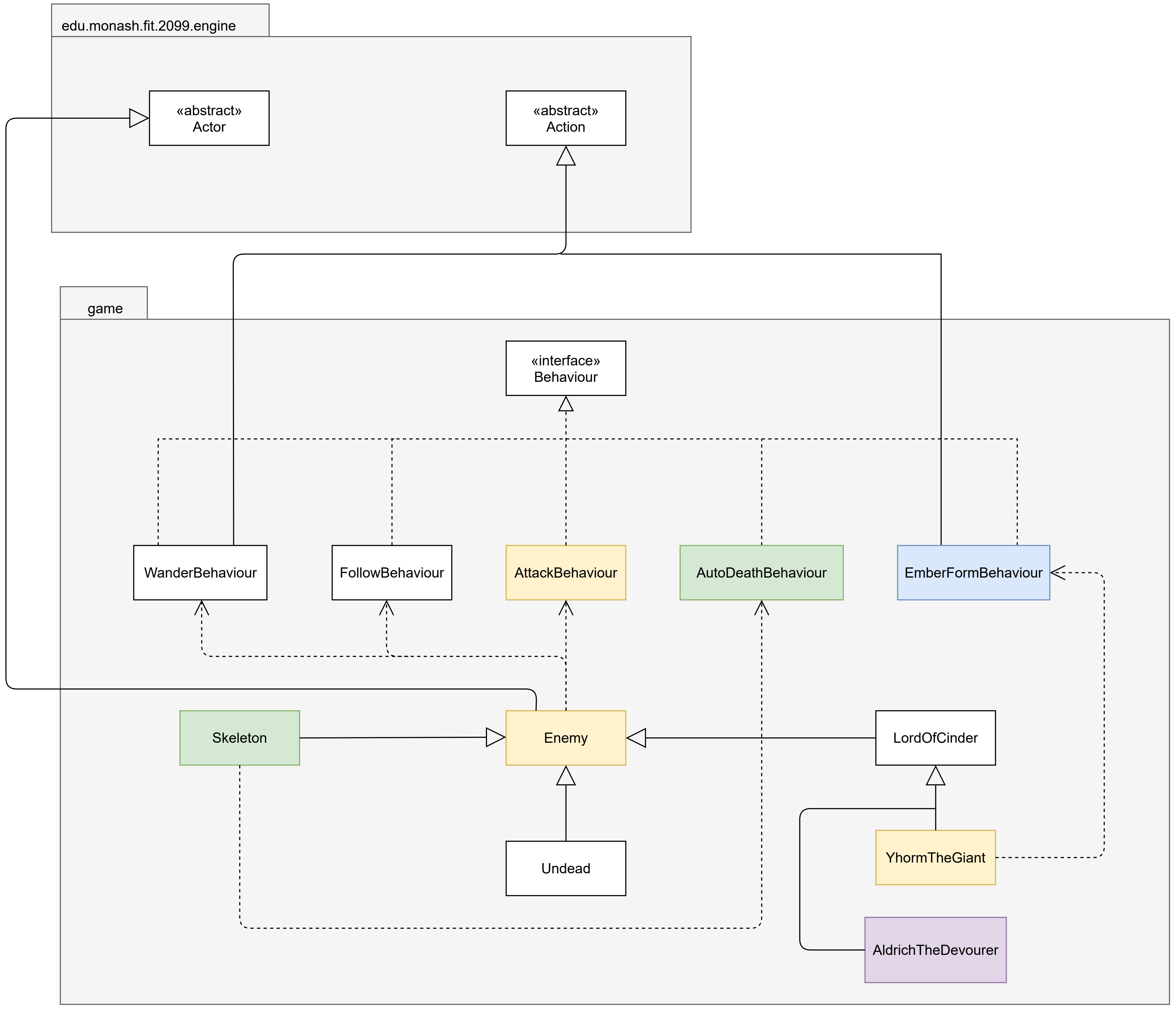
This class diagram shows the relationship between the Pet Class and its functionalities.



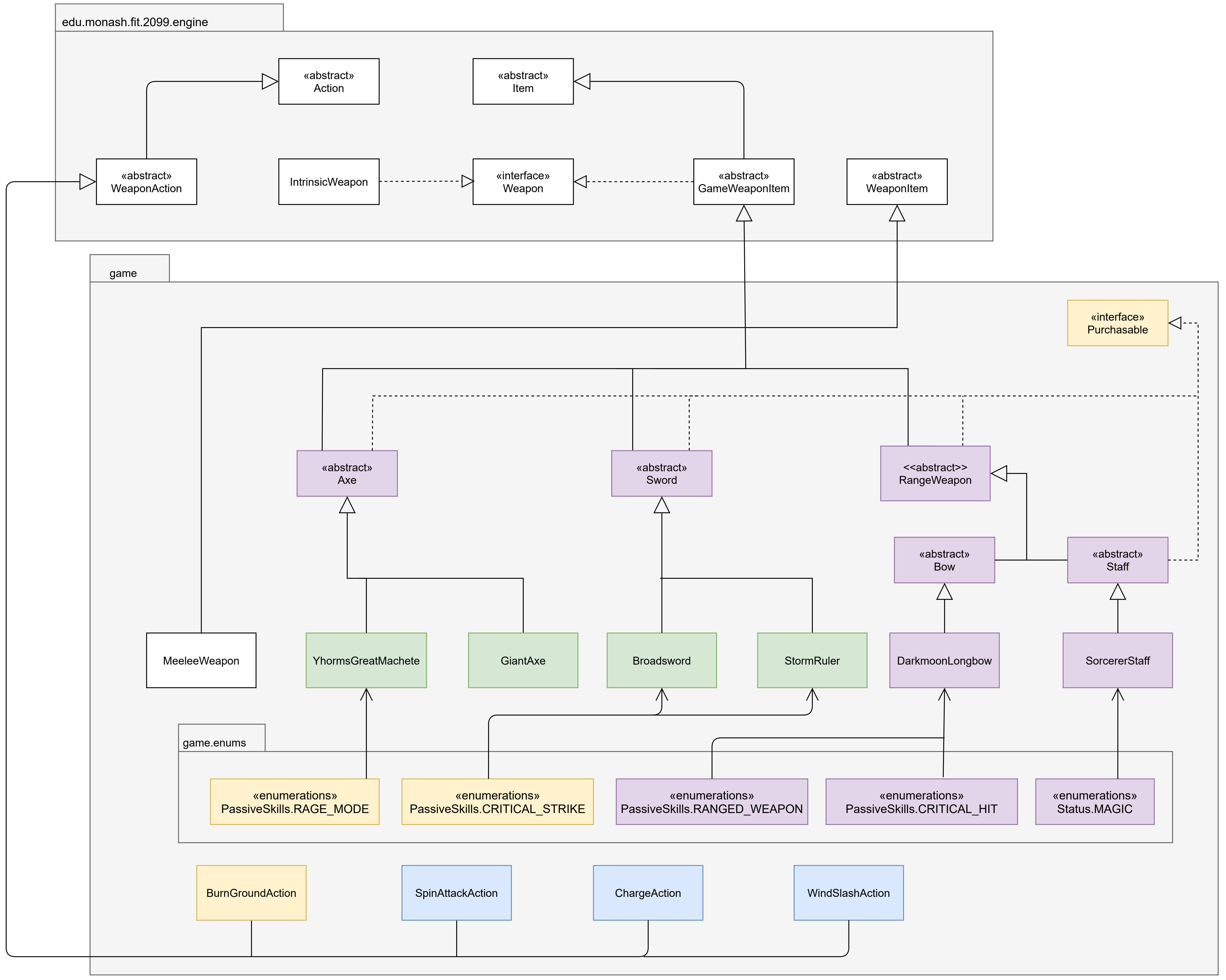
This class diagram shows the relationship between the Staff Class and its functionalities.



