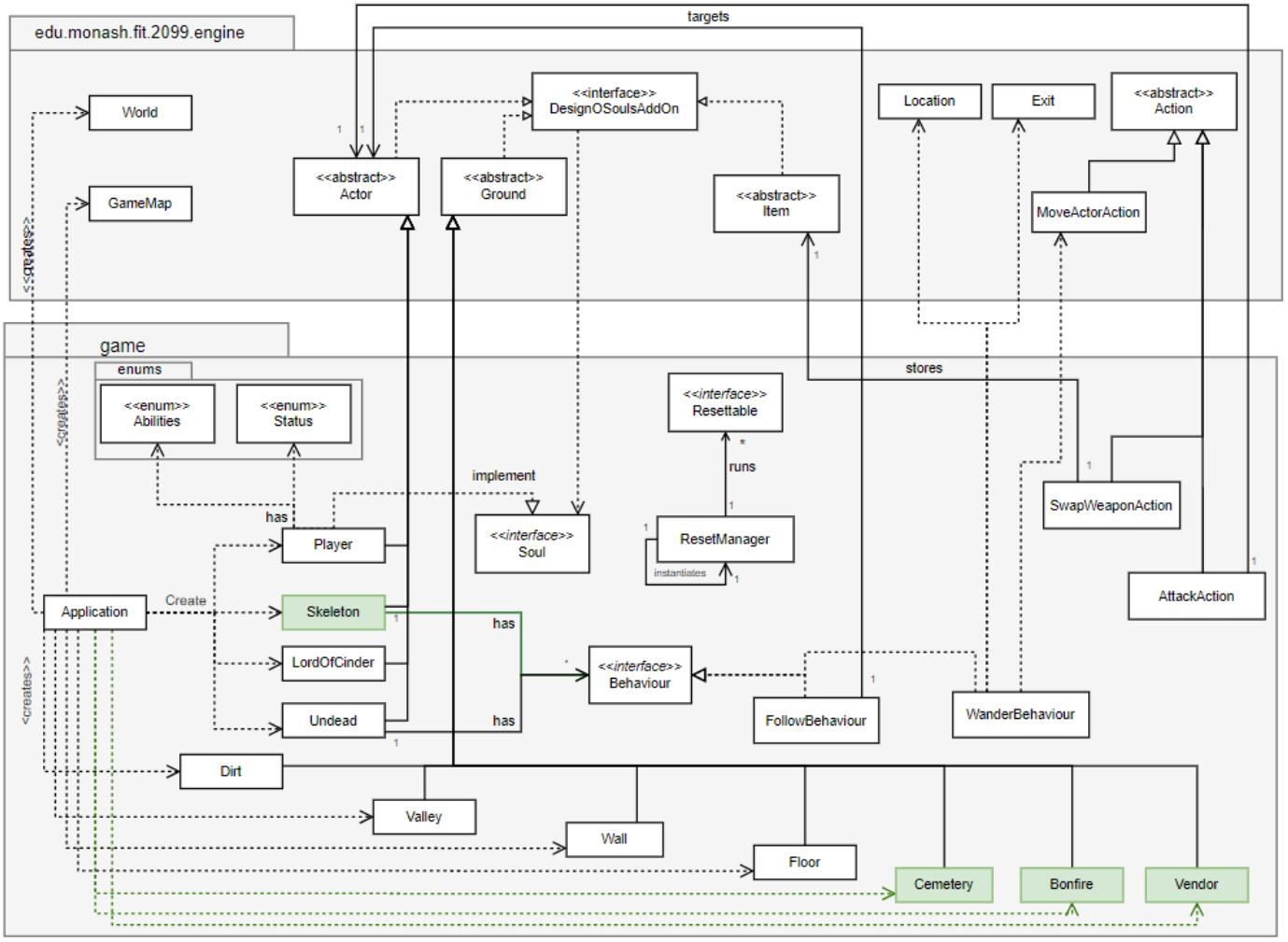
Design Rationale

FIT2099 Team 8

Team Members:

Chok Ming Jie	31869157
Soh Jin Huei	30721040
Rickie Cheong Jun Weng	30897580

Class Diagram 1



This class diagram shows the relationship between the classes in the engine package and game package. In this diagram, new classes are created and represented as green coloured boxes, and are placed in the game package. Due to the requirements, the new classes needed are the Skeleton class, Cemetery class, Bonfire class and Vendor class. These new classes are added as they are required when the world starts, which will be created by the Application class in the game map of the world.

Existing Classes:

- Abstract Classes: Actor, Ground, Item, Action
- Interface Classes: DesignOSoulAddOn, Resettable, Behaviour, Soul
- Enum: Abilities, Status
- Application, Dirt, Valley, Wall, Floor, FollowBehaviour, WanderBehaviour, SwapWeapnAction, AttackAction, Undead, LordOfCinder, Player, ResetManager, World, GameMap, Location, Exit, MoveActorAction

New Classes:

Skeleton

This class is used to create an instance of an enemy of the type Skeleton. We create 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 Skeletons in the game by just simply re-using the class to create new Skeletons.

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. This class has an association relationship with the Behaviour Interface. It is designed in this way so that the Skeleton has many behaviours. A Skeleton is an enemy that cannot be controlled by the Player, hence we have an array of behaviours as attributes for the Skeleton such as WanderBehaviour and FollowBehaviour. The relationship

between the Skeleton Class and the Application is that when the game starts, the Application class creates several Skeleton objects on the map.

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 adheres to low coupling. 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. It has the characteristics of the Ground Class such as the actors (i.e. Enemies) not being able to enter and other methods in the Ground Class. Additionally, the Application class has a dependency relationship with the Cemetery Class as when the game starts, it will create several Cemetery objects on the 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. 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. 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.

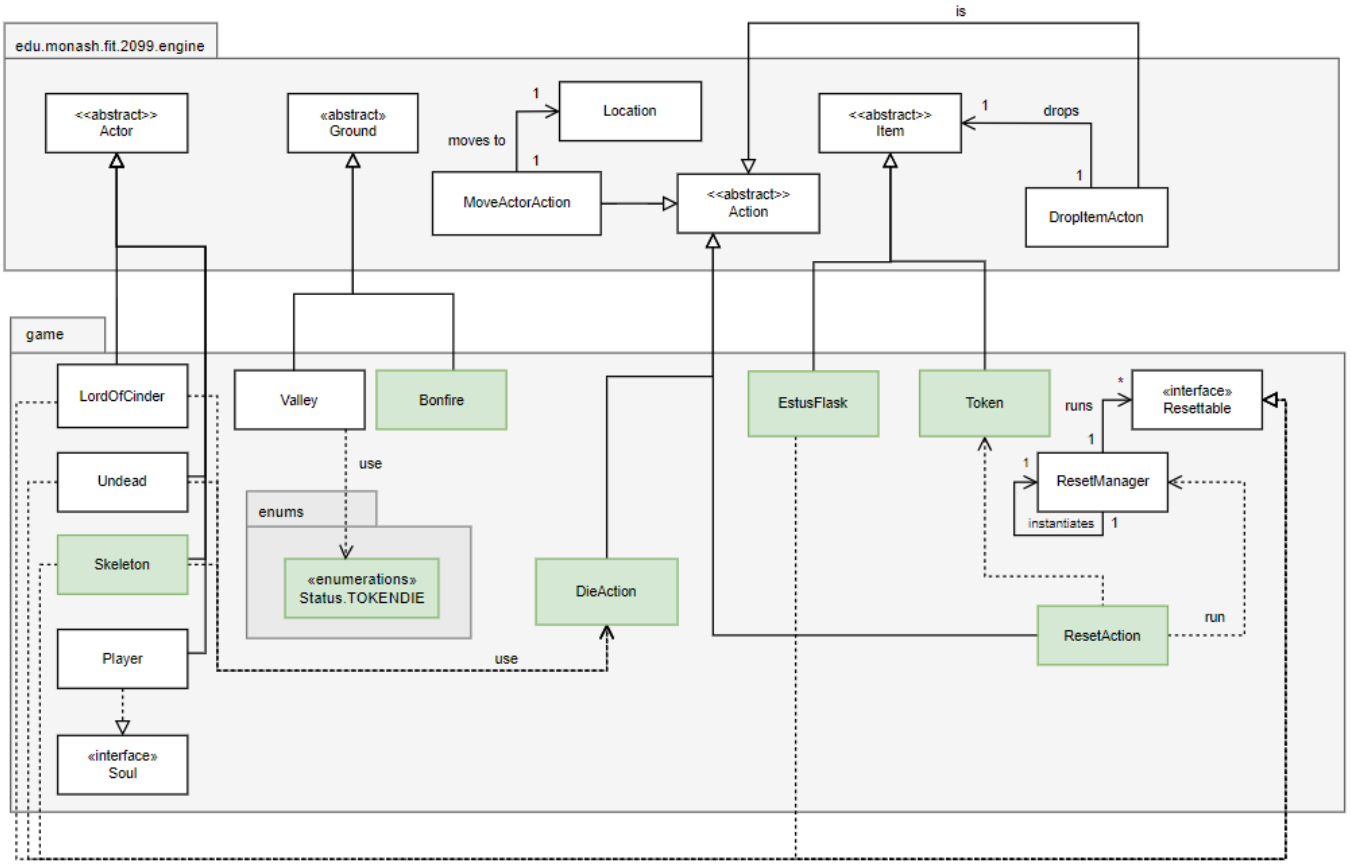
It extends to the Ground Class and by doing so, it inherits the method from the Ground Class as it will stay permanently in the game map. It has the characteristics of the Ground Class such as actors (i.e. Enemies) not being able to enter and other methods in the Ground Class. Additionally, the Application class has a dependency relationship with the Bonfire Class as when the game starts, it will create a Bonfire object on the map.

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 adheres to low coupling. Additionally, we can easily increase the number of Vendor objects in the map by simply using this class to create a new Vendor object.

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 and it has the characteristics of the Ground Class such as actors (i.e. Enemies) not being able to enter and other methods in the Ground Class. Additionally, the Application class has a dependency relationship with the Vendor Class as when the game starts, it will create a Vendor object which places it in the Firelink Shrink on the map.

Class Diagram 2



This class diagram shows the relationship between the Actor class and other classes when an actor dies. In this diagram, we show how the Player dies due to either stepping in the Valley or getting killed by the enemies, such as the Skeleton, Undead and the LordOfCinder. The new classes are created and represented as green coloured boxes, and are placed in the game package while others are existing classes that already exist in the system. New classes are added so that we satisfy the RESET features according to the requirements.

Existing classes:

- Abstract classes: Actor, Ground, Action, Item
- Interface classes: Resettable, Soul
- Enum class: Status.TOKENDIE
- Location, MoveActorAction, DropItemAction, LordOfCinder, Undead, Player, Valley, ResetManager

New Classes:

Skeleton + DieAction

The Skeleton class is used to create an instance of an enemy of the type Skeleton. We create 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 Skeletons in the game by just simply re-using the class to create new Skeletons.

The action death is created by having a DieAction Class. This class is used to focus on the transferring of souls from one actor to another as well as the removal of actors, except the player from the map. It is applied to the actors (i.e. Enemies) that are killed by the Player so that the souls can be transferred to the Player. We created this class so that when the actors (i.e. Enemies) die, they are removed from the map. This makes the system work better as it has an action that is able to 'clean' the map.

The DieAction Class is extended to the Action Class as it is an action to be used by the enemies. By doing so, it will inherit the methods from the Action Class. Additionally, the Skeleton Class and the DieAction Class have a dependency relationship. This is because the Skeleton Class uses this action to remove themselves from the map and transfer souls when they are killed. This is the same with the rest of the enemy classes.

