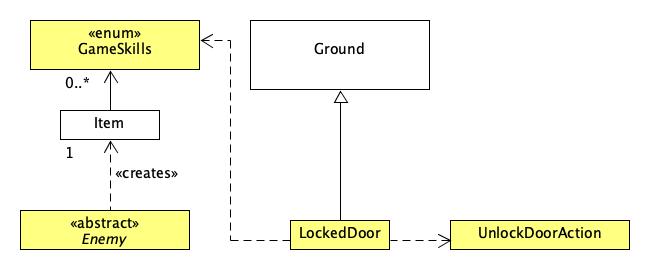
**Doors and keys**

Based on the **class diagram** above, the Enemy class creates the Item, key in a method. The key has the skill GameSkills.UNLOCKDOOR. The locked door can be unlocked if the player has a key with GameSkills.UNLOCKDOOR. So, the LockedDoor class depends on GameSkills. If the door can be unlocked, the LockedDoor class has an overridden method allowableActions that calls the class UnlockDoorAction.

Class LockedDoor

public class **LockedDoor** extends Ground

The LockedDoor class inherits from Ground and instantiates a new locked door.

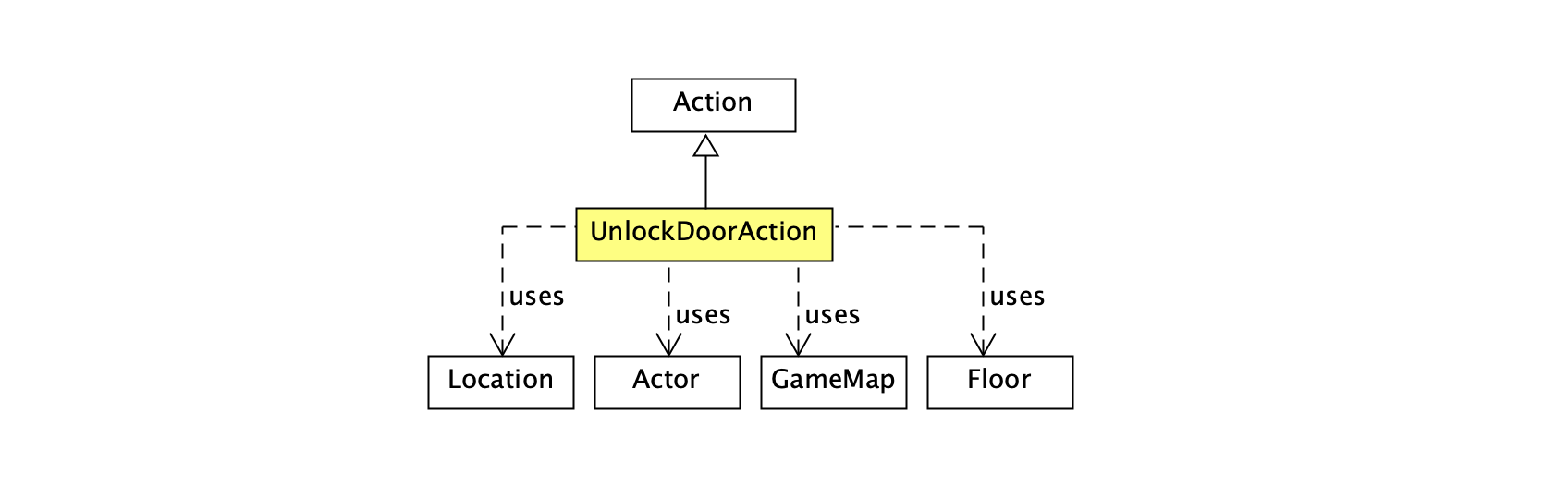
**Design Rationale:**

The LockedDoor class inherits from Ground because it can inherit the existing methods in the Ground class and override the methods that have a different behaviour from its superclass. Since inheriting methods reduces duplicated code, this uses the DRY principle.