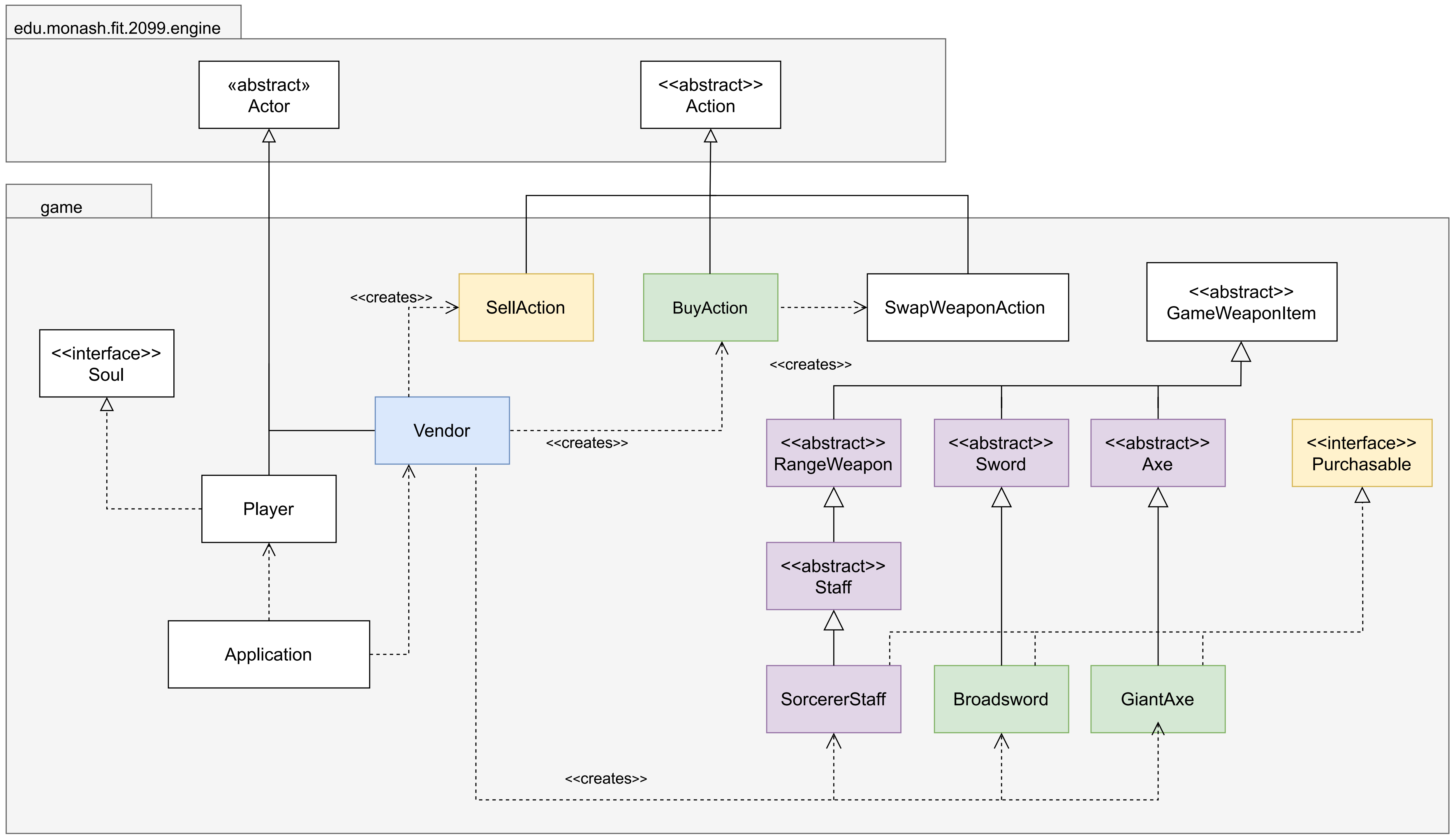
This class diagram shows the relationships between the Enemies and their basic Behaviours.



