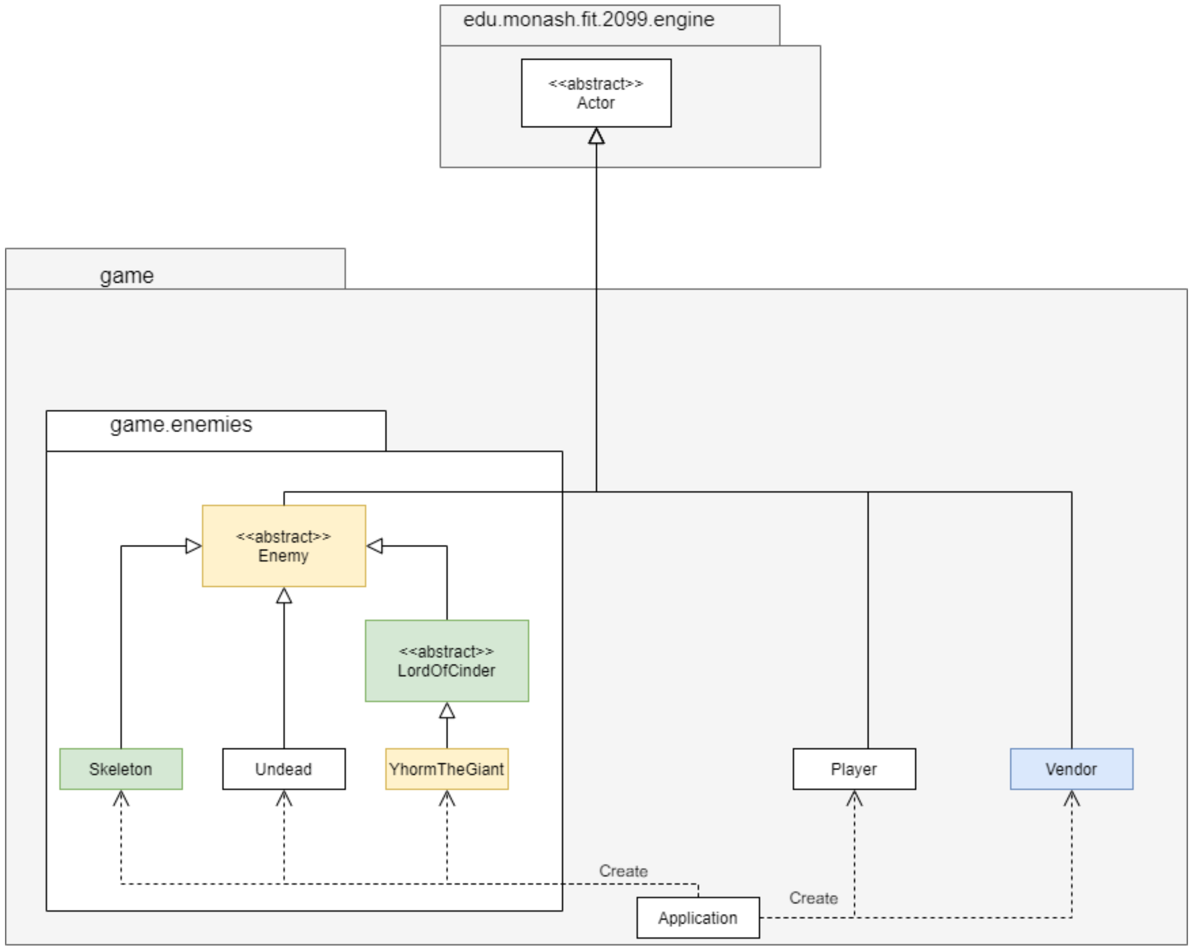
Assignment 2:
Implementation: First Iteration

Amendment Design Rationale

This Document is to show the changes in Assignment 2.

- 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 Class

Class Diagram 1



This class diagram shows the relationship of the different actors in the game.

New Classes:

<<abstract>> Enemy

We create a new abstract class that is named as Enemy. This class is designed as a parent class for all the enemies 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. The benefit of creating this abstract Enemy class is that we are able to add features or any methods it needs for all the enemy characters in the game easily. This corresponds to the Open/Closed Principle. For example, if we want all the enemy characters to have the same allowable action that will attack a Player only, we can implement the getAllowableAction method in the Enemy class so that every enemy 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 Enemy class as all the child classes are extended to the parent Enemy class.

YhormTheGiant

This class is used to create an instance of an enemy of the type YhormTheGiant which is the boss in the game. 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 adheres to low coupling. Additionally, we can easily increase the number of the YhormTheGiant in the game by just simply re-using the class to create a new YhormTheGiant. Having its own class also follows the Single Responsibility Principle as it is only in charge of representing Yhorm the Giant in the game.

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

Changes in classes:

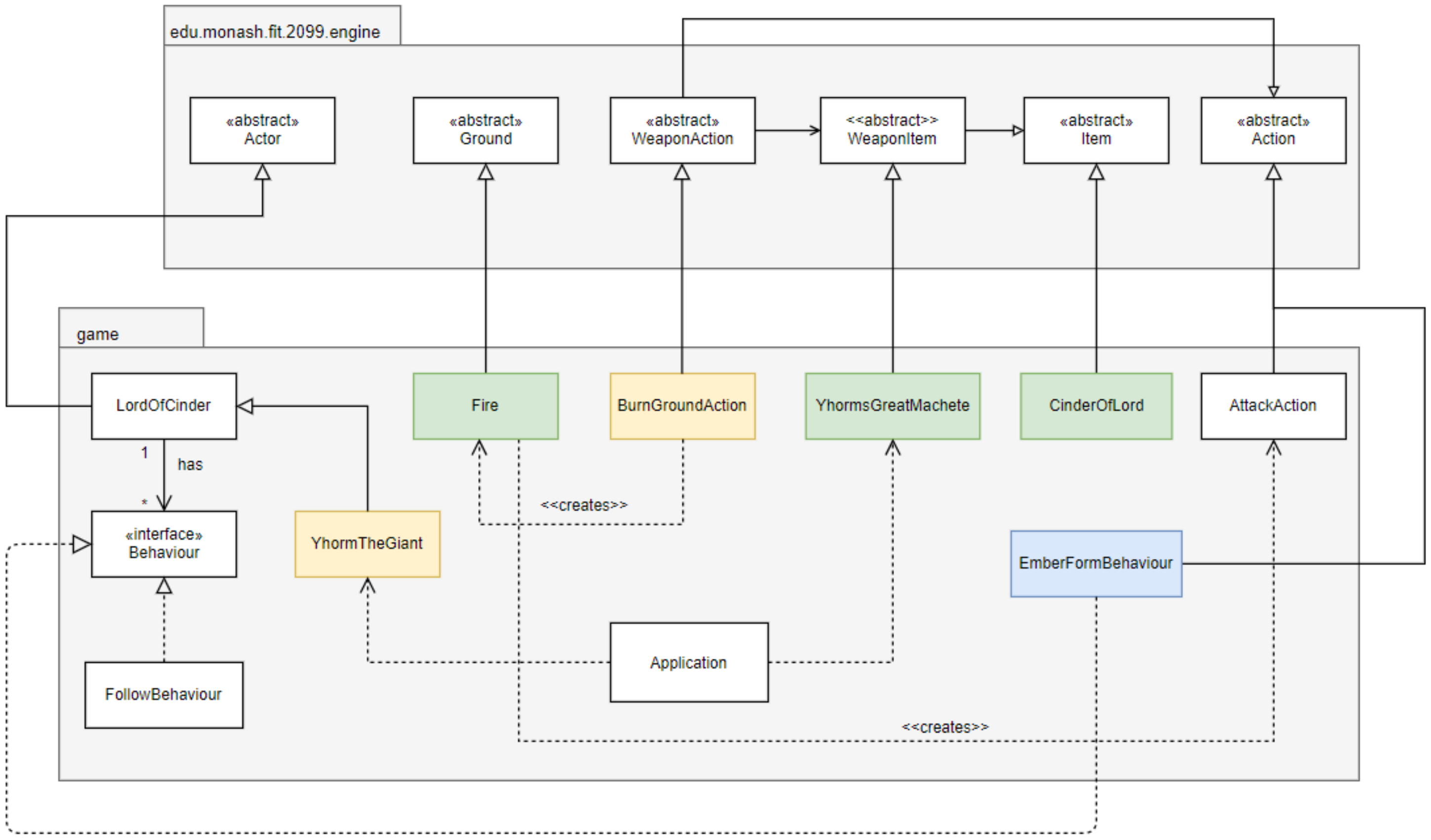
<<abstract>> LordOfCinder

This class has changed so that every boss in the game is able to inherit it. The purpose of this class is that in the future, more new bosses will be added and every boss in the game has many special abilities and behaviours. We can simply create a new boss class that inherits this abstract class that has all the methods or features that a boss needed in the gaem. Benefits of this class is we can reduce the repeating code or method for each of the new boss classes. This is corresponding to the DRY (Do not Repeat Yourself) principle in which we can reduce repeated code. In the future implementation, if there are more than one boss, we just need to add the specific methods or features of the specific boss in its class and all the methods that are similar and all bosses needed, we can simply put those methods in the parent class which is this LordOfCinder abstract class.

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. 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.

<u>Class Diagram 2</u>



This class diagram shows the relationship between the Yhorm the Giant and its actions and behaviours.

New Classes:

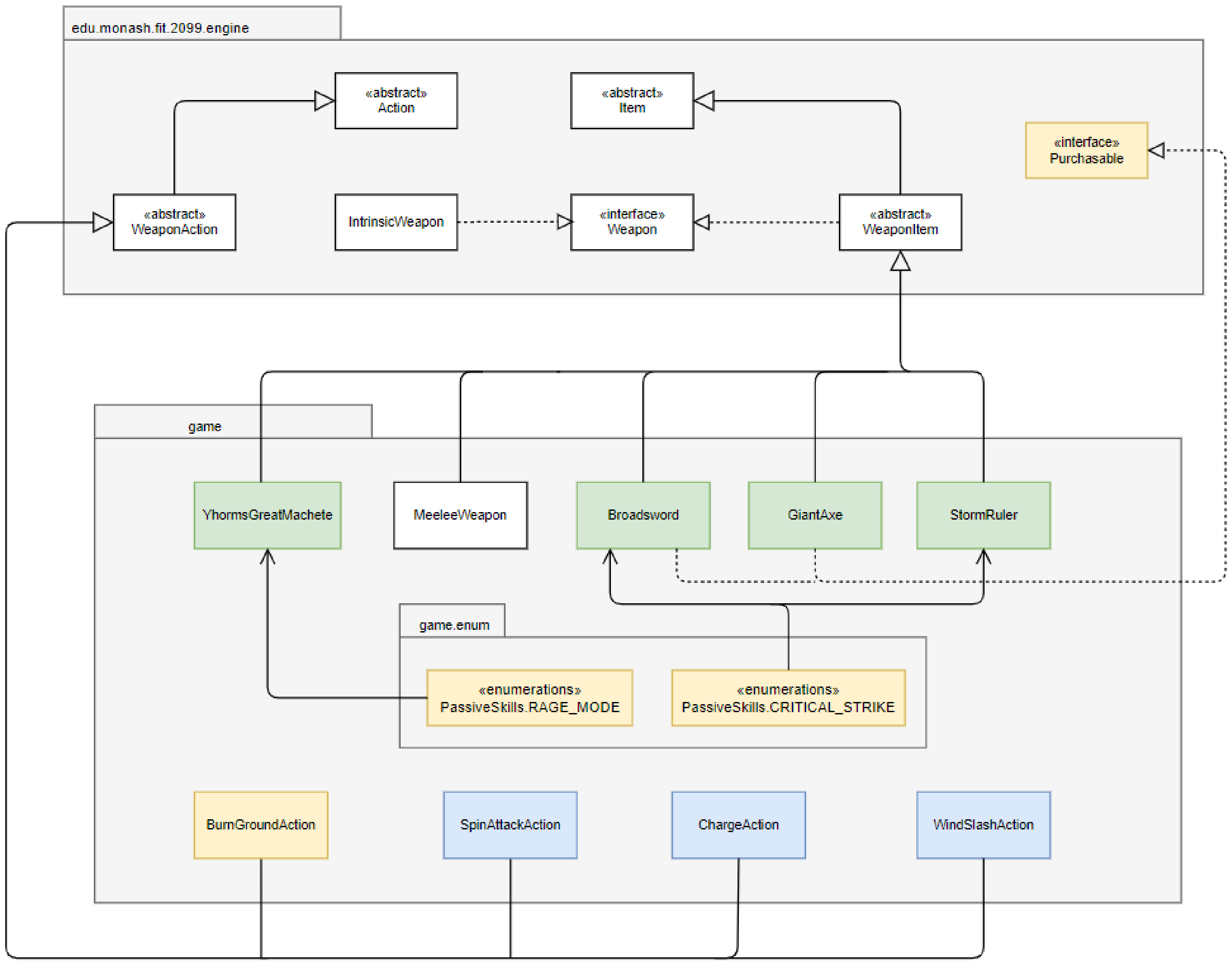
YhormTheGiant

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 while not affecting the rest of the system. This class is considered a type of actor which explains the relationship between Lord of Cinder which further extends to the Actor class. Its relationship with the Lord of Cinder class displays the Liskov Substitution Principle as it is a child of the class Lord of Cinder. By having this relationship, any changes that need to be modified for enemy bosses can be made within the Lord of Cinder class while allowing the same changes to take place in the child classes. Having its own class also follows the Single Responsibility Principle as it is only in charge of representing Yhorm the Giant in the game.

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 as it is considered an active skill of the weapon. 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 which follows the. 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 which adheres 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.

EmberFormBehaviour
The changes in this class includes 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.

Class Diagram 3



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

New Classes:

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 which adheres 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.

Changes in Classes:

SpinAttackAction

Changes that were made to this class involves the changing of the name of the class itself from SpinAttack to SpinAttackAction. The reason being because 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.