Class UnlockDoorAction

public class **UnlockDoorAction** extends Action

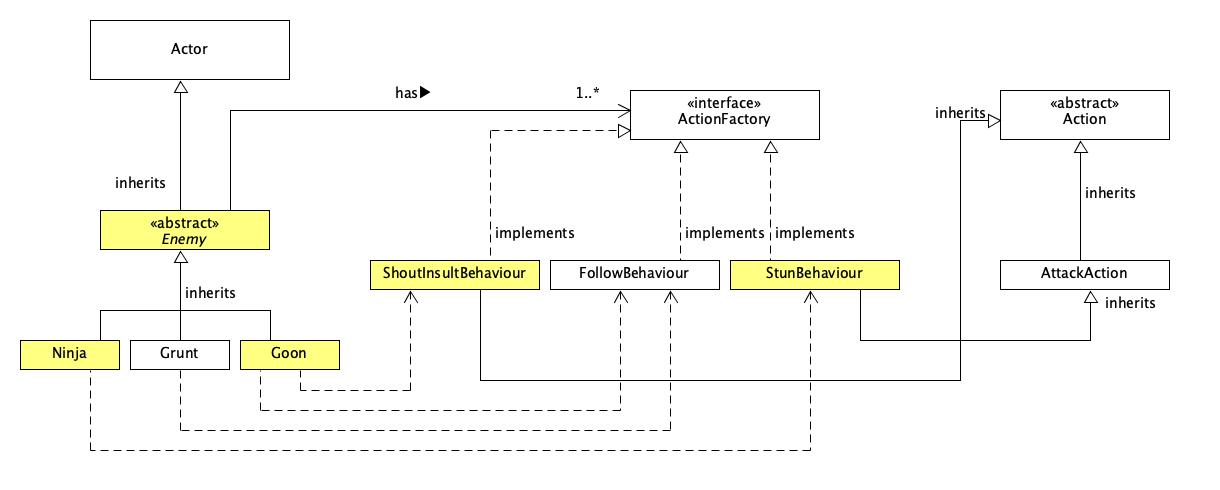
The UnlockDoorAction class is responsible for the actor performing the action to unlock a door.



Based on the **class diagram** above, the UnlockDoorAction inherits from Action. It uses the Location class to get the location of the locked door so that after the door has been unlocked, it uses the Floor class to replace the locked door with a floor display. It uses the Actor class in the overridden menuDescription method to return a String of the actor performing the unlock door action.

**Design rationale:**

The UnlockDoorAction class has been added because it represents an action that the player can do. It is responsible for its own properties whereby its sole responsibility is to represent the action of unlocking a door.

**New types of enemies**

Based on the **class diagram** above, the abstract Enemy class inherits from the Actor class. The Ninja, Grunt and Goon class inherit from the Enemy class. The Enemy class has a List attribute of type ActionFactory that stores the behaviours of the enemies. Grunt class adds its behaviour FollowBehaviour through its superclass’s addBehaviour method. Goon class adds its behaviour FollowBehaviour and ShoutInsultBehaviour through its superclass’s addBehaviour method. Ninja class adds its behaviour StunBehaviour through its superclass’s addBehaviour method. The classes ShoutInsultBehaviour, FollowBehaviour, StunBehaviour implements ActionFactory as these classes represent behaviours of the actors that use it. The ShoutInsultBehaviour class inherits from Action and the StunBehaviour class inherits from AttackAction.

Class Enemy

public abstract class **Enemy** extends Actor

The Enemy class inherits from Actor and serves as a template for subclasses that inherit from it because all subclasses share a common set of instance variables and methods.

How it works:

The methods in this class include a method createKey to create an Item key, a method addBehaviour which adds the behaviour(s) of the enemy to a List of type ActionFactory, a getter method to return the List of behaviours and an overridden playTurn method that returns an action to perform based on the enemy’s behaviour(s). The addItemToInventory method is called in the enemy constructor to add the Item object, key as every enemy has a key item which will be dropped when knocked out.

**Design rationale:**  
Firstly, the Enemy class has been declared abstract because this class serves as a template for subclasses that inherit from it. Furthermore, the Enemy class has been declared abstract as it should not be instantiated to create an Enemy object since it does not represent any specific type of enemy in the game.