Skeleton - Resettable

The relationship between the Skeleton class and the Resettable interface is a generalisation as the Skeleton class is resettable when the player dies. By doing so, the Skeleton class inherits the method from the Resettable interface. In the Resettable Interface, the Skeleton Class will use the registerInstance method to be added in the ResetManager so that it can be resetted. This is the same for other actors that need to be resetted when the player dies.

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. 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. 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.

In this diagram, the Bonfire is related when the Player dies, the Player will be moved to the location beside the Bonfire. In the engine package, there is a MoveActorAction that will be used to move the Player to the Bonfire.

Enum: Status.TOKENDIE

The relationship between this enum and the Valley Class is a dependency. This status is created for the Valley class so that when the Player dies due to stepping in the valley, this enum will be activated and used by the valley. The token is then dropped by the Player and will be placed at the location of the Player's death.

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 methods from the Item Class. It has the relationship of generalisation with Resettable Interface as it needs to be resetted when the Player dies. This class can only be used by the Player three times. It resets when the player dies and refills back to the maximum amount of charges which is three.

Token

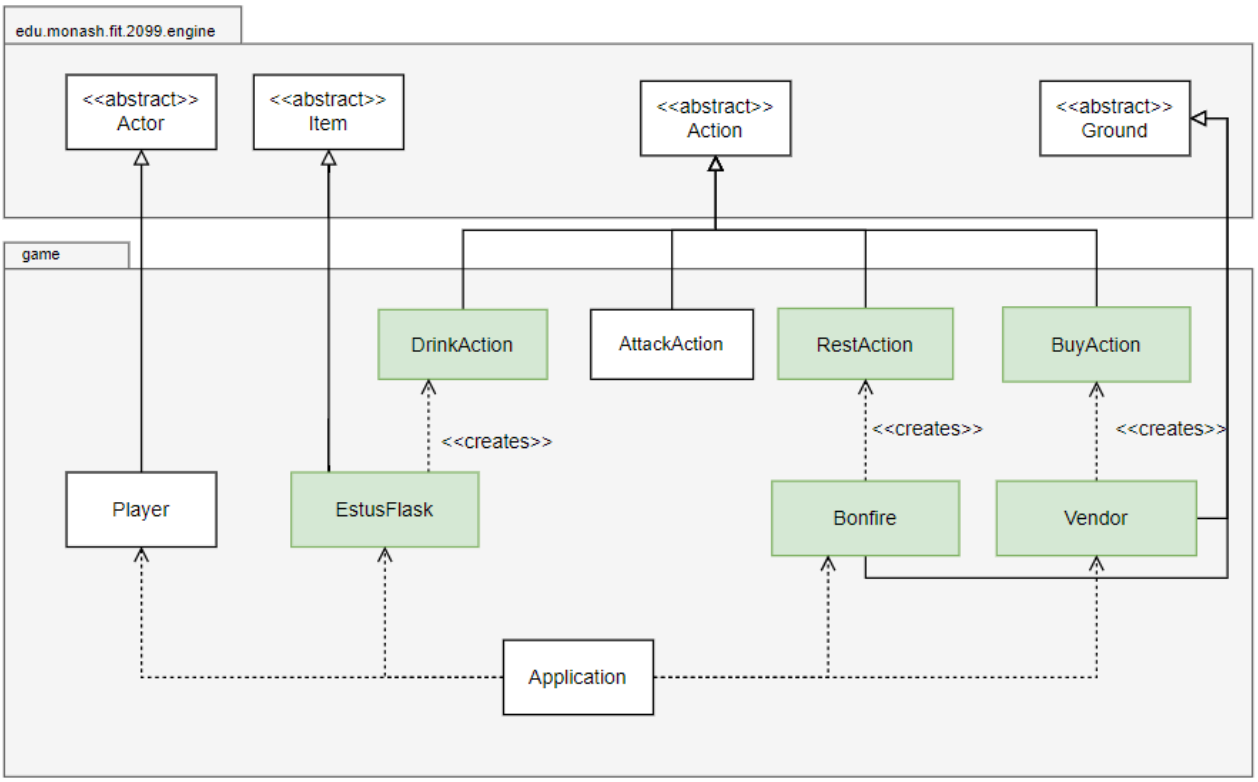
This class is used to create a Token object when the Player dies in the game. We created this class so that we can easily change its characteristics while not affecting the others of the system. This also ensures that the meaning of each class is clear and that it can be reused in other scenarios which adheres to low coupling. We created this class so that it can interact with the player by using a PickUpAction in order to pick up the token.

It extends to Item Class and by doing so, it inherits the methods from the Item class and has the methods for allowing the Player to pick it up. When the Player dies, a Token will be created and placed on the location of the player's death which can later be picked up by the player. The relationship between Token Class and the ResetAction Class is a dependency as the ResetAction will create a token and place it at the Player's dying location.

Reset Action

This class is used to execute the ResetManager in the system so that all things considered as Resettables are executed. To satisfy the requirements of avoiding the player from being removed from the map when it is killed, this class executes. When it is executed, it restarts the game, returns the player back to its starting point (Bonfire) and generates a token at the location of the player's death. 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.

Class Diagram 3



This class diagram shows the relationships between the Player Class and Action Class. This diagram shows the actions that the player has when interacting with a specific item or ground in the game. New classes are created and represented as green coloured boxes, and are placed in the game package while others are existing classes that already exist in the system. New classes are added so that the Player has specific actions to interact with.

Existing classes:

- Abstract classes: Actor, Ground, Action, Item
- Player, Application, AttackAction

New Classes:

EstusFlask + DrinkAction

The DrinkAction class extends to the Action Class. By doing so, it inherits the Action Class and the necessary methods to provide a message before (menuDescription) and after the

class executes. The EstusFlask Class creates the DrinkAction Class in order for the Player to interact with it. This is an ideal way to interact with the object, attaching appropriate actions to its corresponding object. The EstusFlask is the object that gives the Player an action to be used. The Player can use this action to drink the EstusFlask.

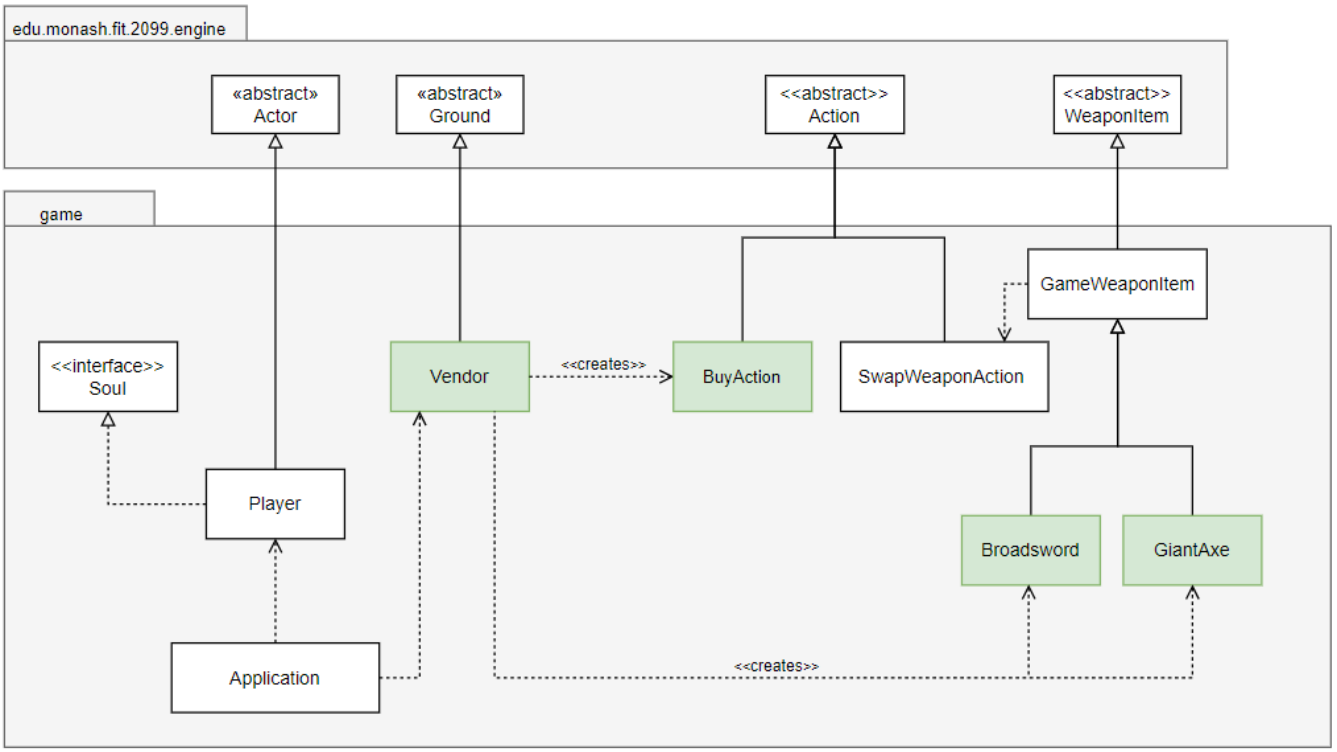
Bonfire + RestAction

The RestAction Class extends to the Action Class. By doing so, it inherits the Action class and the necessary methods to provide a message before (menuDescription) and after the class executes. The Bonfire Class creates the RestAction Class in order for the Player to interact with it. This is an ideal way to interact with the object by attaching appropriate actions to its corresponding object. The Bonfire is the object that gives the Player an action to be used. The Player can then enter the Bonfire and use the action to rest. This RestAction is provided by the Bonfire Class and it will use the ResetAction to reset the world, however it doesn't move the Player and it creates a token.

Vendor + BuyAction

The BuyAction Class extends to the Action Class. By doing so, it inherits the Action class and the necessary methods to provide a message before (menuDescription) and after the class executes. The Vendor Class creates the BuyAction Class in order for the Player to interact with it. This is an ideal way to interact with the object, by attaching appropriate actions to its corresponding object. The Vendor is the object that gives the Player an action to be used. The Player can enter the Vendor and use the action to make a purchase. This BuyAction is provided by the Vendor.

Class Diagram 4



This class diagram shows the relationships between the Player and Vendor class when a player purchases an item from the vendor. In this diagram, new classes are created and represented as green coloured boxes, and are placed in the game package. Others are the existing classes already in the system. New classes are added in order to adhere to the requirements given to us.

Existing Classes:

- Abstract Classes: Actor, Ground, Action, Weapon Item
- Interface: Soul
- Player, Application, SwapWeaponAction, GameWeaponItem

New Classes:

Vendor + BuyAction

The Vendor class is created to represent the vendor that sells and trades items with the player. The vendor is said to locate one within the boundaries of the Fireline Shrink and does not exist anywhere else on the map. This class extends to the Ground Class as the vendor is a permanent place on the map and it will not move anywhere else. The Application class creates the Vendor class when the system runs as it is considered a Ground object, hence its relationship in the diagram above. Having a Vendor Class will also allow us to create any additional vendors on the map in the future.

The BuyAction Class is created in order for a player to perform the buying action when interacting with the vendor class. This class extends to the Action Class which inherits the necessary methods like providing a message when the action is being performed in order to let the user know. As well as the execute method which allows the action to take place. Having its own class will also allow us to implement this action for any other actors in the game.

An object can only be interacted with if an appropriate action is attached to the corresponding object. The Vendor is the object that gives the Player an action that it can interact with. Hence, it creates a BuyAction in order to be interacted with.

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.