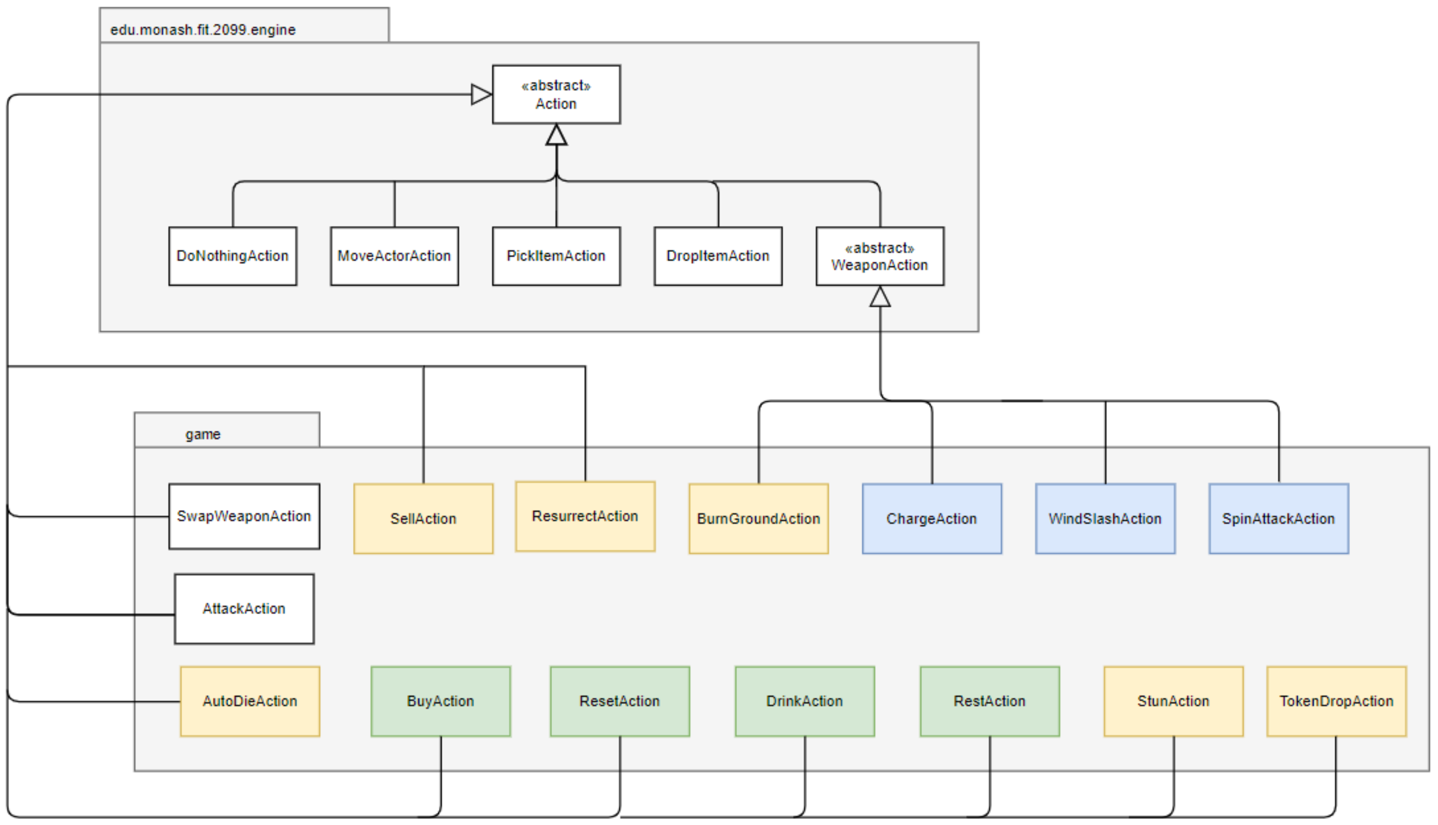
ChargeAction

Changes that were made to this class involves the changing of the name of the class itself from Charge to ChargeAction. The reason being because 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 being because 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 4



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

New 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 to die 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 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 it’s 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 wishes 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.

TokenDropAction

This class represents the logic behind the dropping of a token when a player dies at the location of the player. This class is created so that other actors are able to inherit this action if needed in the future, by creating this class we adhere to the DRY principle as we reduce the chances of any repeating code. The design of this class also follows the Single Responsibility Principle as it only has one responsibility that is to drop the token when the player dies. This class extends to the Action class as it is considered a type of action.

Changes in classes:

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

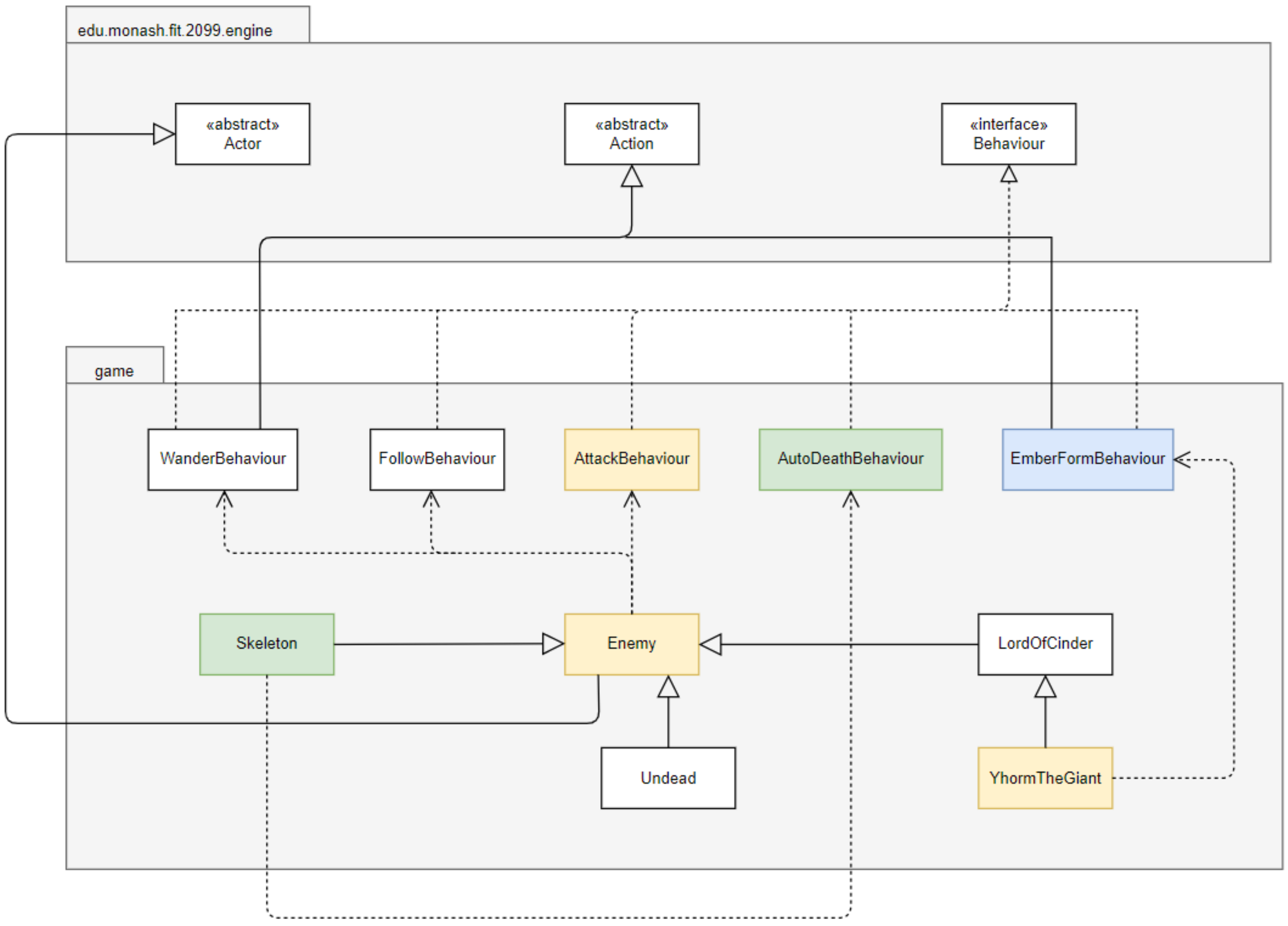
We decided to change the name of the class from Charge to ChargeAction. We decided that this was best because it is considered a type of action after all. By adding the word 'Action' would help to better distinguish which classes are actions and which are not. Creating this class allows other weapons 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.