Next, the Enemy class contains a method addBehaviour which lets the enemies to have behaviours. Since all subclasses of Enemy has one or more behaviours, the behaviours can be added by calling their superclass’s addBehaviour method. The Enemy class also instantiates a new Item object, key whenever a subclass of Enemy is instantiated. This reduces code and implements the DRY concept. This is because each subclass does not have duplicated code to add behaviour to their classes and to create a key.

In order to create an Item key, a method createKey has been added. This follows the design principle to declare things in the tightest possible scope as the Item object is local and only instantiated in a method when the method is called.

Class Grunt

public class **Grunt** extends Enemy

The Grunt class inherits from Enemy and instantiates a new Grunt object. The class has been created to represent a Grunt type of enemy in the game.

**Design rationale:**  
The class has been refactored to reduce duplicated code and applies the DRY concept. It calls its superclass’s addBehaviour method to add FollowBehaviour so that it follows the player. The playTurn method has been removed as it inherits the playTurn method from its superclass. The Grunt class has an overridden method, getIntrinsicWeapon so that Grunt can slap the player.

Class Goon

public class **Goon** extends Enemy

The Goon class inherits from Enemy and instantiates a new Goon object. The class has been created to represent a Goon type of enemy in the game.

How it works:

It calls its superclass’s addBehaviour method to add FollowBehaviour and ShoutInsultBehaviour. The Goon class has an overridden playTurn method so that there is a 10% chance to return the action which shouts insults. The Goon class gets the List of behaviours by calling its superclass’s getter method.

**Design rationale:**  
The Goon class calls it superclass’s methods to reduce code duplication and this uses the DRY principle. Since the Goon class cannot access the instance variable of its superclass, a getter is used to access its List of behaviours. This promotes encapsulation and data hiding to prevent illegal data access.

Class ShoutInsultBehaviour

public class **ShoutInsultBehaviour** extends Action implements ActionFactory

The ShoutInsultBehaviour class inherits from Action and implements ActionFactory. This class is responsible for the shout insult behaviour.

How it works:

The class has an ArrayList of strings which are insults. It has an overridden execute method which returns a randomly selected insult. The class also has an overridden getAction method which returns the current ShoutInsultBehaviour object instance.

**Design rationale:**

The ShoutInsultBehaviour inherits the Action class because shouting an insult is an action performed by an actor and implements the Interface ActionFactory as it is a behaviour called by the actor in every turn.

Class Ninja

public class **Ninja** extends Enemy

The Ninja class inherits from Enemy and instantiates a new Ninja object. The class has been created to represent a Ninja type of enemy in the game.

How it works:

It calls its superclass’s addBehaviour method to add StunBehaviour which stuns the player.

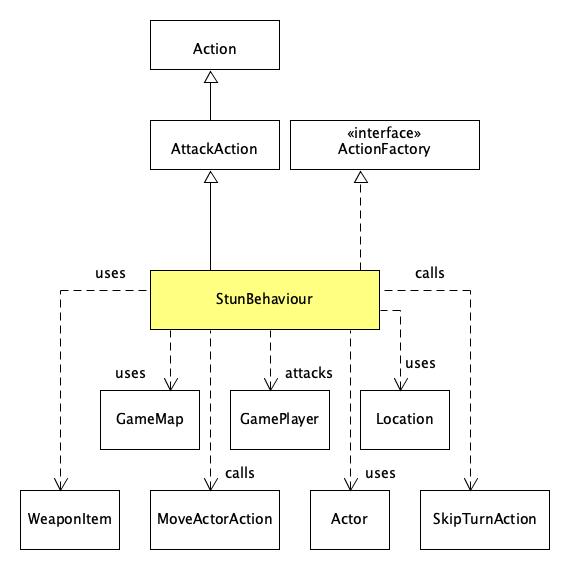
**Design rationale:**

By using the inherited methods, this reduces duplicated code and uses the DRY principle.