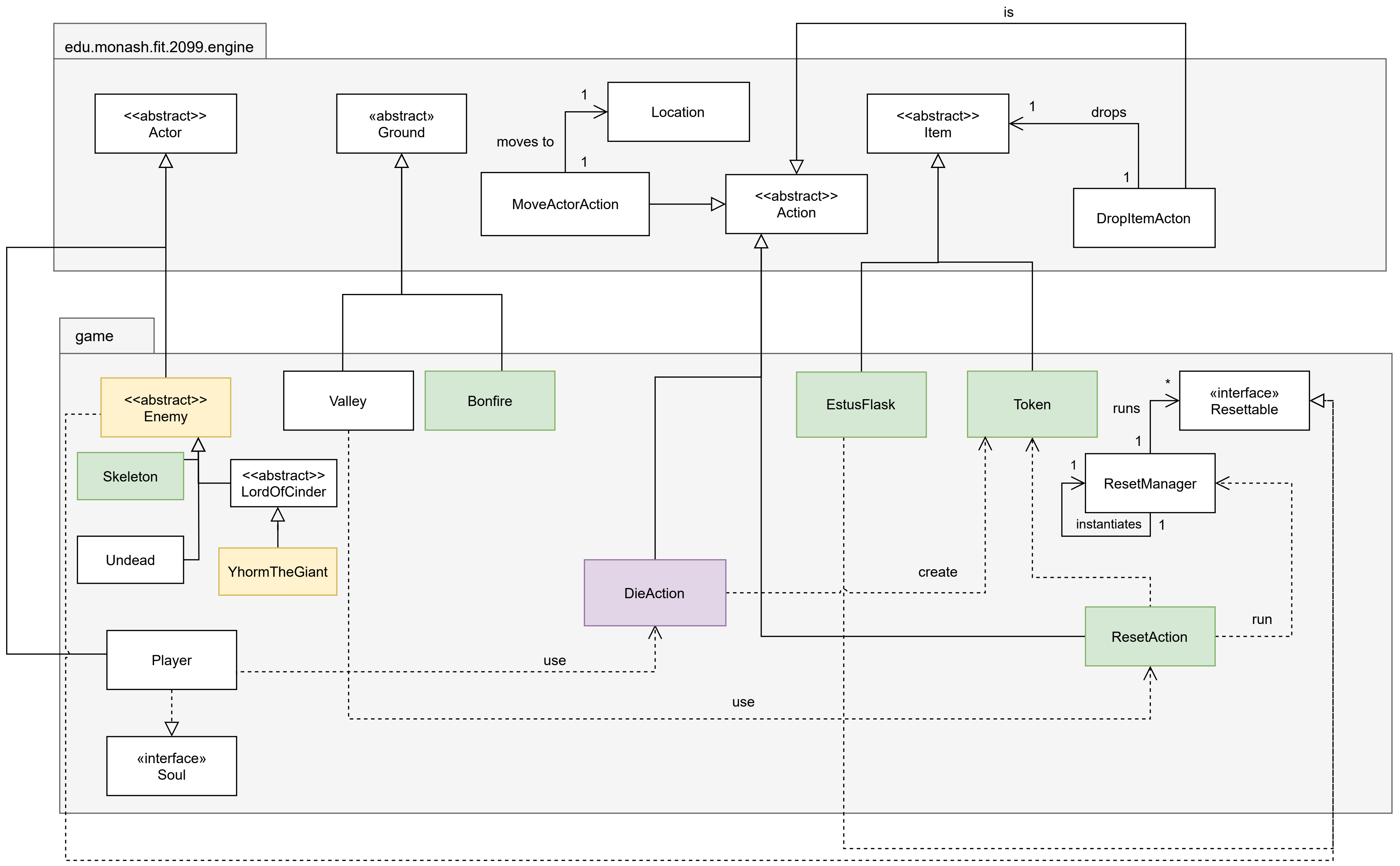
This class diagram shows the relationships between Weapons and Weapon Actions.



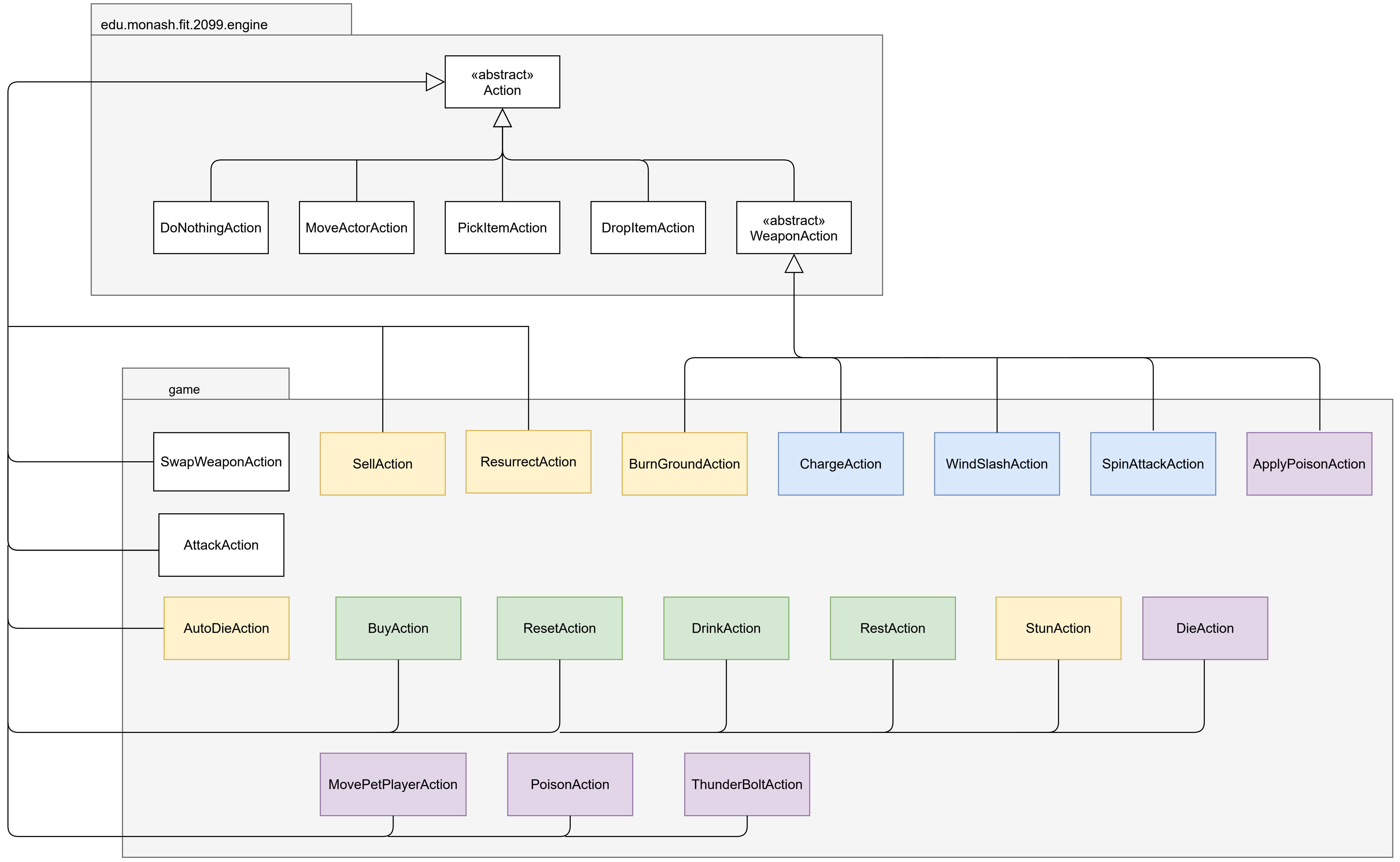
This class diagram shows the relationships between Vendor and its actions.