It extends to the GameWeaponItem Class which then extends to the WeaponItem Class as it is considered a weapon that can be used by an actor. By doing so, it inherits the necessary methods in order to activate the characteristics of the weapon. The relationship between

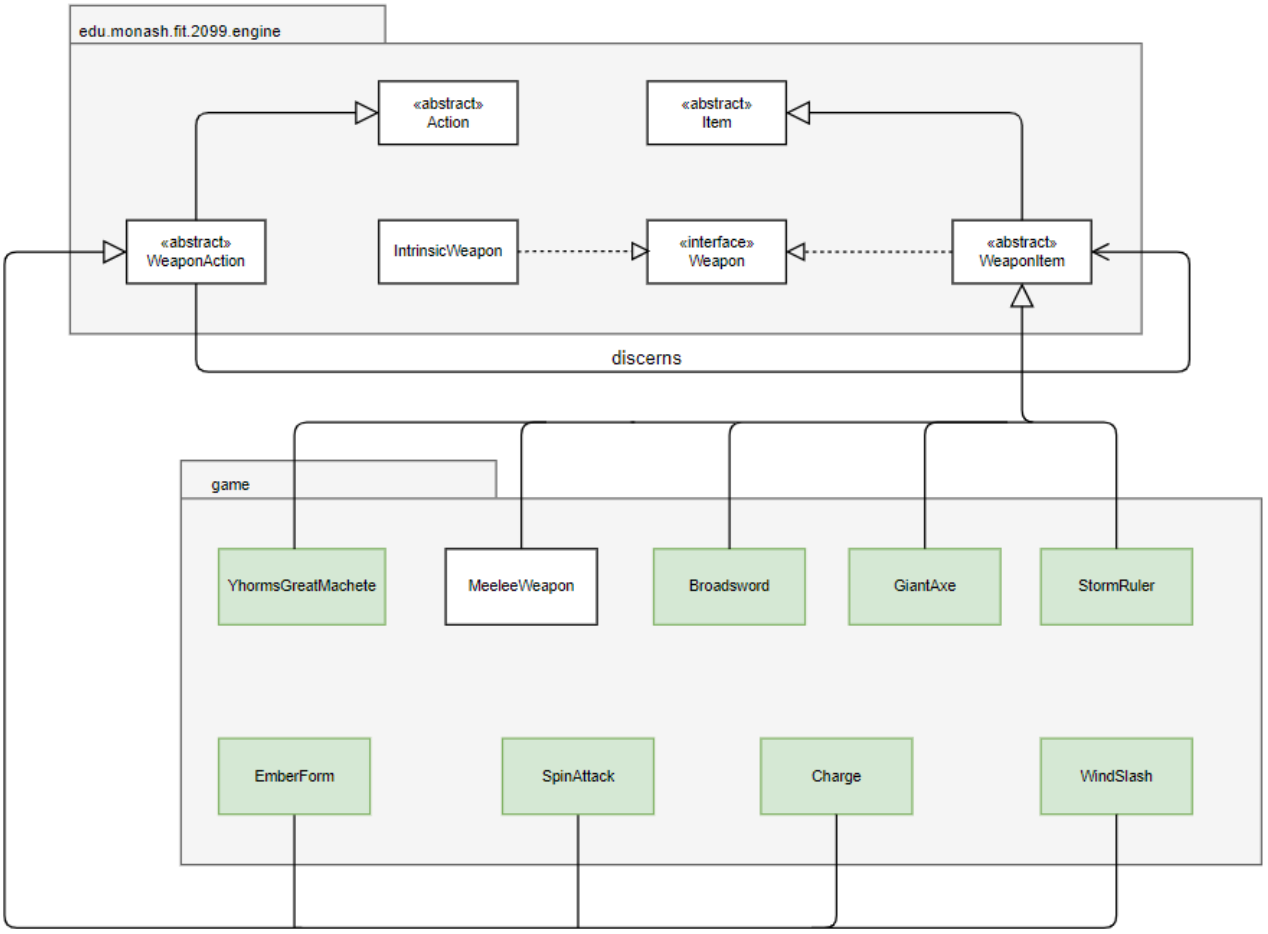
the Broadsword Class and the Vendor Class is that when the system runs, the Vendor Class creates the broadsword which will be made available to the player when it purchases the weapon using souls at the start of the game.

Giant Axe

This class is used to create a Giant Axe object instead. Just like the Broadsword Class, 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.

Since it is also considered a weapon, it extends the GameWeaponItem Class and the WeaponItem Class in order to inherit the necessary methods to activate the characteristics of the weapon. The relationship between the GiantAxe Class and the Vendor Class is that when the system runs, the Vendor class creates the giant axe which will be made available to the player when it purchases the giant weapon using souls at the start of the game.

Class Diagram 5



This class diagram shows the relationships between the Weapon Interface and the WeaponAction Class as it represents the available weapons and its skills in the game. In this diagram, new classes are created and represented as green coloured boxes while the rest are existing classes in the system. The new classes satisfy the requirements given to us and are placed in the game package.

Existing Classes:

- Abstract Classes: Action, Item, WeaponItem, WeaponAction
- Interface: Weapon
- Classes: IntrinsicWeapon, MeleeWeapon

New 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 being 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 WeaponItem 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 WeaponItem Class which further extends to the Item Class as it is also considered a type of item in the game.

Giant Axe

This class is used to create a Giant Axe object instead. Just like the Broadsword Class, 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. This class extends to the WeaponItem Class as it is a weapon that can be used by an actor. 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 be much easier and maintainable. Like Yhorm's Great Machete, it is also considered a weapon as well as a type of item in the game. Hence, StormRuler extends to the WeaponItem Class which further extends to the Item Class.

Ember Form

This class is used to contain the logic for the skill called Ember Form. This skill is provided by the weapon Yhorm's Great Machete. This class was created in order to allow us to set unique characteristics that other skills may not obtain. Although it is a skill of the weapon Yhrom's Great Machete, we decided that there would be no relationship between the EmberForm Class and YhormsGreatMachete Class in the diagram above. The reason being, it provides flexibility in terms of allowing other weapons to possibly inherit this skill, if needed in the future which adheres to the coding practice DRY. This skill is also categorized as an active skill hence, it extends to the WeaponAction Class as it represents the active skills of all the different weapons in the game. This class then further extends to the Action Class as it is considered a type of action.

Spin Attack

This class is used to contain the logic behind the skill Spin Attack, a skill provided by the weapon Giant Axe. This class was created in order for us to set its unique characteristics without affecting the rest of the different skills. Despite it being a skill of a Giant Axe, there is no relationship between the GiantAxe Class and SpinAttack Class. This is because we have decided that by doing so, this skill will not be limited in allowing only one weapon to inherit it if other weapons/items were to inherit it in the future. Additionally, it adheres to the coding practice DRY if future requirements include (new) weapons to inherit this skill. Since this skill is also categorised as a type of an active skill, it extends to the WeaponAction Class as it represents the active skills of all the different weapons in the game. This class then further extends to the Action Class as it is considered a type of action.

Charge

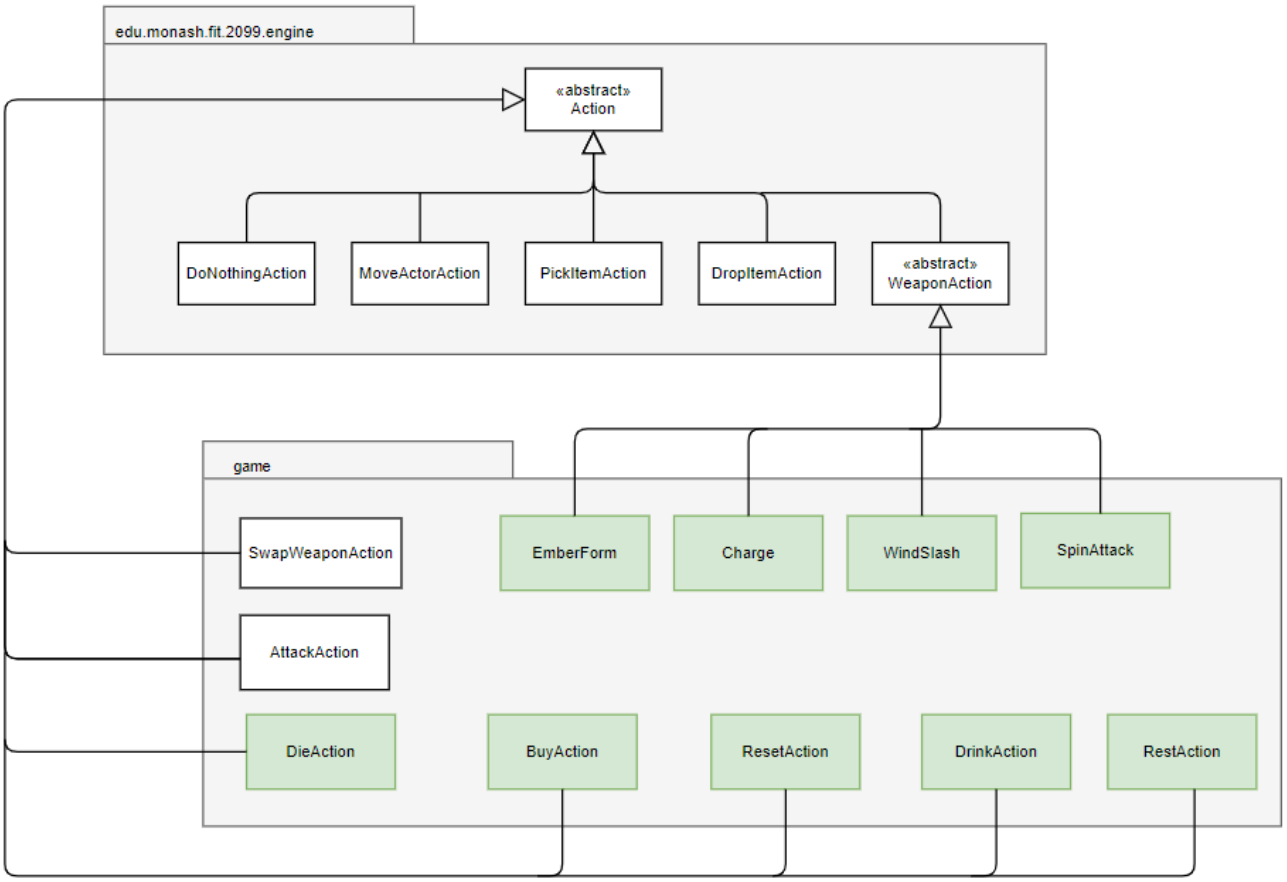
This class is used to contain the logic of Charge, a skill provided by the weapon StormRuler. Having its own class allows us to add and change the features that the skill provides without affecting the rest of the system. Although this skill comes from the weapon StormRuler, there is no relationship between the Charge Class and the StormRuler Class as presented in the diagram above. The reason behind this is that it will not be restricted to only having one weapon that can inherit this skill. This means that if other weapons/items need this skill to be implemented, it can be done easily without the need of repeating the same code. This

adheres to the coding practice DRY. This class extends to the WeaponAction Class as it is categorized as a type of active skill. Additionally, this class then further extends to the Action Class as it is considered a type of action.

Wind Slash

This class is used to contain the logic for the skill Wind Slash. Like the Charge Class, it is also a skill that comes from the weapon StormRuler. Having its own class allows us to add and change the features that the skill provides without affecting the rest of the system. As shown in the diagram, there is also no relationship between the WindSlash Class and the StormRuler Class. This is because we have decided that by doing this, it provides flexibility in terms of allowing other weapons to possibly inherit this skill, if needed in the future which also helps follow coding practice DRY. This class is also another type of active skill and hence, it extends to the WeaponAction Class which then further extends to the Action Class as it is categorized as a type of action.

Class Diagram 6



This class diagram shows the relationships between the different types of actions that are available in the game. It represents actions for both the actors and weapons in the game. In this diagram, new classes are created and represented as green coloured boxes while the rest are existing classes in the system. The new classes satisfy the requirements given to us and are placed in the game package.