Class StunBehaviour

public class **StunBehaviour** extends AttackAction implements ActionFactory

The StunBehaviour class is responsible for implementing the stun attack behaviour whereby the actor that has this behaviour stays in one place unless the player is within 5 squares away from them. Then, they will attack by throwing a bag of stun powder with a 50% chance of hitting and move away. If the player is already stunned, the bag of stun powder has no effect. The class has an overridden execute method and an overridden getAction method.



Based on the **class diagram** above, the StunBehaviour class inherits from AttackAction and implements ActionFactory. It uses the Location class to get the current location of the Actor object and the GamePlayer object to measure the distance between them. If the distance is less than or equals to 5 squares apart, the StunBehaviour class uses WeaponItem to instantiate a new WeaponItem object, stunPowderBag in the execute method to attack the GamePlayer object. Then, it calls the MoveActorAction class to move away. If the distance is more than 5 squares apart, the StunBehaviour class calls the SkipTurnAction class to stay in one place and do nothing.

How it works:

Overridden execute method

The method checks if the subject is an instance of GamePlayer. If it is not, it calls its superclass’s execute method as stun attack should only be executed on a GamePlayer object. If it is an instance of GamePlayer, it checks if the GamePlayer object is stunned by calling its getPlayerStunned method.

* If the GamePlayer object is stunned, it returns the String that the attack is missed.
* If the GamePlayer object is not stunned, there is a 50% chance of the bag of stun powder hitting.
  + If the attack is not successful, it returns the String that the attack is missed.
  + If the attack is successful, the GamePlayer’s setPlayerStunned method is called. The WeaponItem’s damage method is called to get its object, stunPowderBag’s damage. Then, the GamePlayer’s hurt method is called to hurt the player with the damage done by the stunPowderBag. Lastly, it returns the String that the player has been hit by a stun powder bag with the damage inflicted.

Overridden getAction method

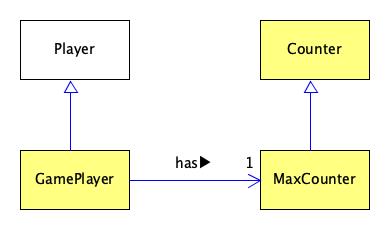
* If the current distance between the Ninja object and GamePlayer object is less than or equals to 5 squares apart and the new distance is more than the current distance, the method will call and print the execute method onto the console and returns new MoveActorAction so that the Ninja object will move away from the GamePlayer object. This means that the Ninja will both stun the GamePlayer object and move away in one turn.
* If the current distance between the Ninja object and GamePlayer object is more than 5 squares apart, the method returns new SkipTurnAction so that the Ninja object stays in one place and does nothing.

**Design rationale:**  
The StunBehaviour class extends Action because the actor that uses this class will perform stun attack which is an action. The StunBehaviour class also implements ActionFactory as this class represents a behaviour that can be executed by the actor in each turn. The behaviour of the actor that uses this class is such that the actor remains in one place unless the player is 5 squares away from them. Then, they will throw a bag of stun powder (this is an action). Hence, the StunBehaviour class uses both extends and implements. Furthermore, by using both extends and implements, the StunBehaviour class can execute the stun attack while moving away from the player in one turn.

Class GamePlayer

public class **GamePlayer** extends Player

The GamePlayer class inherits from Player and instantiates a new GamePlayer object. This class represents a player in the game.



Based on the **class diagram** above, the GamePlayer class inherits from the Player class. The GamePlayer has a MaxCounter attribute. The MaxCounter class inherits from the Counter class.

Reasons why a new GamePlayer class is added:

* A MaxCounter attribute has been added to the GamePlayer class.
* A boolean attribute playerStunnedStatus has been added to the GamePlayer class to keep track if the GamePlayer object is stunned. There is a public setter and getter so that other classes can get and set the current GamePlayer object’s stunned status.

**Design rationale:**

Firstly, the DRY principle is used because MaxCounter is the subclass of Counter and inherits its methods. This allows the subclass (MaxCounter) to reuse all the methods in the Counter class.

Furthermore, by using the MaxCounter attribute, the GamePlayer object does not have to add additional methods to increment and reset the counter. The MaxCounter attribute knows its own maximum value and resets itself. Thus, the design principle that classes should be responsible of their own properties is applied.