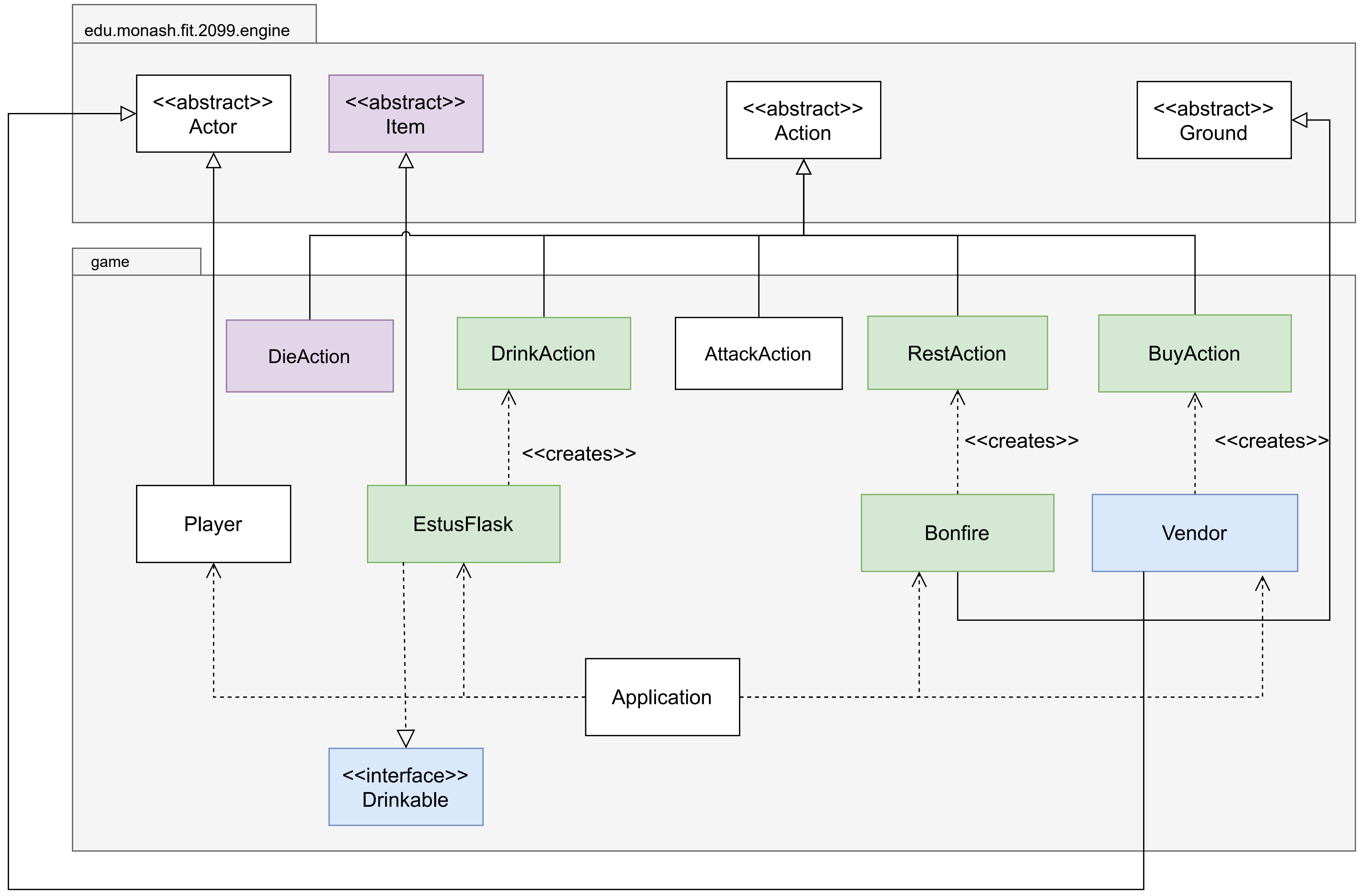
This class diagram shows the relationships between the classes when an actor dies in the game.