Class Diagram 5



This class diagram shows the relationship between the different types of enemies and its behaviours in the game.

New 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.

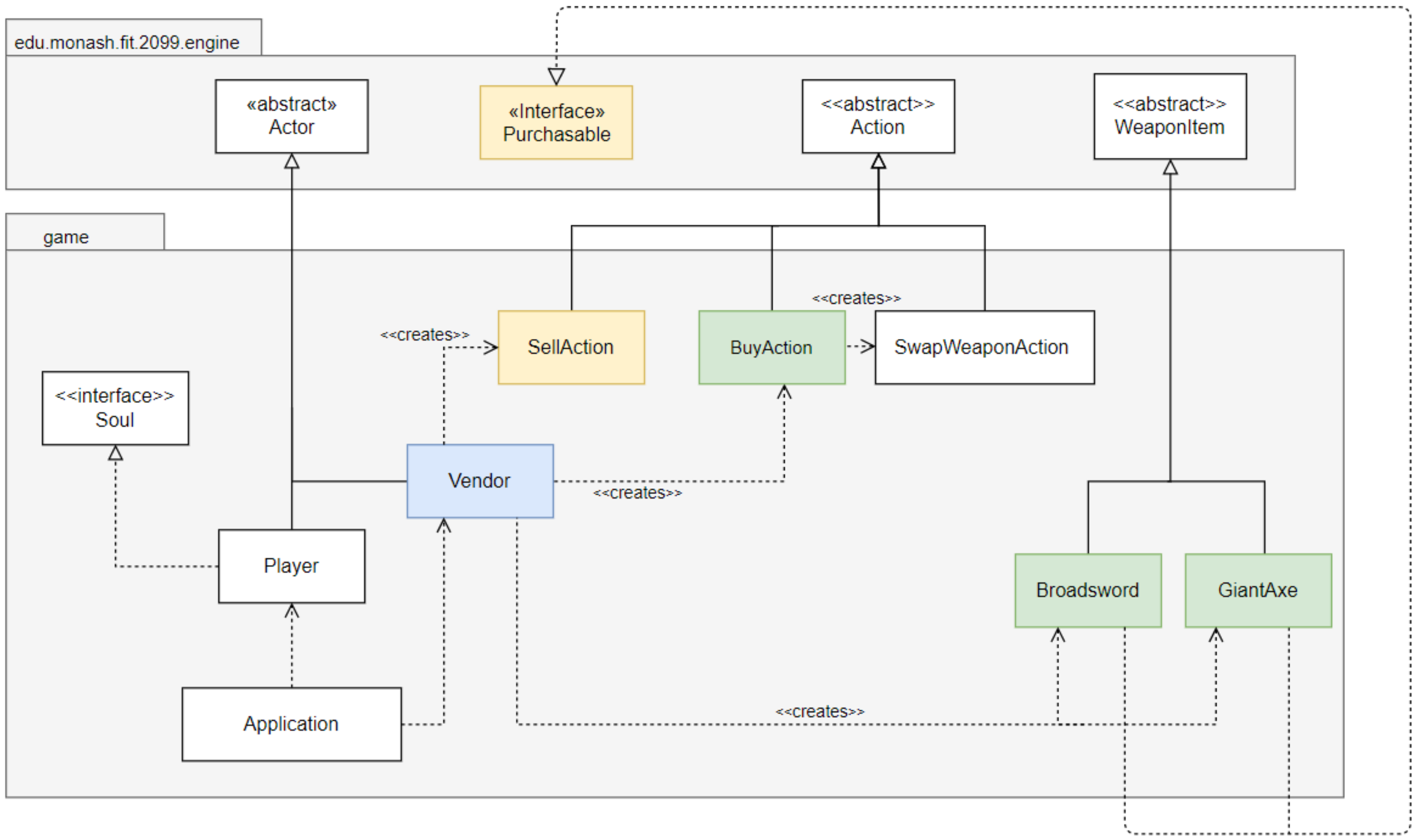
YhormTheGiant
This class represents the enemy Yhorm the Giant that is an extension of the Lord Of Cinder class as it is an enemy boss character in the game. Having its own class would allow it to have different characteristics that may differ from any future enemy bosses. This class has the ability to create the Ember Form Behaviour that only belongs to Yhorm the Giant itself. This class is considered a type of actor which 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.

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.

Changes in Classes:

EmberFormBehaviour
The changes in this class includes 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.

Class Diagram 6



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

New Classes:

SellAction

This class is used to represent the logic 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.

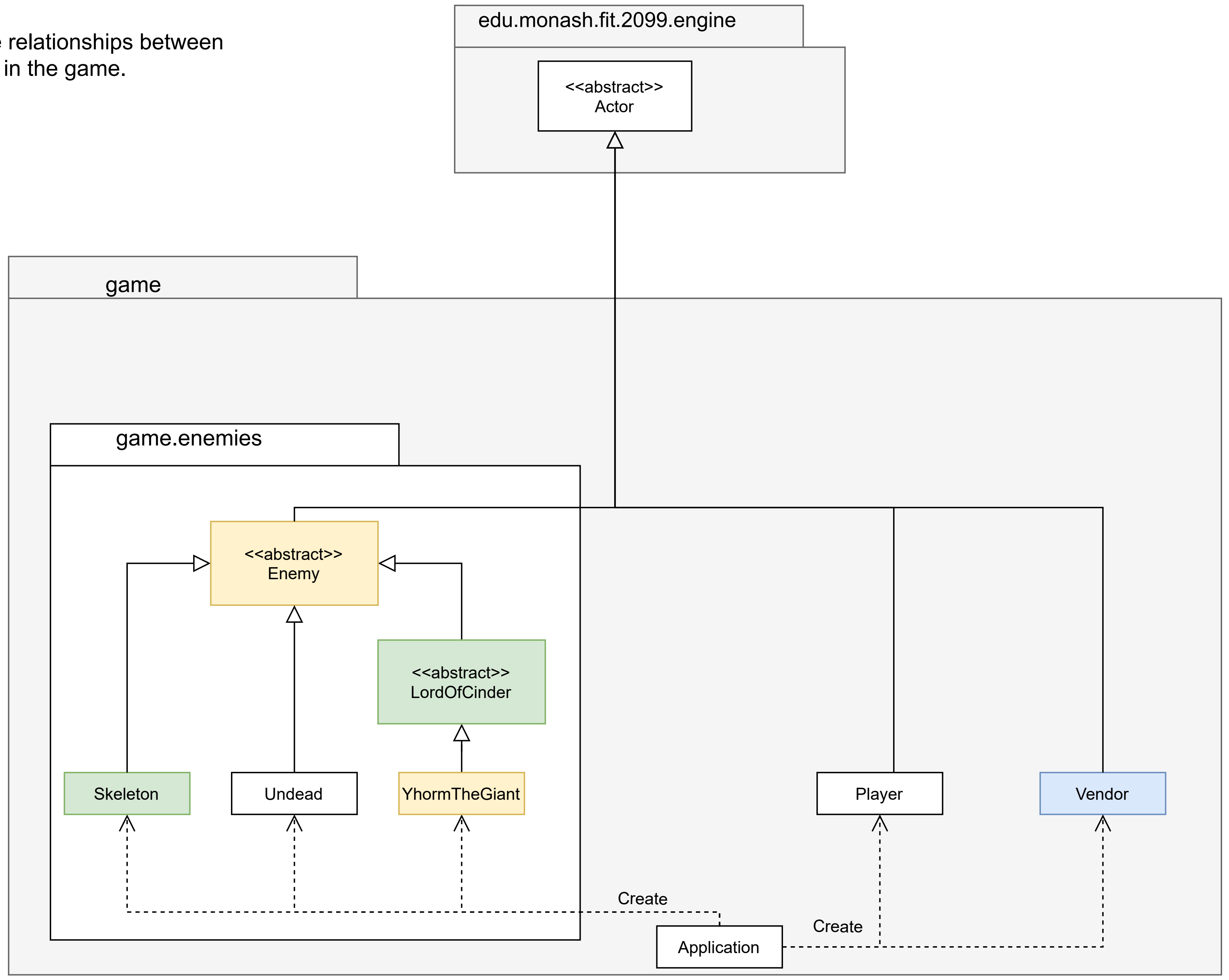
Changes in Classes:

Vendor

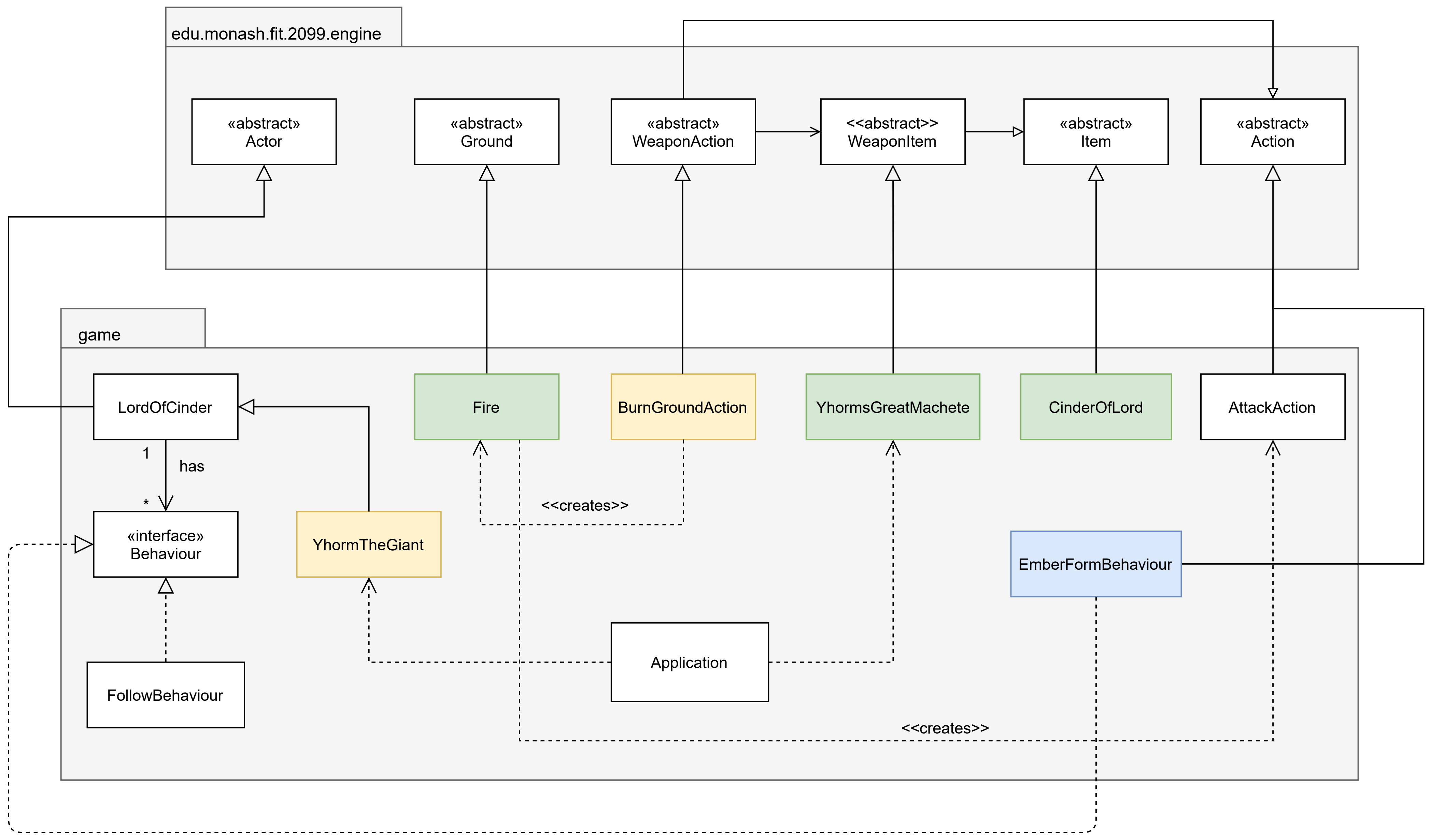
This class contains the logic behind the Vendor in the game. The changes made to this class is that it now extends to the Actor class where before it was extended to the Ground class. The reason for this change is that the Vendor class creates an action that can be performed by the Player as well as 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.