Existing Classes:

- Abstract Classes: Action, WeaponAction
- DoNothingAction, MoveActorAction, PickItemAction, DropItemAction, SwapWeaponAction, AttackAction

New Classes:

Die Action

This class is used to focus on the transferring of souls from one actor to another as well as the removal of actors, except the player from the map. Having its own class allows us to apply it to the other actors in the game (i.e. Enemies) that are killed by the player. By creating this class, we reduce the possibility of repeating codes. Since this is a type of action, it extends to the Action Class which inherits the necessary methods in order to take place such as providing a message and the execute method.

Buy Action

The BuyAction class is created in order for a player to perform the buying action when interacting with the vendor class. This class extends to the Action Class which inherits the necessary methods like providing a message when the action is being performed in order to let the user know. As well as the execute method which allows the action to take place. Having its own class will also allow us to implement this action for any other actors in the game.

Reset Action

This class is used to execute the reset manager in the system so that all things considered as Resettables are executed. To satisfy the requirements of avoiding the player from being removed from the map when it is killed, this class executes. When it is executed, it restarts the game, returns the player back to its starting point (Bonfire) and generates a token at the location of the player's death. 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.

Drink Action

This class is used in order for a player to interact with the EstusFlask object. 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. This includes providing a message

when the action is being performed in order to let the user know. As well as the execute method which allows the action to take place.

Rest Action

This class is used for the player when it wants to rest at the Bonfire when the status.REST is detected, in doing so it triggers all the reset features that take place in the game. As this is considered a type of action, it extends to the Action Class which also inherits the necessary methods for this class to be executed during the game.

Ember Form

This class is used to contain the logic for the skill called Ember Form. This skill is provided by the weapon Yhorm's Great Machete. This class was created in order to allow us to set unique characteristics that other skills may not obtain. This class extends to the WeaponAction Class as it is categorized as a type of active skill. It then further extends to the Action Class because it is also considered a type of action.

Spin Attack

This class is used to contain the logic behind the skill Spin Attack, a skill provided by the weapon Giant Axe. This class was created in order for us to set its unique characteristics without affecting the rest of the different skills. This class extends to the WeaponAction Class as it is categorized as a type of active skill. It then further extends to the Action Class because it is also considered a type of action.

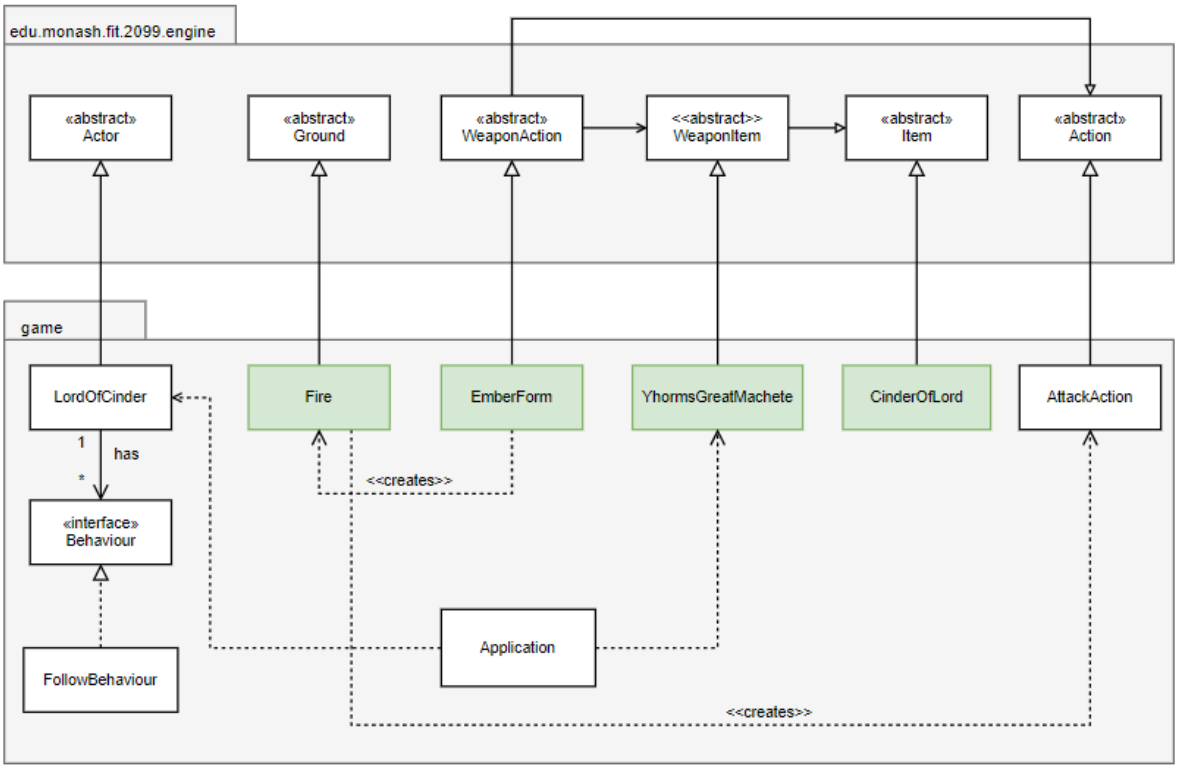
Charge

This class is used to contain the logic of Charge, a skill provided by the weapon StormRuler. Having its own class allows us to add and change the features that the skill provides without affecting the rest of the systemThis class extends to the WeaponAction Class as it is categorized as a type of active skill. It then further extends to the Action Class because it is also considered a type of action.

Wind Slash

This class is used to contain the logic for the skill Wind Slash. Like the Charge Class, it is also a skill that comes from the weapon StormRuler. Having its own class allows us to add and change the features that the skill provides without affecting the rest of the system. This class extends to the WeaponAction Class as it is categorized as a type of active skill. It then further extends to the Action Class because it is also considered a type of action.

Class Diagram 7



This class diagram shows the relationship between the Lord of Cinder and it's related classes. This represents the possible actions and behaviour that is possible for the Lord of Cinder. The green boxes shown here are classes that need to be created, meanwhile the rest are classes and interfaces that already exist.

Existing Classes:

- Abstract Classes: Actor, Ground, WeaponAction, WeaponItem, Item, Action,
- Interface: Behaviour
- LordOfCinder, FollowBehaviour, Application, AttackAction

New Classes:

EmberForm + Fire

The Fire class is created to represent the fire that is emitted from Yhorm the Giant (Lord of Cinder) when it is in it's Ember Form and is attacking. It burns the surrounding area and damages the player for 25 hit points except itself. The Fire class extends to the Ground Abstract class since the Fire would be temporarily on the map and won't move for 3 turns when it is originally created. The AttackAction class and Ember Form create the Fire class when Yhorm The Giant's Ember Form is active and attacking, thus justifying the relationship between the classes above. Having a Fire class would also allow us to create any fire on the map if we so wish.

The EmberForm class is created in order for Yhorm The Giant to enter it's Ember Form whenever it is holding its weapon (Yhorm's Great Machete) that only he can equip. This Form also allows for an increased successful hit rate by 30%, combined with the original 60%, that would make it 90%. This class extends to the WeaponAction class which inherits the necessary methods such as "WeaponAction" and "execute" which are the capabilities of the weapon as well as for the action to take place. Having its own class will allow us to implement this action whenever we want in the game.

An object can only be interacted with if an appropriate action is attached to the corresponding object. The Ember Form is the object that gives the Boss (Yhorm the Giant) an action to interact with, such as creating Fire.

Yhorm's Great Machete

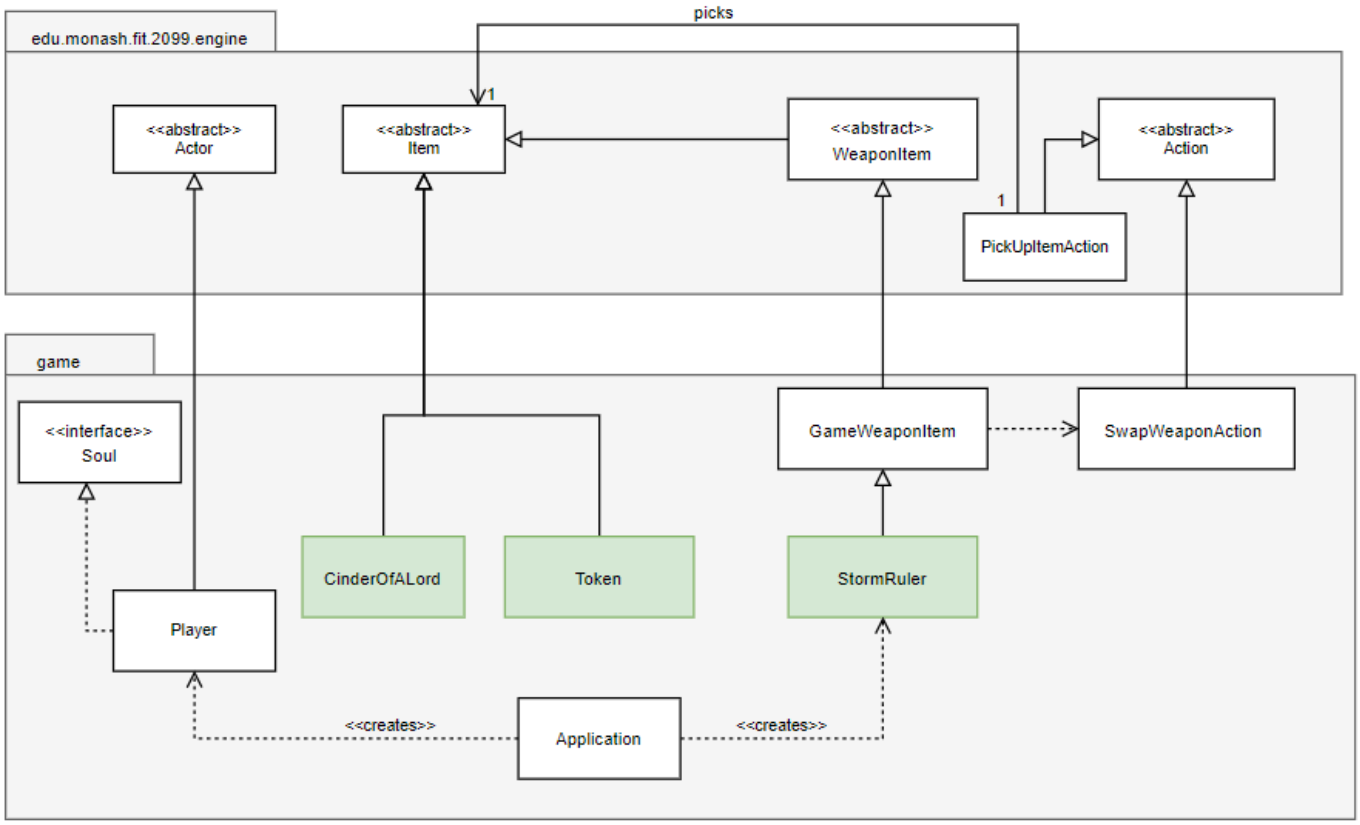
This class's main purpose is to create Yhorm's Great Machete, a weapon that can only be used by Yhorm the Giant (Lord of Cinder). This class was created with the main intention of making it easier for managing all the characteristics and limitations of the weapon. A few special characteristics of this weapon is that it is only to be equipped by Yhorm the Giant (Lord of Cinder), and that it allows Yhorm The Giant to enter its Ember Form. Since Yhorm's Great Machete is considered a weapon in the game, it extends upon the WeaponItem class.

Whenever the Lord of Cinder is created, the weapon is also created, therefore the Application class creates the machete.

Cinder of Lord

This class's main purpose is to create Cinder of Lord, an item that is dropped upon the death and defeat of Yhorm the Giant. The players can also choose to drop it in the Firelink Shrine in order to continue the story, temporarily, this has no other explanation for the continuation of the story. Since Cinder of Lord is also considered an item, it is an extension of the item class as it would make it easier to implement the characteristics. Therefore, whenever the Lord of Cinder is defeated, the Application class would create Cinder of Lord to drop to the player.