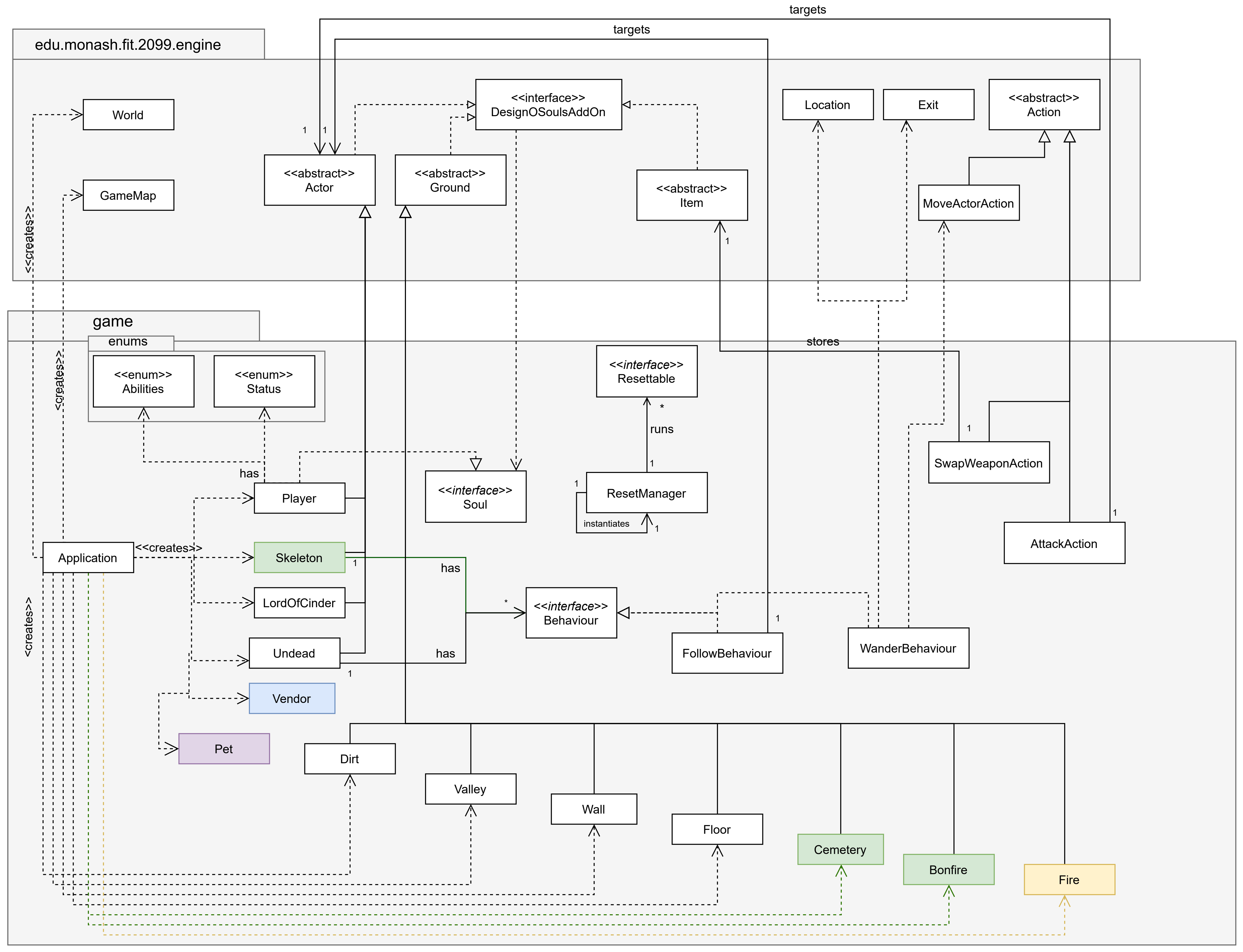
This class diagram shows the relationships between the different actions available in the game.



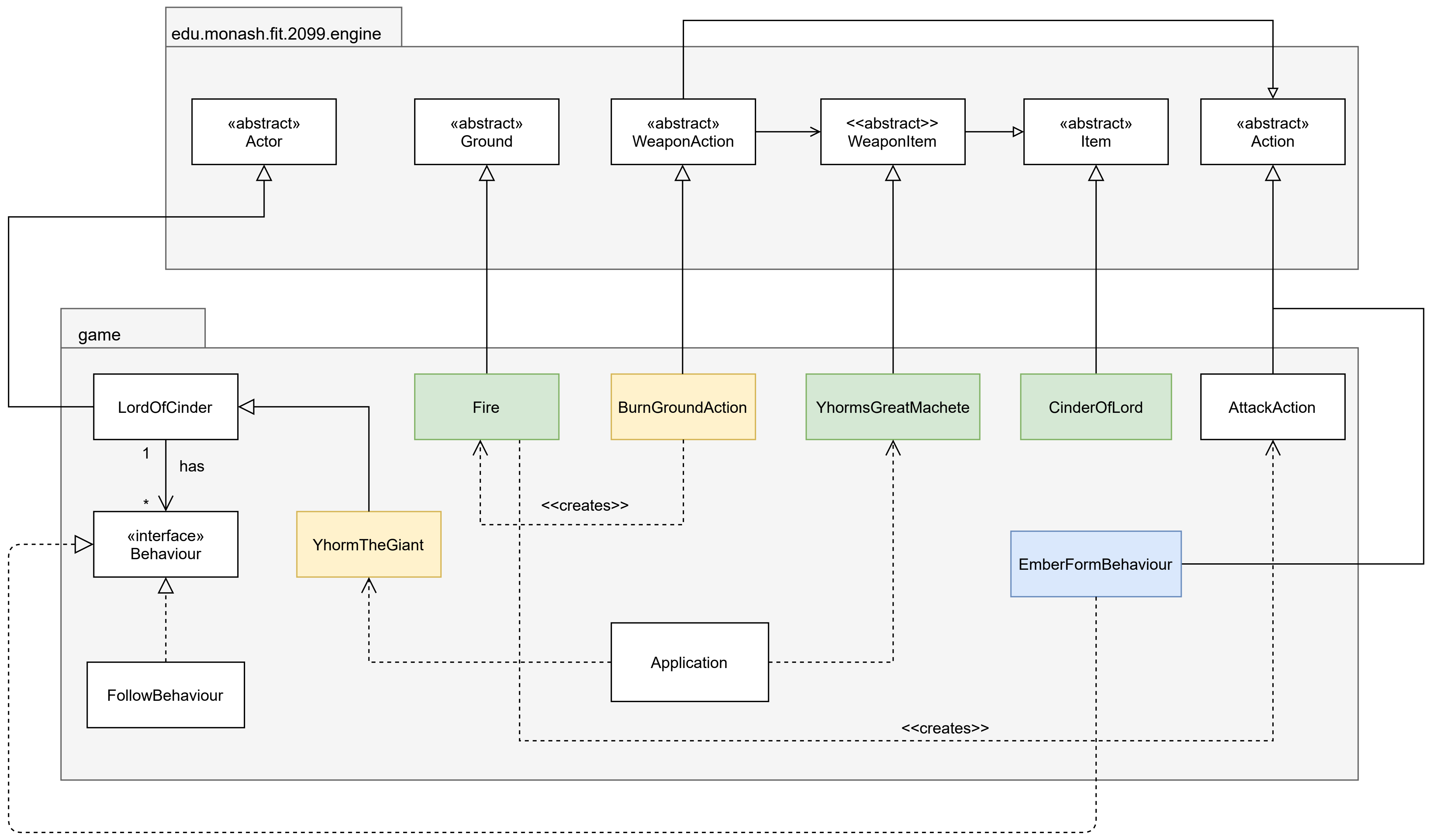
This class diagram shows the relationship between the Player and Action class.
It shows the available actions as a player.