UML DIAGRAMS

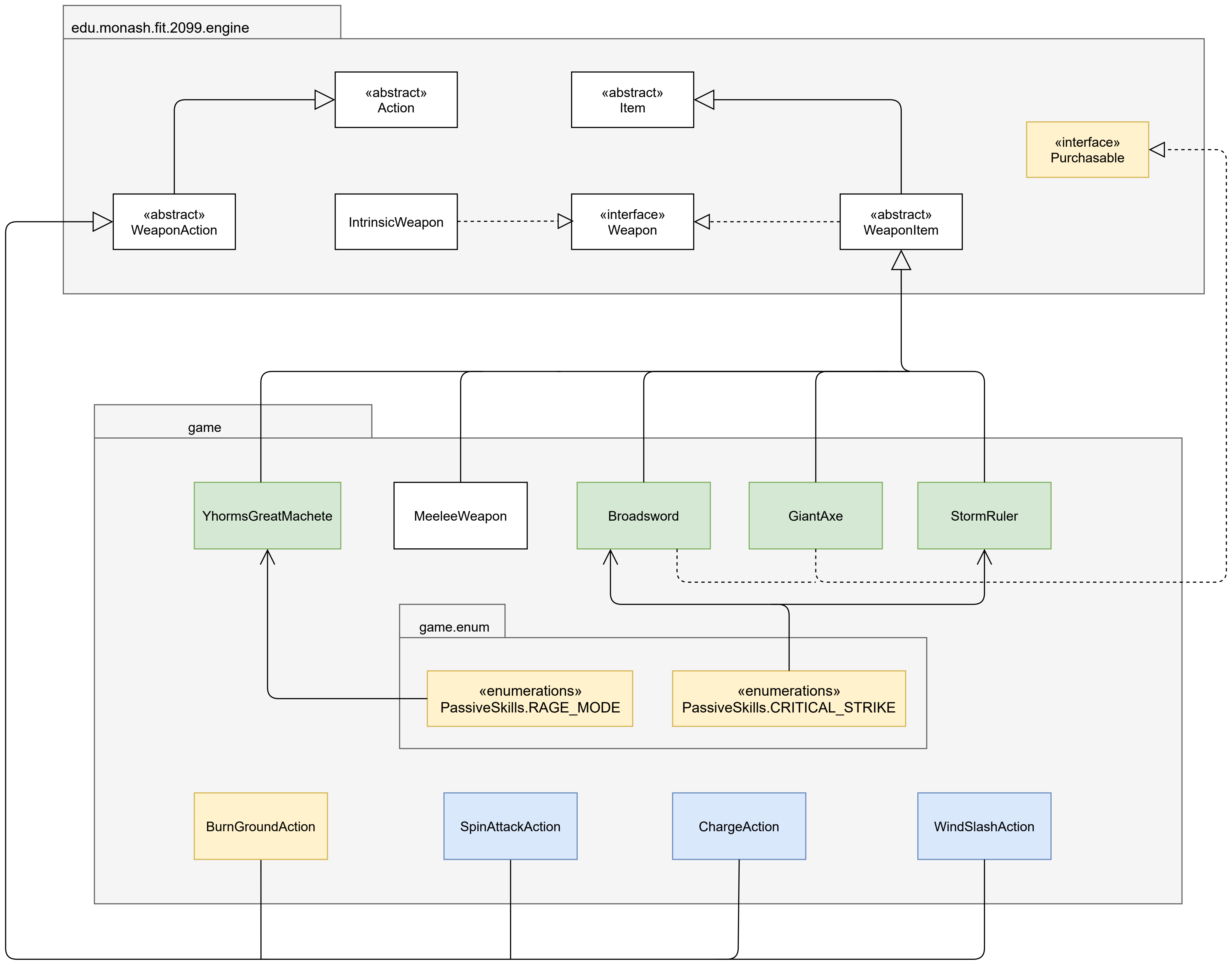
This class diagram shows the relationships between the different actors in the game.



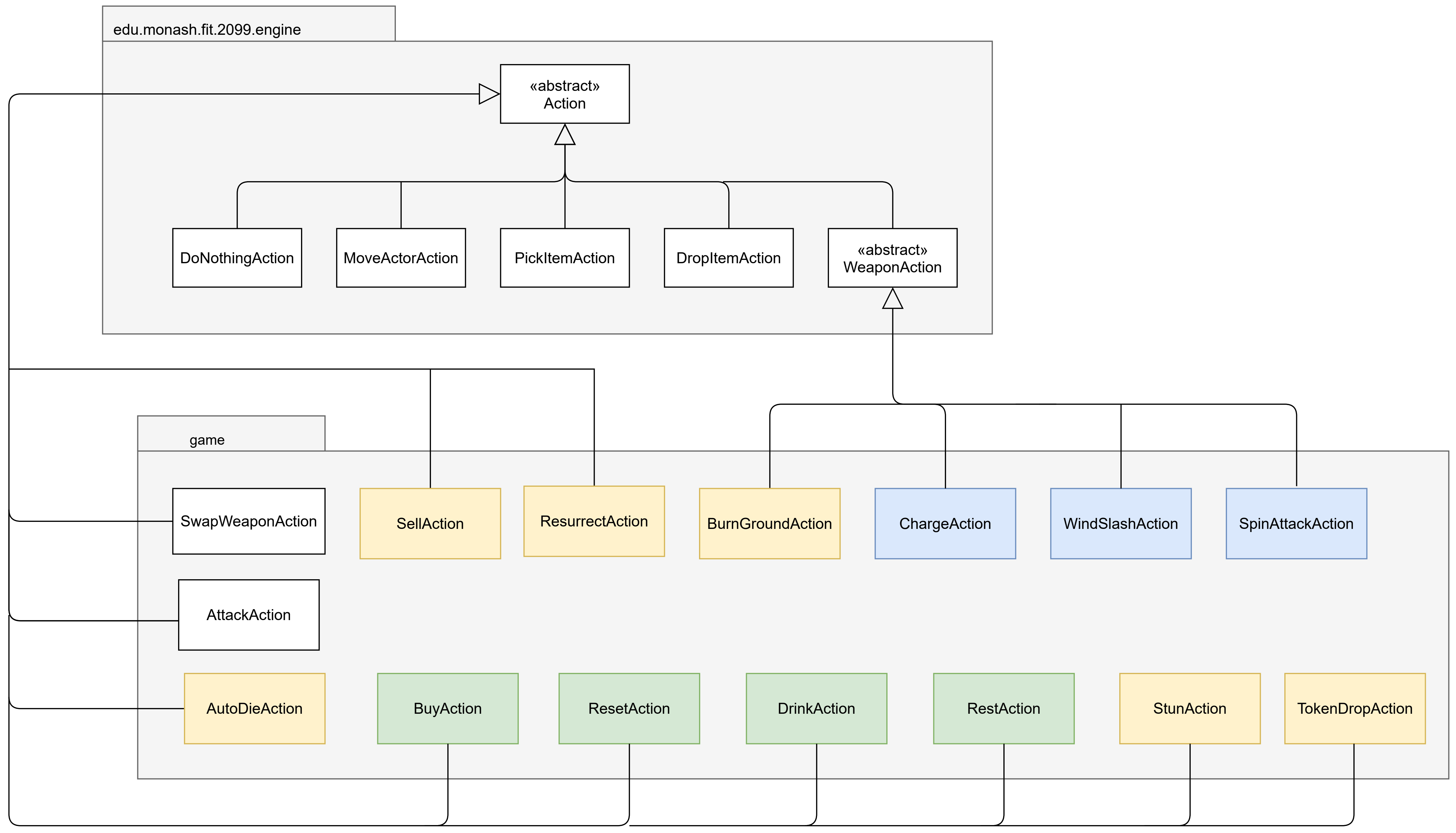
This class diagram shows the relationships of Lord Of Cinder and its related classes.