Class Diagram 8



This class diagram shows the relationships between the Items and Player. It shows what items a player can pick up in the game. In this diagram, new classes are created and represented as green coloured boxes while the rest are existing classes in the system. The new classes satisfy the requirements given to us and are placed in the game package.

Existing Classes:

- Abstract Classes: Actor, Item, WeaponItem, Action
- Interface: Soul
- PickUpItemAction, Player, Application, GameWeaponItem, SwapWeaponAction

New Classes:

Cinder of Lord

This class's main purpose is to create Cinder of Lord, an item that is dropped upon the death and defeat of Yhorm the Giant. The players can also choose to drop it in the Firelink Shrine in order to continue the story, temporarily, this has no other explanation for the continuation

of the story. Since Cinder of Lord is also considered an item, it is an extension of the item class as it would make it easier to implement the characteristics.

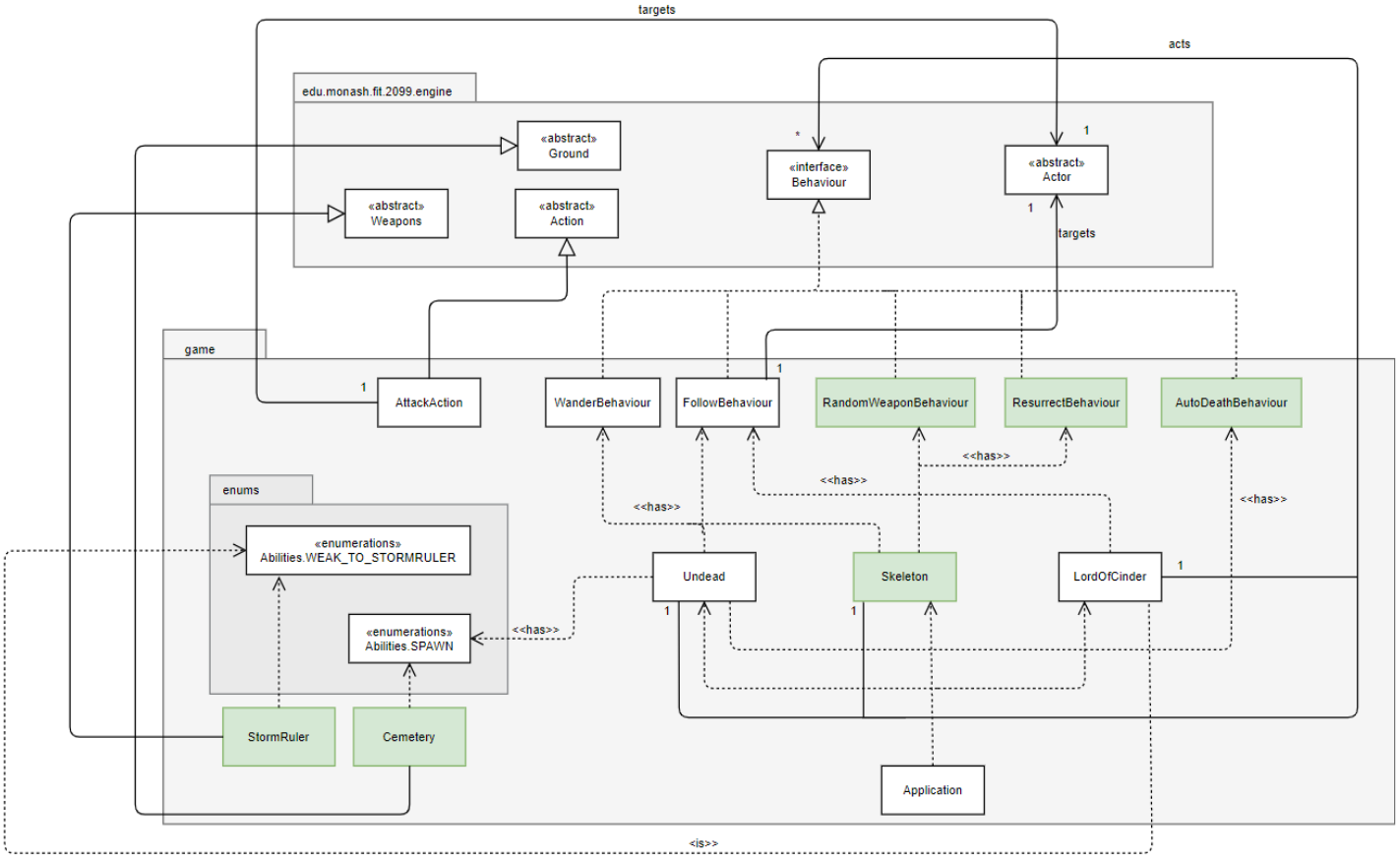
Token

The class's main purpose is to create Tokens. Upon the death of a player, the player will drop "token of souls" on the ground and have their souls resetted back to 0. This class extends to Item Class so that it inherits the methods from the Item class and has the methods for allowing the Player to pick it up. When the Player dies, it will create a Token Class and be placed on the ground which will later be picked up by the player.

StormRuler

This class is used to create a StormRuler object that will be placed next to Yhorm the Giant and it can be equipped by a player. By creating this class, it can help with the process of adding/changing characteristics provided by the weapon be much easier and maintainable. It is considered a weapon as well as a type of item in the game. Hence, StormRuler extends to the WeaponItem Class which further extends to the Item Class.

Class Diagram 9



This class diagram shows the relationships between the different Enemy classes and its Behaviours. It represents how the Enemies behave when the game is executed. In this diagram, new classes are created and represented as green coloured boxes while the rest are existing classes in the system. The new classes satisfy the requirements given to us and are placed in the game package.

Existing Classes:

- Abstract Classes: Ground, Weapon, Action, Actor
- Interface: Behaviour
- Enums: Abilities.WEAK_TO_STORMRULER, Abilities.SPAWN
- AttackAction, WanderBehaviour, FollowBehaviour, Undead, Application, LordOfCinder

New Classes:

StormRuler + Abilities.WEAK_TO_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 be much easier and maintainable. It is considered a weapon in the game. Hence, StormRuler extends to the WeaponItem Class which further extends to the Item Class.

Yhorm the Giant (Lord of Cinder) is also weak to this weapon, therefore it creates WEAK_TO_STORMRULER ability where the Lord of Cinder is weak against. This is created as every time the Lord of Cinder is striked with the weapon Storm Ruler, it would be weak against it.

Cemetery

This class is used for the main purpose of spawning the Undeads and they cannot be equipped with any weapons at all. They also wander around aimlessly and attack the player once they (the player) stand next to it (undead). Since they are spawned from the ground only if they are in the cemetery based on where they are standing hence, the class is an extension of the ground class. Therefore this justifies the relationship shown above between the classes. Another thing to note is that the Cemetery class creates the ability called SPAWN which spawns the undead.

Skeleton

The Skeleton class is used to create an instance of an enemy of the type Skeleton. We create 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 Skeletons in the game by just simply re-using the class to create new Skeletons. Therefore, the application class can create as many skeletons based on the requirements.

RandomWeaponBehaviour

The RandomWeaponBehaviour Class represents the behaviour of the Skeleton that allows them to equip a random weapon. By creating this class, it grants us the flexibility to edit/change characteristics of this behaviour without affecting the rest of the system. It also allows for reusability of the code if needed in the future. This class implements the Behaviour Interface in order to inherit the necessary methods for a behaviour to be executed. The relationship between the RandomWeaponBehaviour Class and the Skeleton Class shows that only a Skeleton is able to equip a random weapon.

ResurrectBehaviour

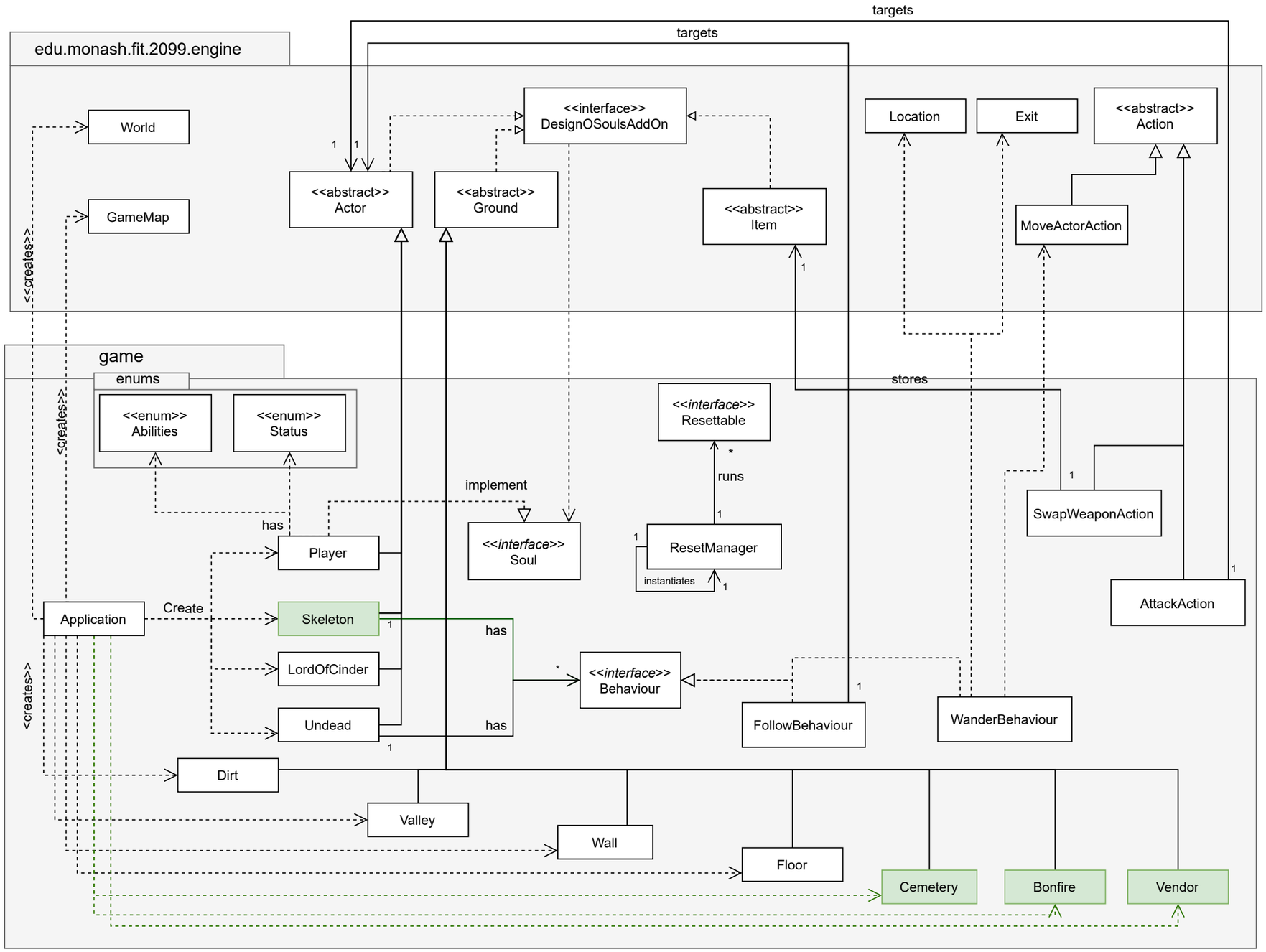
The ResurrectBehaviour class's main purpose is mainly for the Skeletons. This is because every time the Skeletons are defeated, they have a 50 percent chance of being resurrected, but only on their first death. They will not have any chance of being resurrected again if they have already been revived. We created this class because it grants us the flexibility to edit/change characteristics of this behaviour without affecting the rest of the system. It also allows for reusability of the code if needed in the future for other actors in the game. This class implements the Behaviour Interface in order to inherit the necessary methods for a behaviour to be executed which explains the relationship shown in the diagram above.