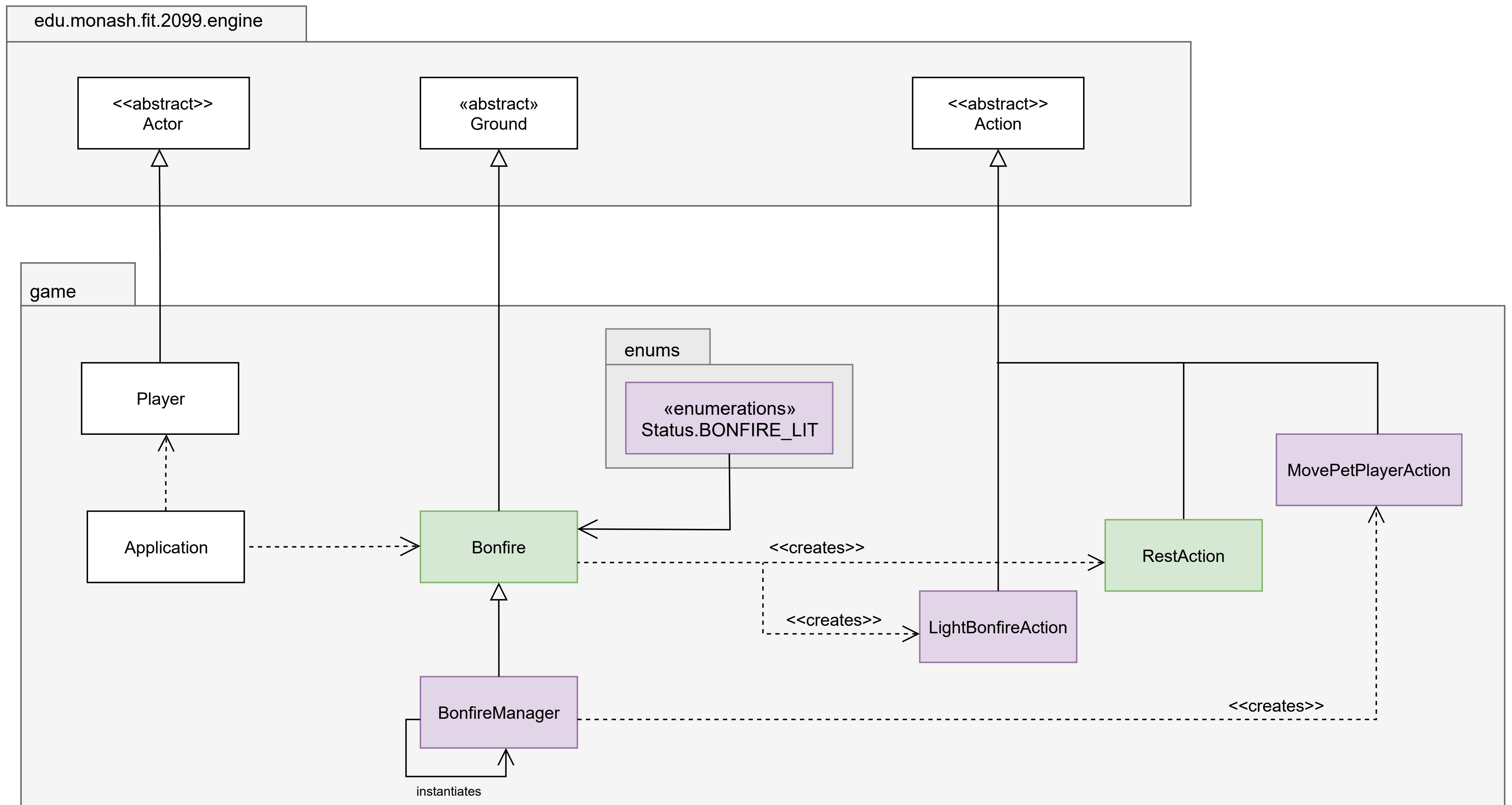
This class diagram shows the relationships between classes in the engine and game packages.