The getter and setter in the GamePlayer class is used to achieve encapsulation. This ensures that other classes are not permitted to access the GamePlayer class’s instance variables while still being able to access and set its values.

Class Counter

public class **Counter**

The Counter class is responsible for operations that are performed on a Counter object, for instance; reset, increment, decrement, and getValue.

Class MaxCounter

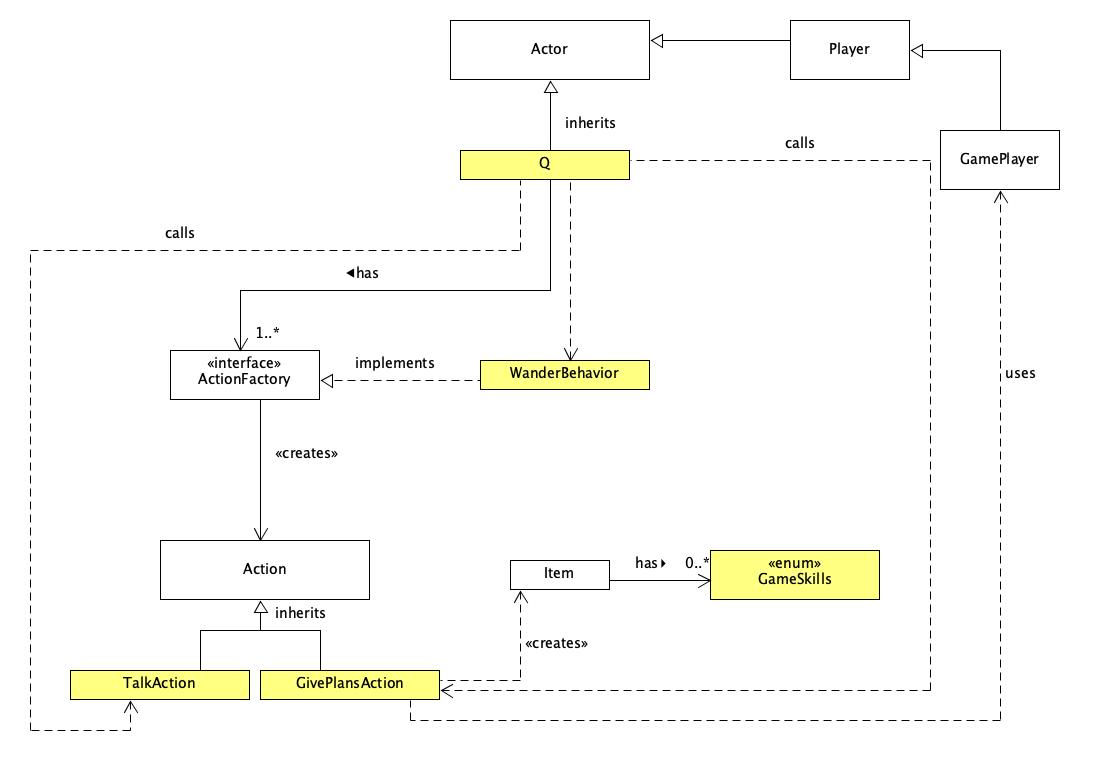
public class **MaxCounter** extends Counter

The MaxCounter class inherits from Counter. It resets itself as it knows its own maximum value.

**Design rationale:**

By inheriting from Counter, the DRY design principle is applied, because the inherited method can reduce duplicated codes. MaxCounter has an instance variable called max. This new variable max will store the maximum value of the counter, as a result, uses of magic number / string literal is avoided. This improves the maintainability and also reduces the risk of introducing unwanted bugs in the future.

**Q**



Based on the **class diagram** above, the Q class inherits from Actor. It has a List of type ActionFactory and calls the WanderBehaviour class through the addBehaviour method. This allows Q to wander around the map. Q has an overridden method getAllowableActions which calls the TalkAction class to enable Q to talk and calls the GivePlansAction class to enable the player to give the Item rocket plans. The GivePlansAction classes uses the GamePlayer class as it will remove the rocket plans item and add a rocket body item into its inventory. If the player successfully gives their rocket plans, Q will disappear from the GameMap with a cheery wave.