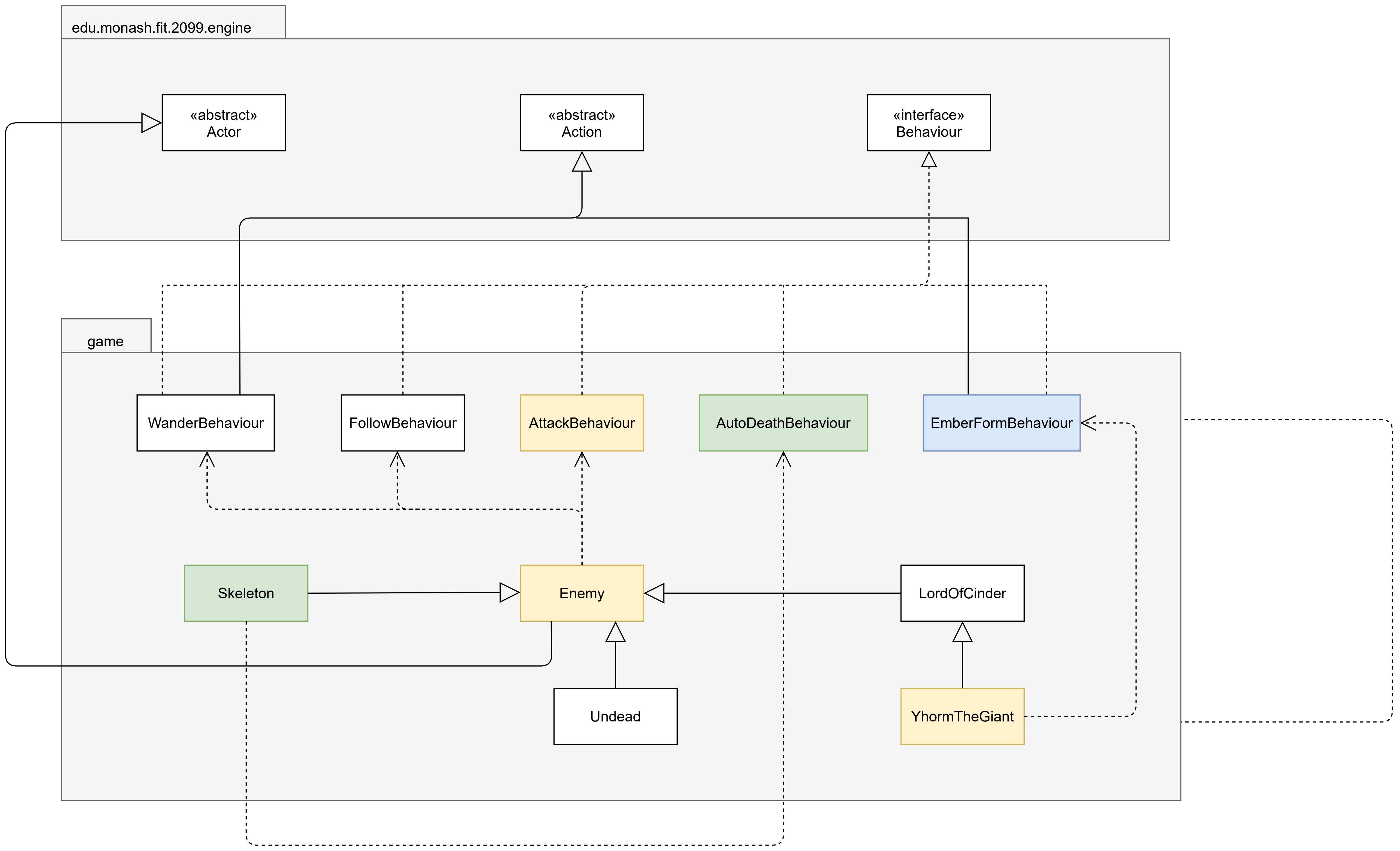
This class diagram shows the relationships between Weapons and Weapon Actions.



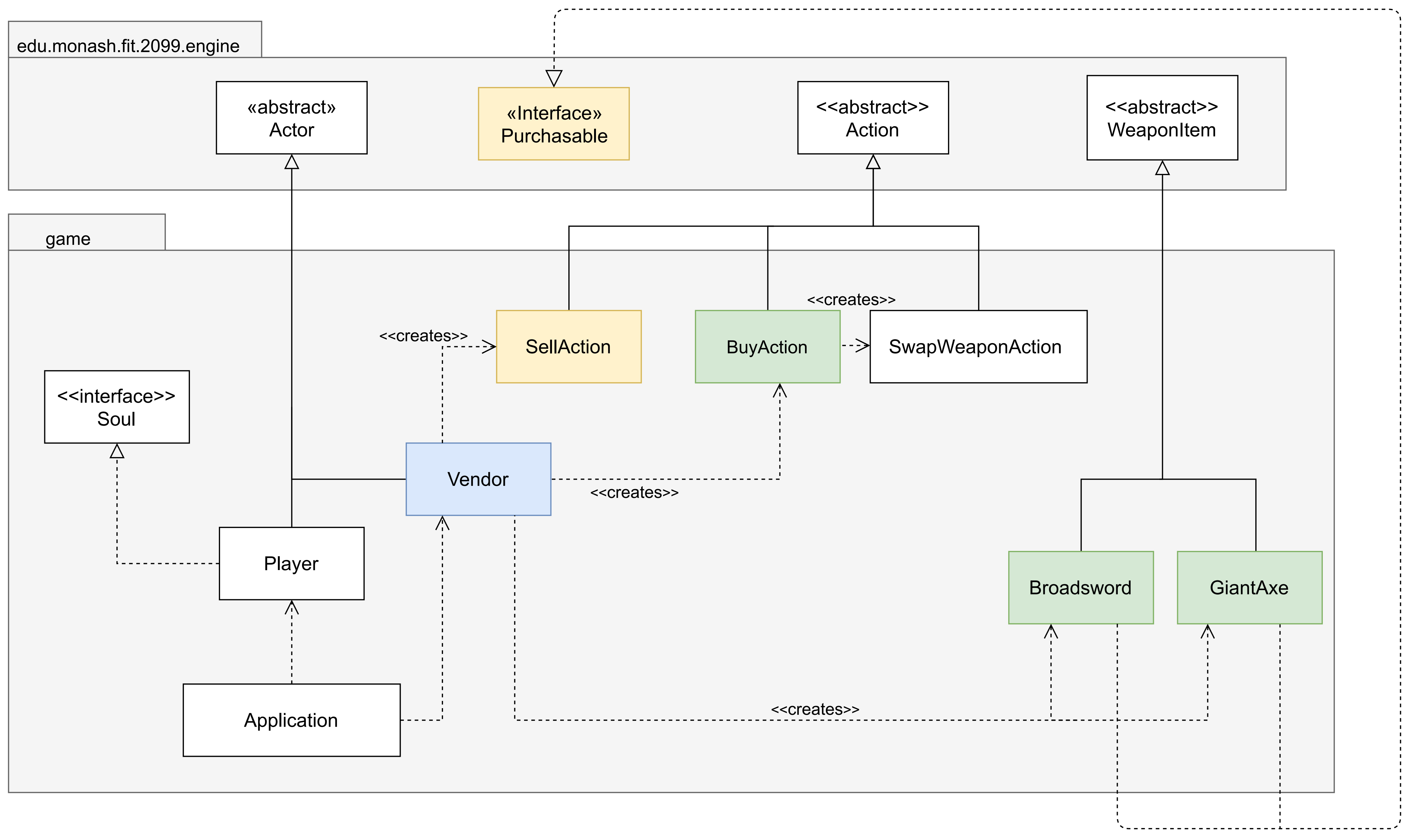
This class diagram shows the relationships between the different actions provided in the game.