AutoDeathBehaviour

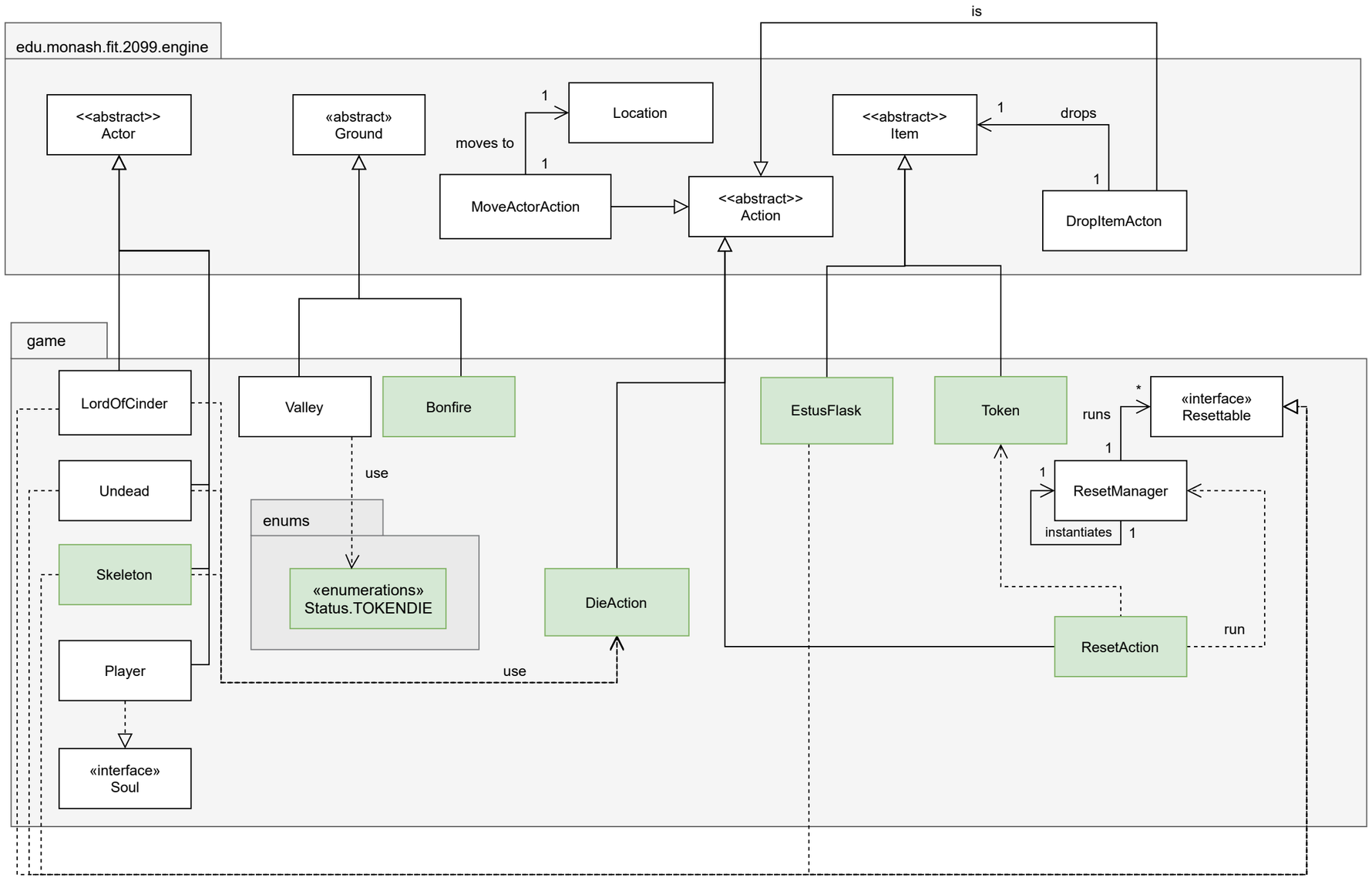
The AutoDeath behaviour class's main purpose is to be implemented with the Undead. This is because for every turn, the Undead has a 10 percent chance of dying. Creating this class grants us the flexibility to edit/change characteristics of this behaviour without affecting the rest of the system as well 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. Therefore, this explains the relationship between all the classes related to the ResurrectBehaviour class as shown in the diagram.

UML Diagrams

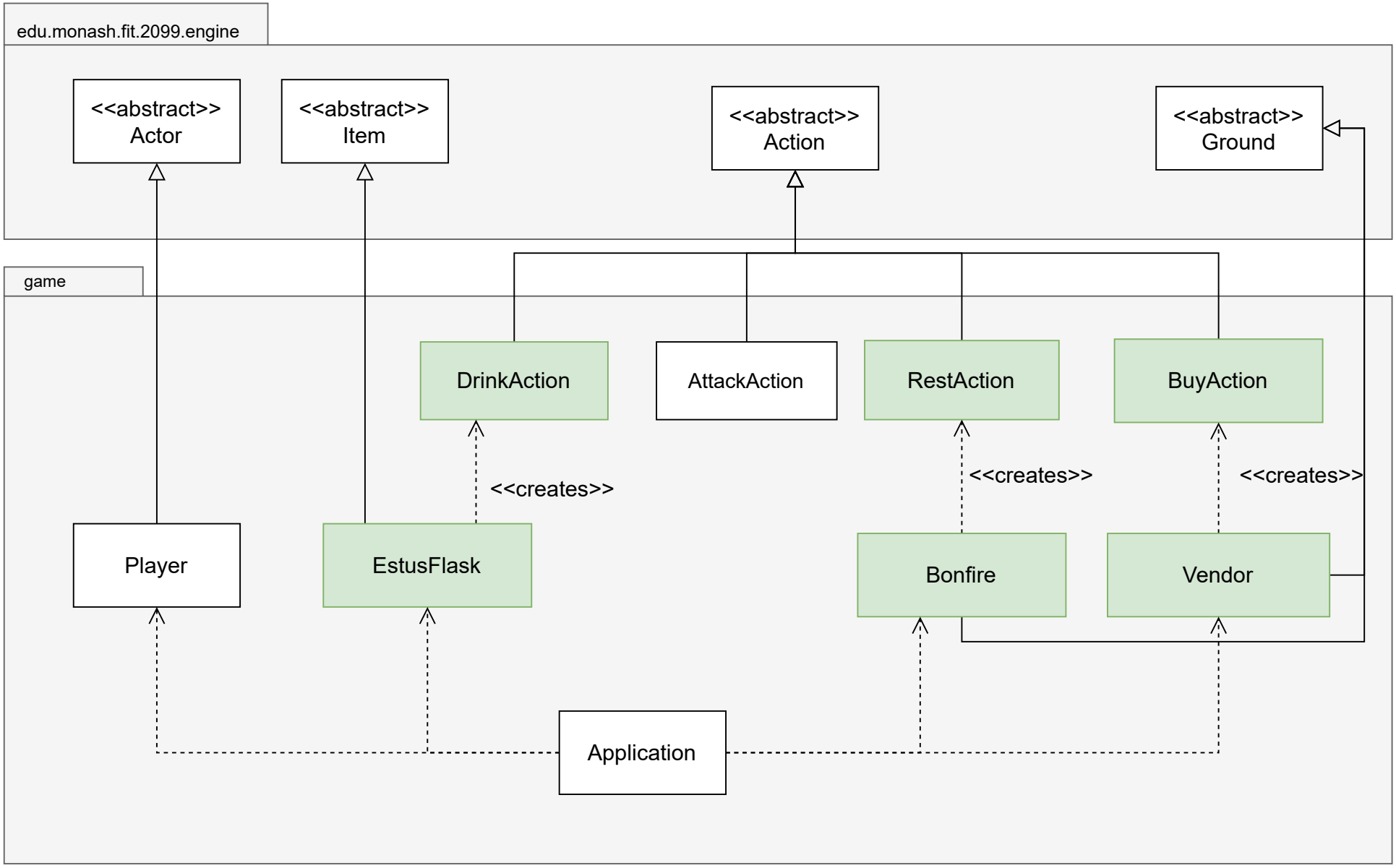
This class diagram shows the relationships between classes in the engine and game packages.



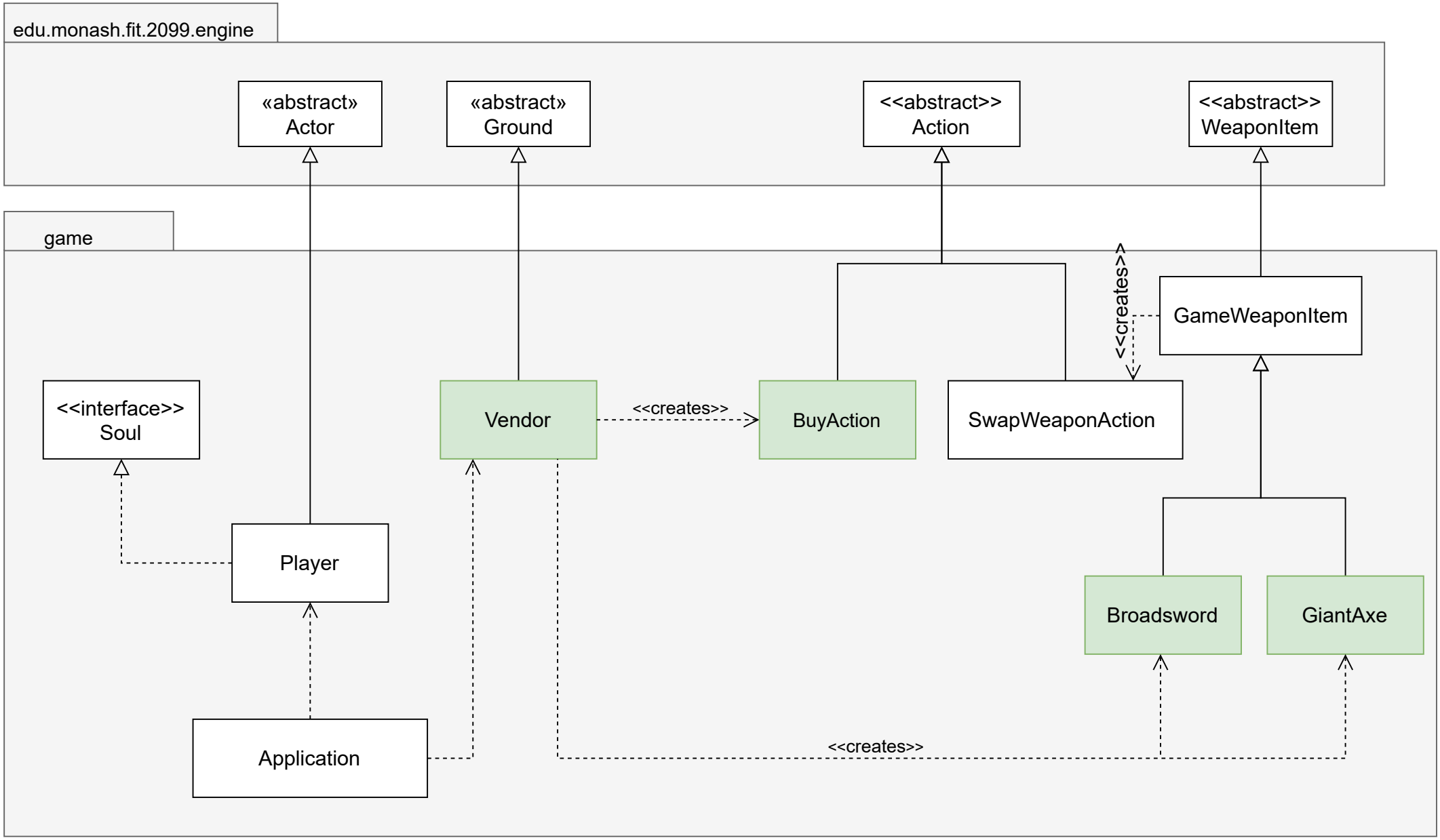
This class diagram shows the relationships between the classes when an actor dies in the game.