Class Q

public class **Q** extends Actor

The Q class inherits from Actor and instantiates a new Q object. The Q class is responsible for creating a non-player character in the game which supplies the player with a part of the rocket and support the actions for it to give plans and talk.

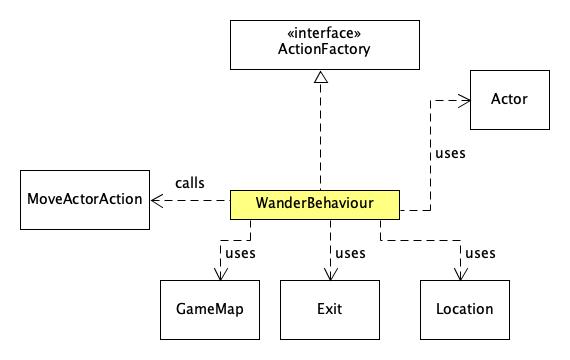
**Design rationale:**

The Q class inherits from the Actor class because it can inherit the existing methods in the Actor class and override the methods that have a different behaviour from its superclass. Since inheriting methods reduces duplicated code, this uses the DRY principle.

Class WanderBehaviour

public class **WanderBehaviour** implements ActionFactory

The WanderBehaviour class implements ActionFactory and creates a new Action in which the actor wanders around the map.



How it works:

Based on the **class diagram** above, the WanderBehaviour class implements ActionFactory. The WanderBehaviour class uses GameMap and Location to get the current location of the Actor. Then, it uses the Exit class to get a possible route from one location to another. Since it is random, there is no final destination. Then, the WanderBehaviour class calls the MoveActorAction class to move the actor.

**Design rationale:**

The Interface ActionFactory has a method which is used by classes that implement it to give an Actor in the game a behaviour. Since Q wandering around the map is a behaviour, the WanderBehaviour class implements the Interface ActionFactory. Furthermore, WanderBehaviour implements ActionFactory for abstraction as WanderBehaviour has to implement the method getAction based on the requirements of its own class. WanderBehaviour implementing the Interface ActionFactory also introduces loose coupling between the classes that implement ActionFactory which reduces dependencies and uses the ReD concept.

Class GivePlansAction

public class **GivePlansAction** extends Action

The GivePlansAction class inherits from Action. The GivePlansAction class allows the actor to give rocket plans and receive a rocket body.

How it works:

The GivePlansAction class contains a method which allows the player to give Item rocket plans which has the skill, GameSkills.GETROCKETBODY and subsequently have the rocket plans Item removed from the player’s inventory. Then, it calls another method which creates an Item, rocketBody has skill, GameSkills.BUILDROCKETTOP and is added to the player’s inventory.

**Design rationale:**

The GivePlansAction inherits the Action class because giving rocket plans represents an action that can be performed by the player. Furthermore, it can override its superclass’s methods to provide the specific implementation required.

Class TalkAction

public class **TalkAction** extends Action

The TalkAction class inherits from Action. The TalkAction class overrides the menuDescription method to return the correct String based on whether the player has an Item with the skill GameSkills.GETROCKETBODY.

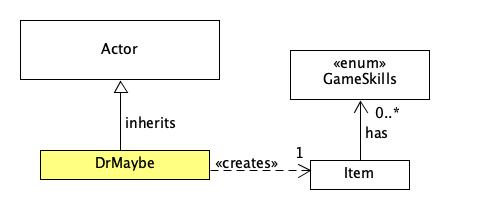
How it works:

* If the player does not have an Item with the skill GameSkills.GETROCKETBODY, the String “I can give you something that will help, but I’m going to need the plans” will be returned
* If the player has an Item with the skill GameSkills.GETROCKETBODY, the String “Hand them over, I don’t have all day!” will be returned

**Design rationale:**

The TalkAction class inherits the Action class because talking represents an action that can be performed by Q. Furthermore, it can override its superclass’s methods to provide the specific implementation required.