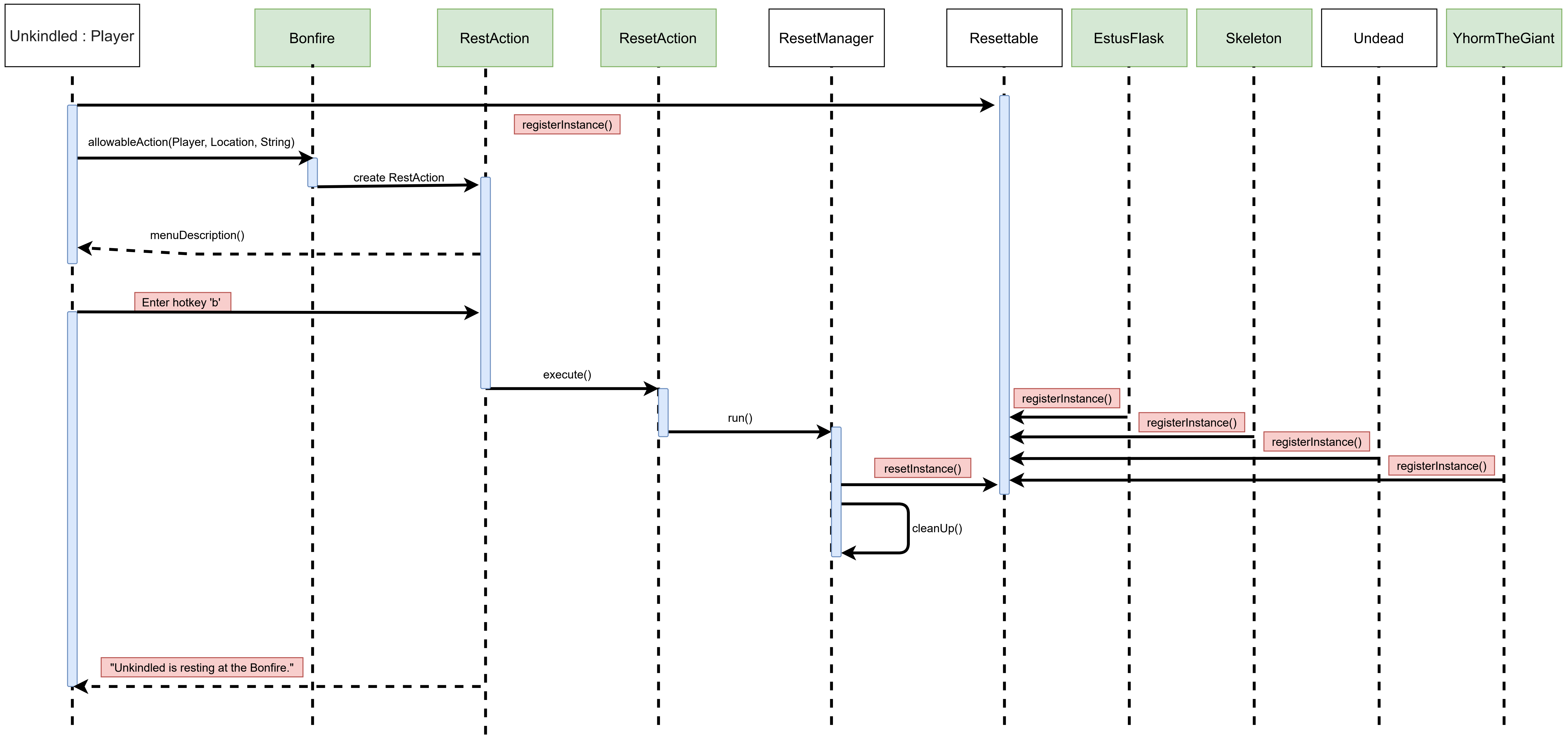
This class diagram shows the relationships between the Enemies and their basic Behaviours.



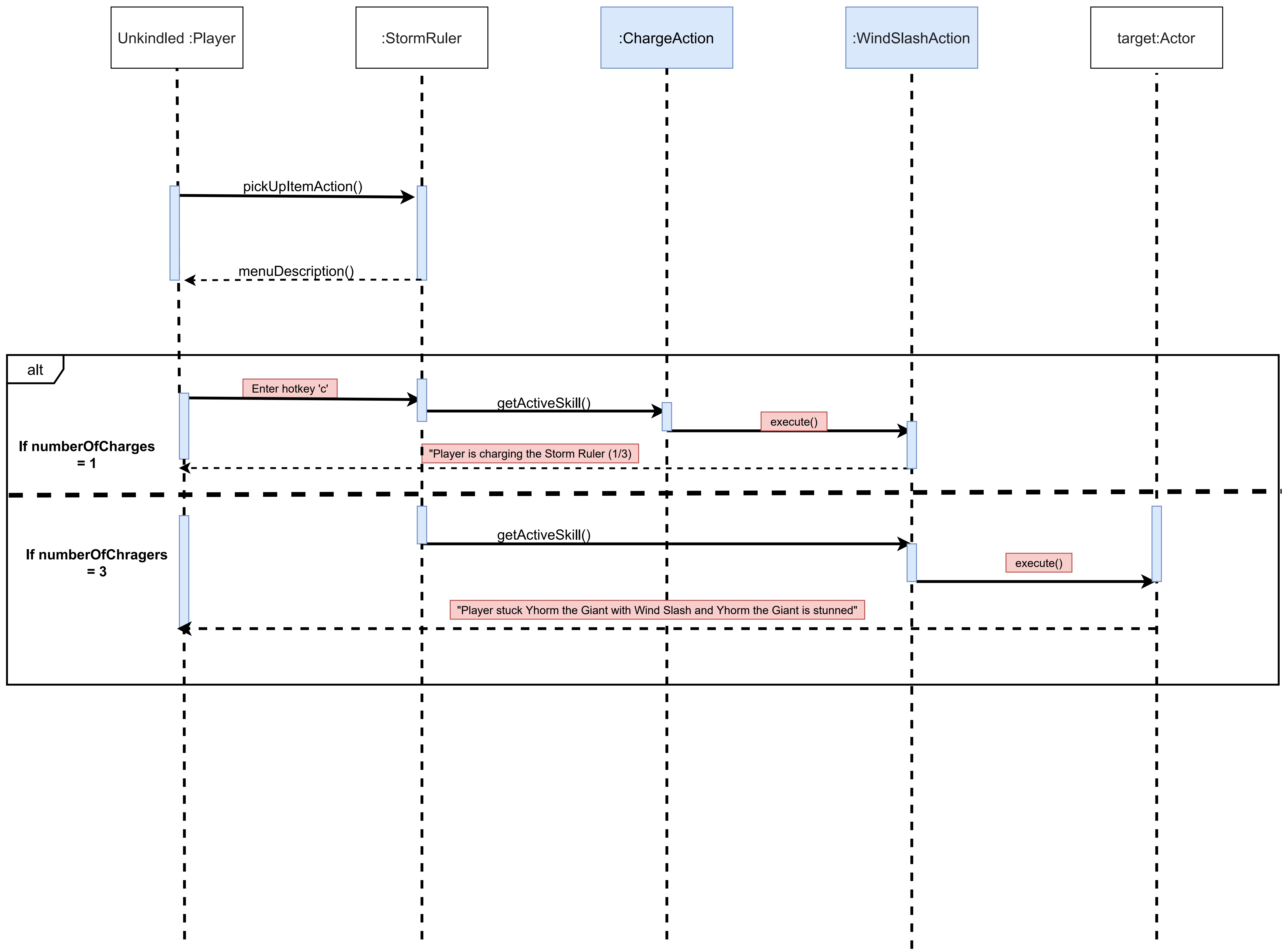
This class diagram shows the relationships between Vendor and its actions.



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)
getActiveSkill()
If numberOfChragers = 3
execute()
"Player stuck Yhorm the Giant with Wind Slash and Yhorm the Giant is stunned"