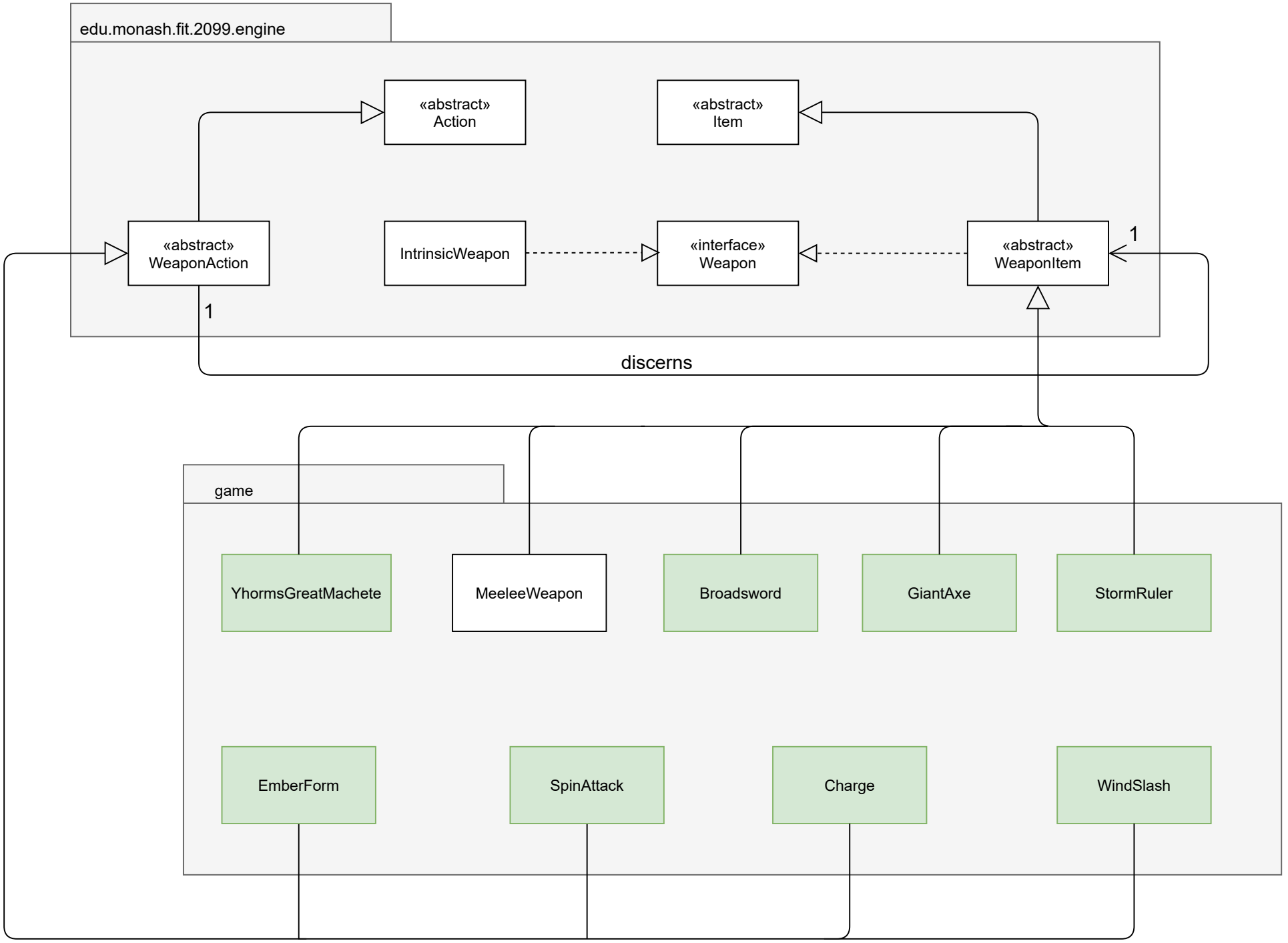
This class diagram shows the relationship between the Player and Action class.
It shows the available actions as a player.



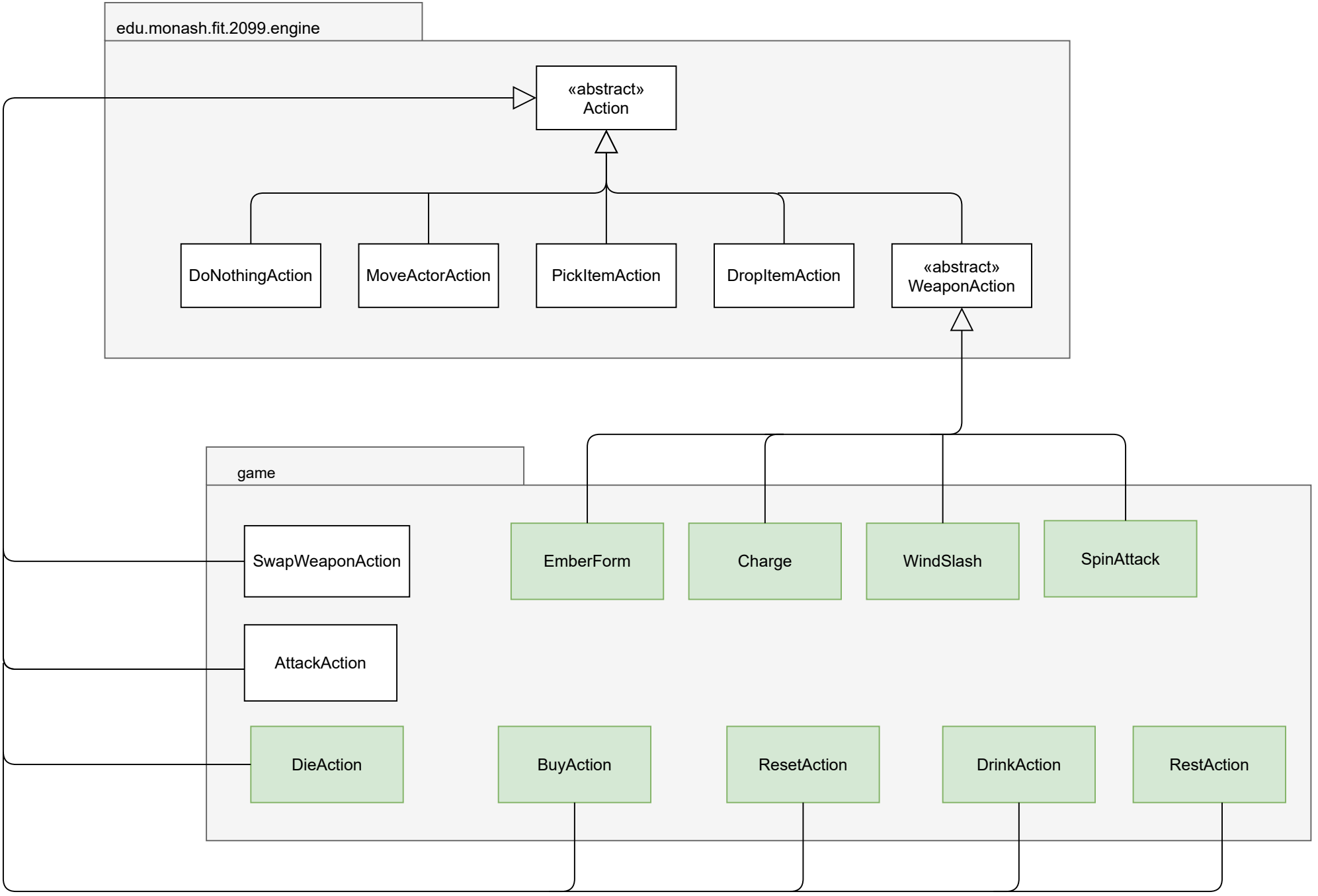
This class diagram shows the relationships between the Player, Vendor and Item classes.



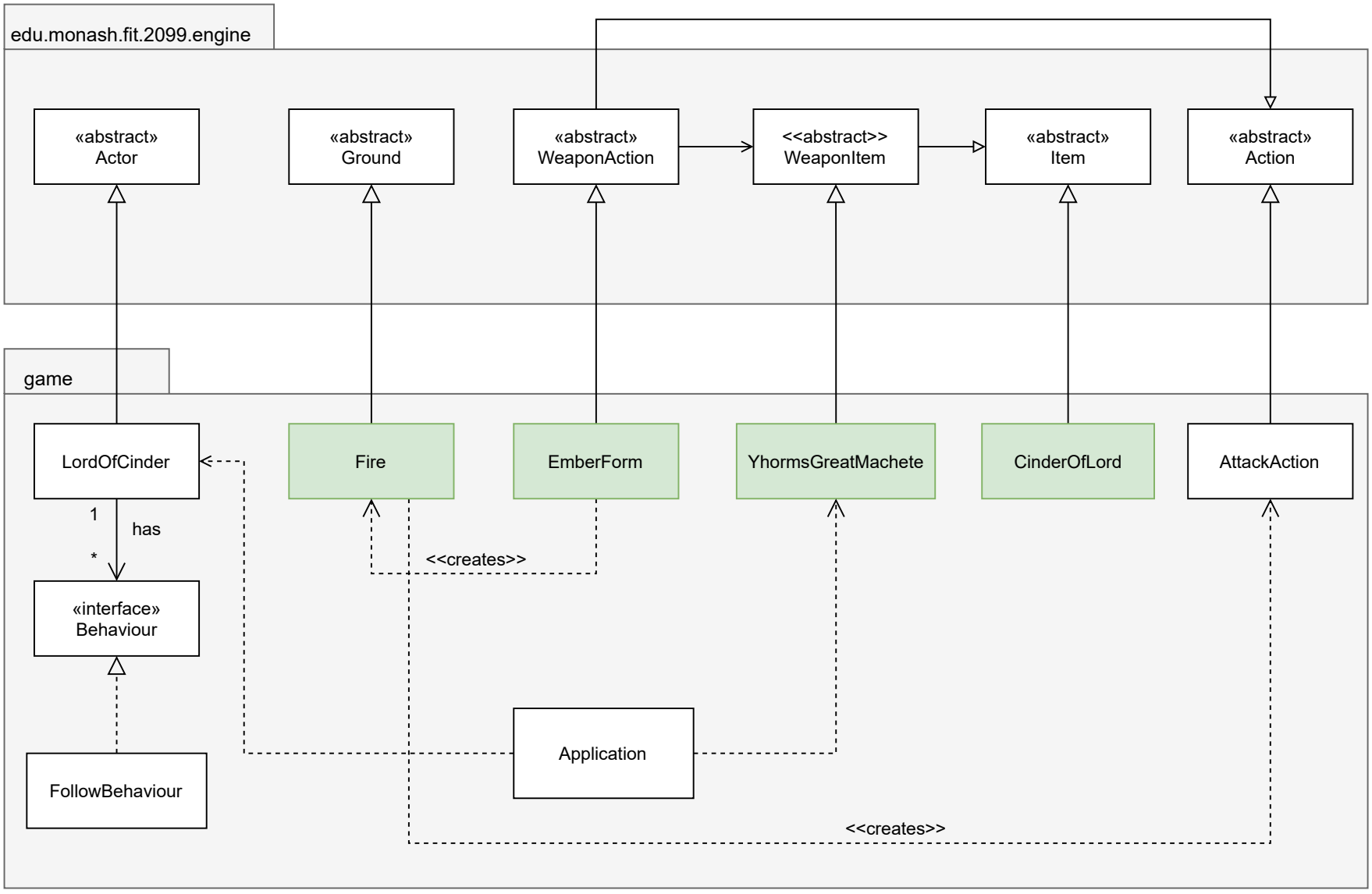
This class diagram shows the relationships between Weapons and Weapon Actions.