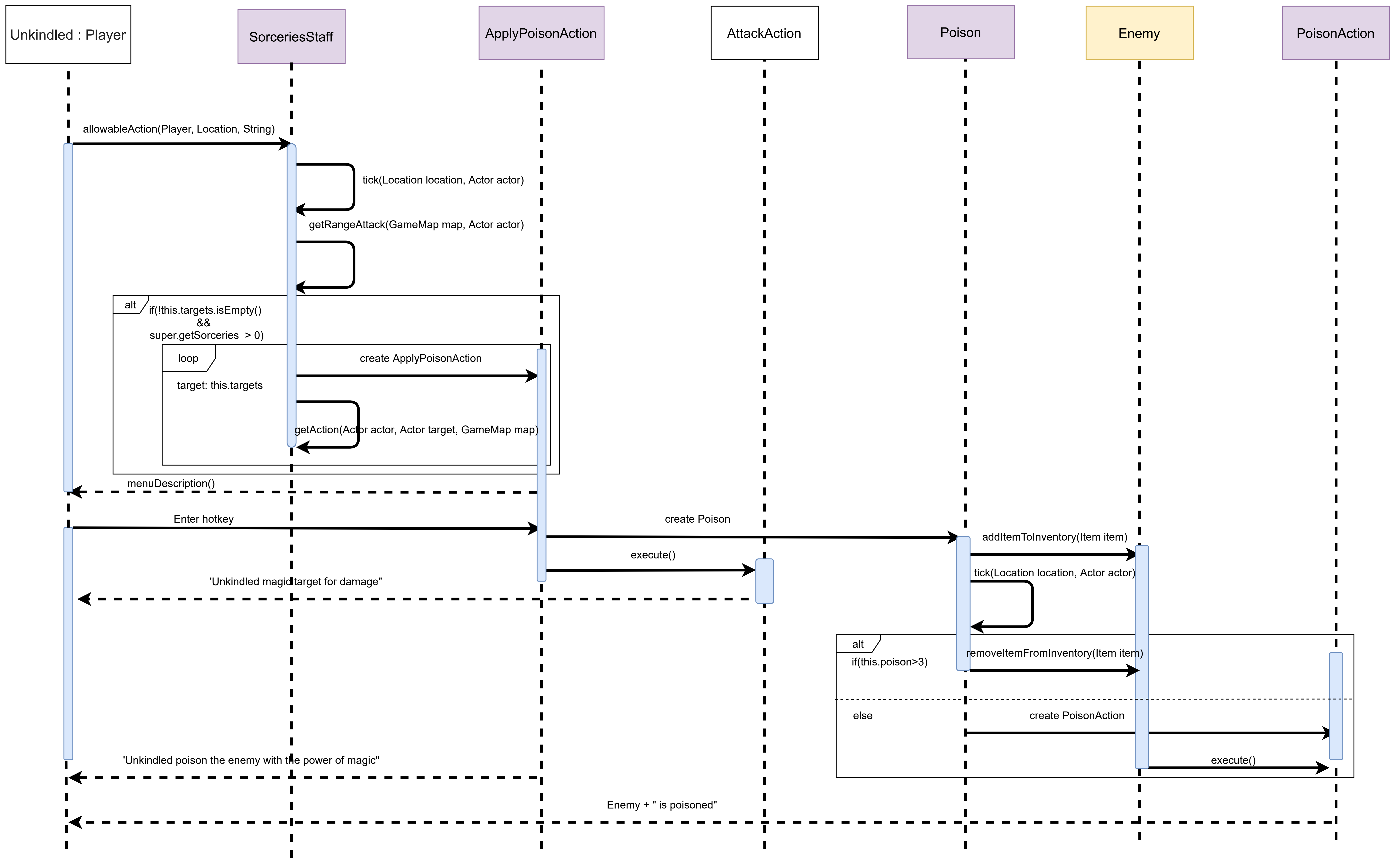
This class diagram shows the relationship between Yhorm The Giant and its related classes.



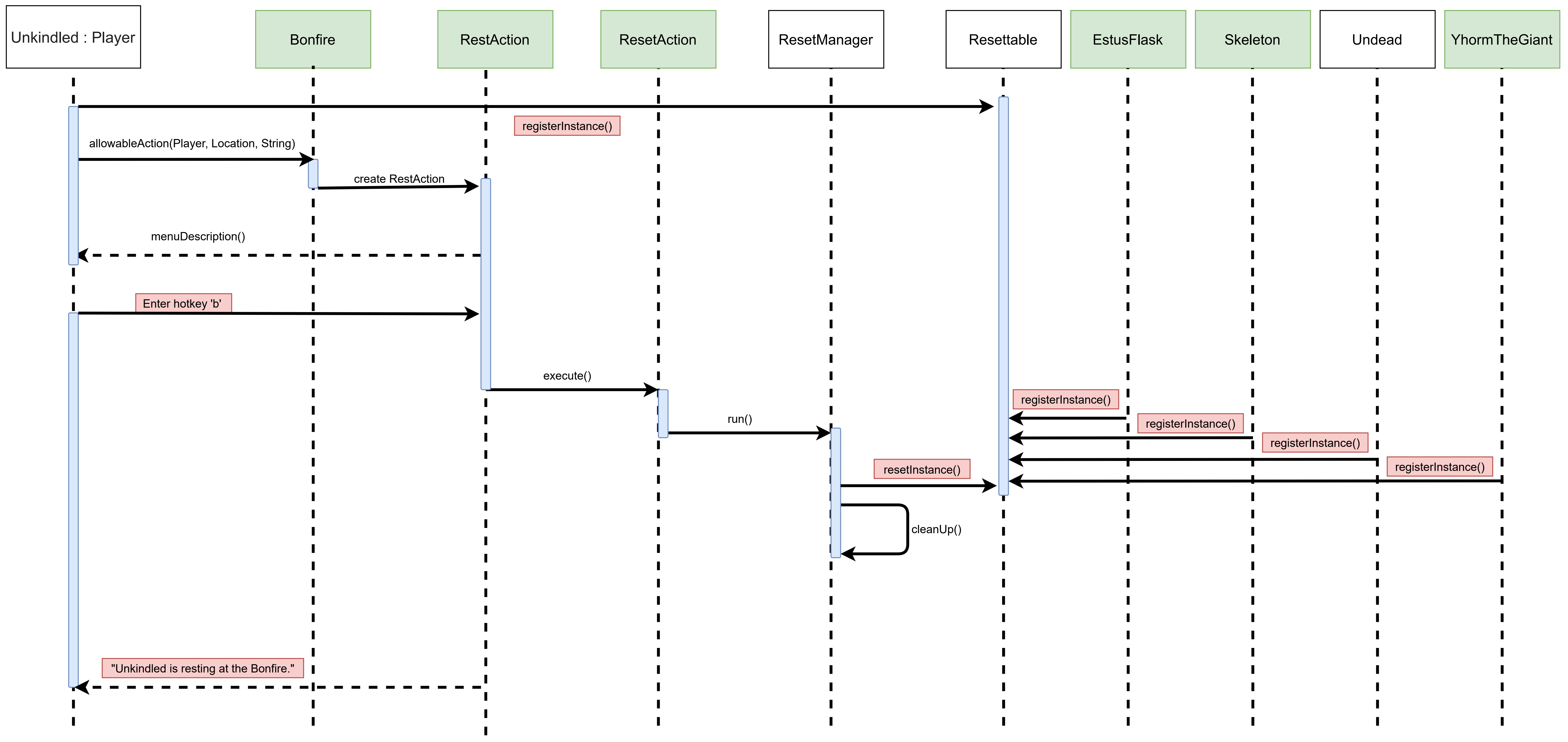
This class diagram shows the relationships between the Bonfire and all of it’s related classes.