**Miniboss: Doctor Maybe**

****

Based on the **class diagram** above, DrMaybe inherits from Actor. The DrMaybe class has a method that creates an Item, rocket engine. The rocket engine has GameSkills.BUILDROCKETBASE.

Class DrMaybe

public class **DrMaybe** extends Actor

The DrMaybe class inherits from Actor and instantiates a new DrMaybe object.

How it works:

DrMaybe contains a method which creates an Item rocket engine and a method addItemToInventory to add the rocket engine into the inventory.

**Design rationale:**

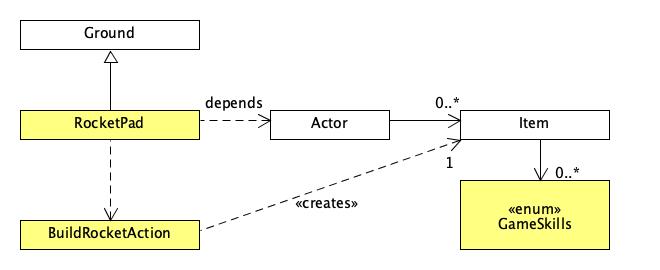
The DrMaybe class inherits from Actor because it can override the abstract methods to include its own implementation of the methods. Since inheriting methods reduces duplicated code, this uses the DRY principle.

**Building a rocket**

How it works:

The rocket can only be built on the Rocket Pad (which is a subclass of Ground). In order to build the rocket, Player must possess two Item objects, namely; the rocketEngine and rocketBody. Both Item objects will be assigned to a specific skill in the enum, GameSkills. The Item object, rocketEngine dropped by Dr Maybe has skill GameSkills.BUILDROCKETBASE. Meanwhile, Q gives an Item object, rocketBody in exchange for an Item object, rocketPlans. The rocketBody item has the skill GameSkills.BUILDROCKETTOP.

During the game play, the method in RocketPad will check if the player (which is a subclass of Actor) have two items which have the required GameSkills in their inventory. If both rocketEngine and rocketBody items are found, an Item object, rocket will be created.

****

Based on the **class diagram** above, RocketPad inherits from Ground. The RocketPad class depends on the Actor object’s inventory that consists of Item objects. If the Item objects have GameSkills.BUILDROCKETBASE and GameSkills.BUILDROCKETTOP, the RocketPad class calls the BuildRocketAction class. The BuildRocketAction class has a method which creates a new Item object called rocket.

Class RocketPad

public class **RocketPad** extends Ground

The RocketPad class inherits from Ground and instantiates a new rocket pad.

How it works:

The class overrides the allowableActions method which checks for each item in the player’s inventory if it has the skills GameSkills.BUILDROCKETBASE and GameSkills.BUILDROCKETTOP. If yes, it calls the BuildRocketAction class.

**Design rationale:**

The RocketPad class inherits from the Ground class because it can inherit the existing methods in the Ground class and override the methods that have a different behaviour from its superclass. Since inheriting methods reduces duplicated code, this uses the DRY principle.

Class BuildRocketAction

public class **BuildRocketAction** extends Action

The BuildRocketAction class inherits from Action and instantiates a new rocket pad. The class has a method which builds an Item object, rocket that is added into the player’s inventory.

**Design rationale:**

Since the Item class contains appropriate methods and Skills as one of its instance variable, rocketBody and rocketEngine are instantiated as an Item object. By doing so, we are able to use the existing methods in the Item class. This reduces code and implements the DRY design principle. Furthermore, this allows easy maintenance and decreases the chances of introducing new bugs into the code.