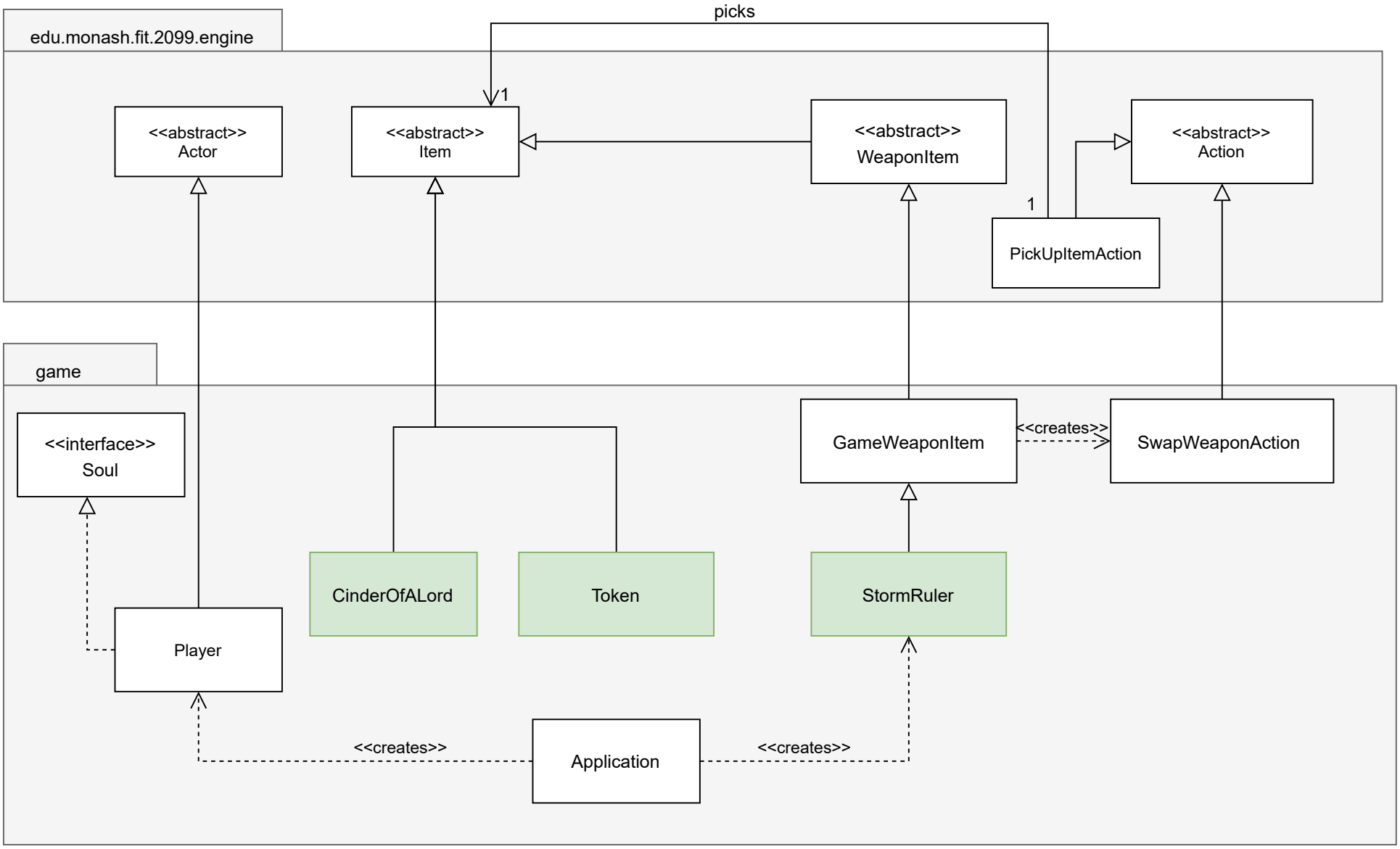
This class diagram shows the relationships between the different actions provided in the game.



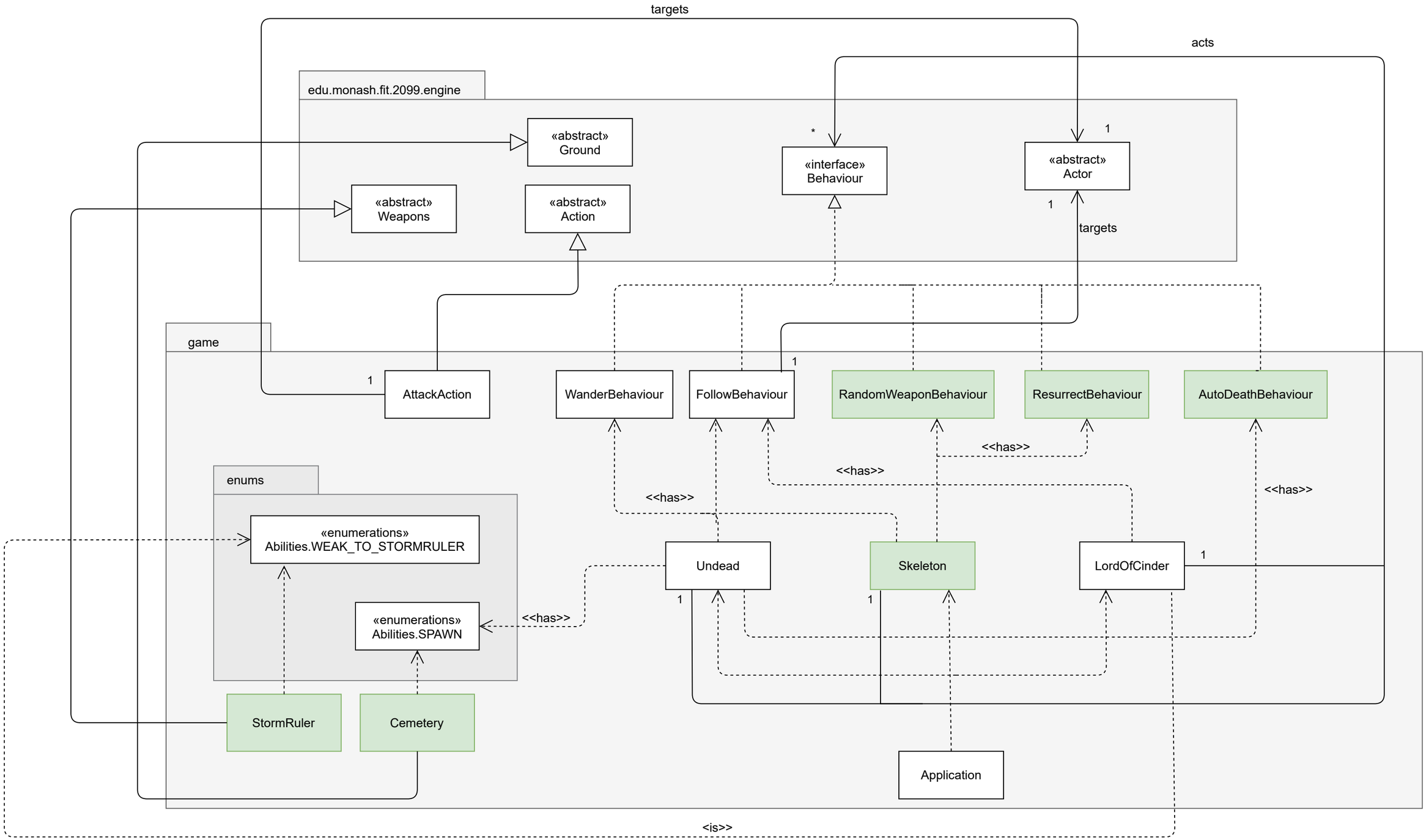
This class diagram shows the relationships of Lord Of Cinder and its related classes.



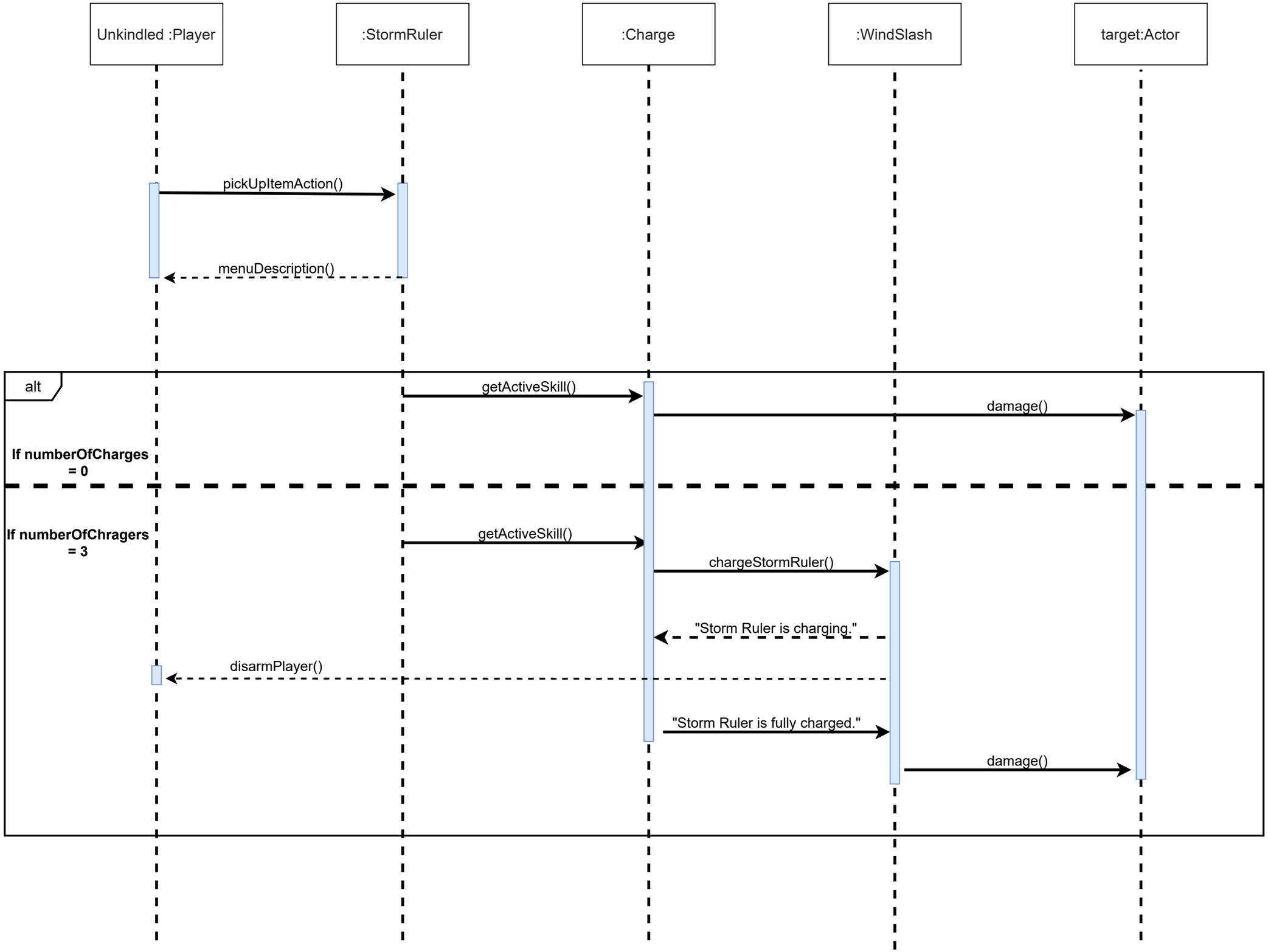
This class diagram shows the relationship between the Item that the Player can pick up



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



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



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