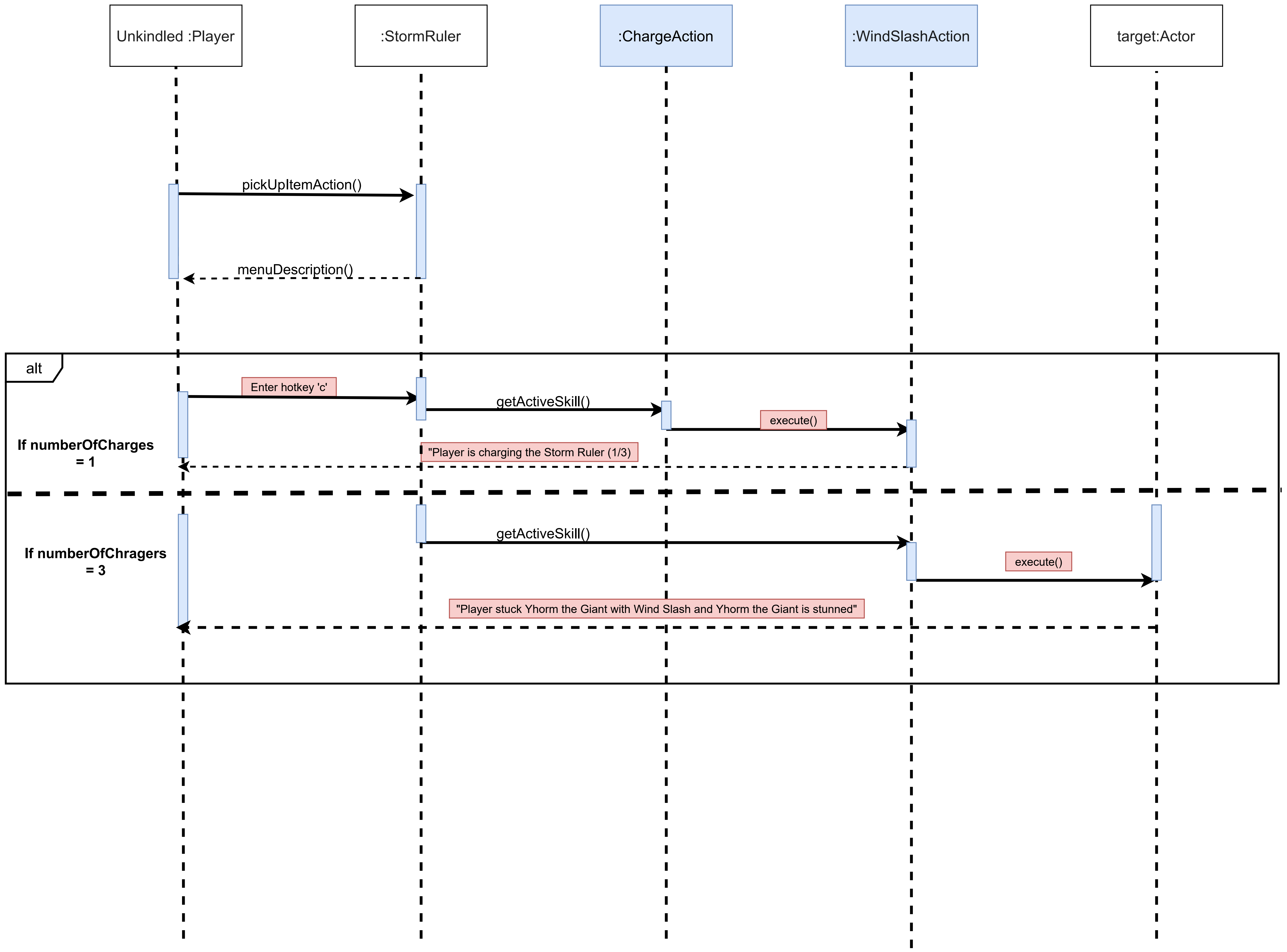
This is the sequence diagram that the Player uses the SorceriesStaff and uses the action of the staff on the enemy.


Unkindled : Player
SorceriesStaff
ApplyPoisonAction
AttackAction
Poison
Enemy
PoisonAction
allowableAction(Player, Location, String)
tick(Location location, Actor actor)
getRangeAttack(GameMap map, Actor actor)
alt
if(!this.targets.isEmpty()
&&
super.getSorceries > 0)
loop
target: this.targets
create ApplyPoisonAction
getAction(Actor actor, Actor target, GameMap map)
menuDescription()
Enter hotkey
create Poison
execute()
'Unkindled magic target for damage"
addItemToInventory(Item item)
tick(Location location, Actor actor)
alt
if(this.poison>3)
removeItemFromInventory(Item item)
else
create PoisonAction
execute()
'Unkindled poison the enemy with the power of magic"
Enemy + " is poisoned"

This is the sequence diagram for when a Player decides to rest in the game.



This is the sequence diagram between StormRuler and its Active Skills; ChargeSkill and WindSlashSkill.



Unkindled :Player
:StormRuler
:ChargeAction
:WindSlashAction
target:Actor
pickUpItemAction()
menuDescription()
alt
Enter hotkey 'c'
getActiveSkill()
execute()
If numberOfCharges = 1
"Player is charging the Storm Ruler (1/3)
If numberOfChragers = 3
getActiveSkill()
execute()
"Player stuck Yhorm the Giant with Wind Slash and Yhorm the Giant is stunned"