**Others**

Enum GameSkills

public enum **GameSkills**

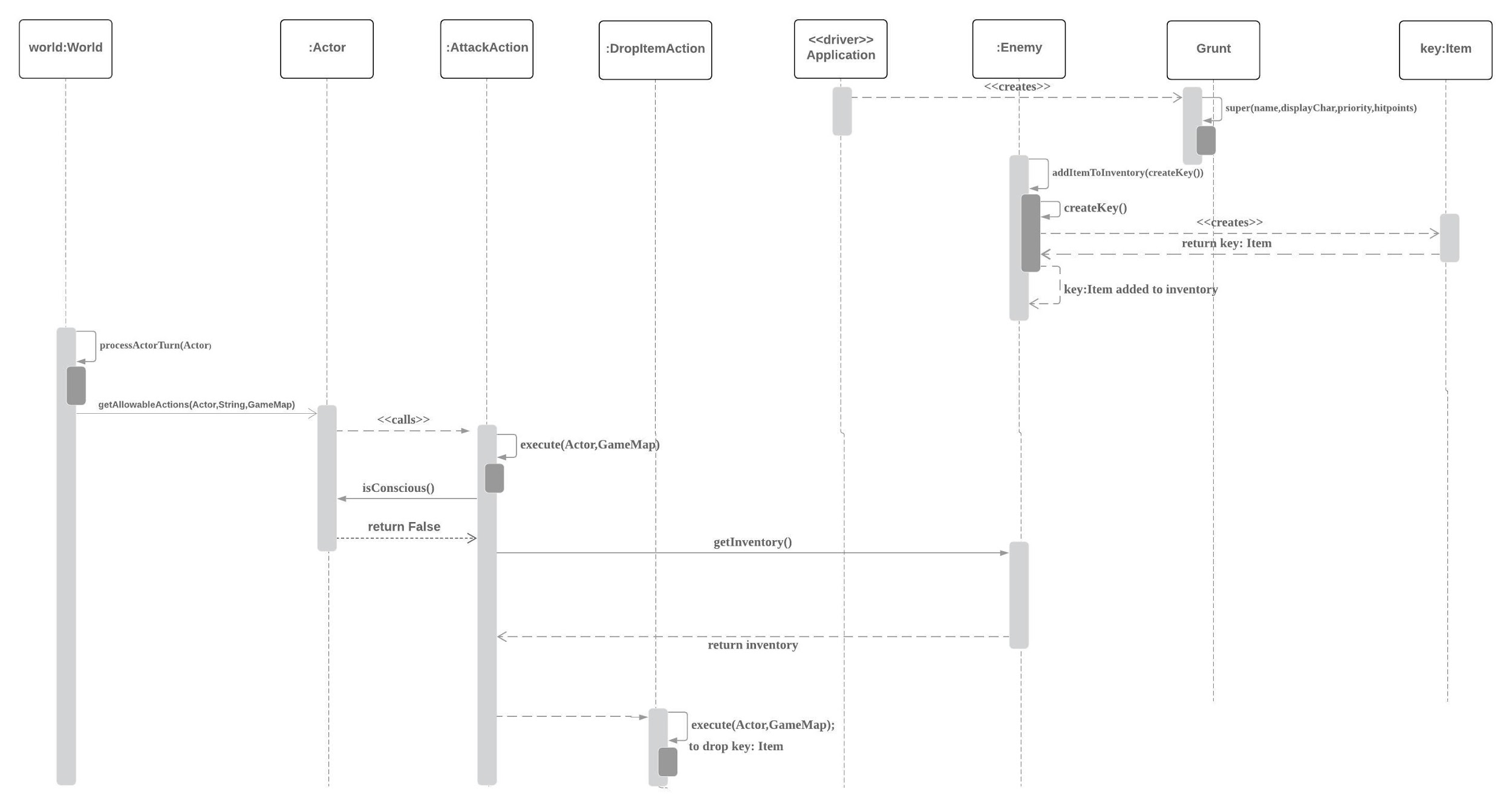
GameSkills is an enum consisting of skills that can be added to Item objects. The enum of skills include UNLOCKDOOR, BUILDROCKETBASE, BUILDROCKETTOP, GETROCKETBODY.

Class Application

public class **Application**

The Application class is the driver class and instantiates objects for the game. The Application class creates 1 GameMap object, namely, startMap which contains the map that the player starts on. The Application class has walls with a locked door (LockedDoor object) which creates a room that contains a DrMaybe object and another room with rocket plans. The Application class also creates a RocketPad object. The Application class instantiates one GamePlayer object, Grunt objects, Ninja objects, Goon objects, one Q object, one DrMaybe object, one Item rocketPlans that has the skills GETROCKETBODY in a locked room.

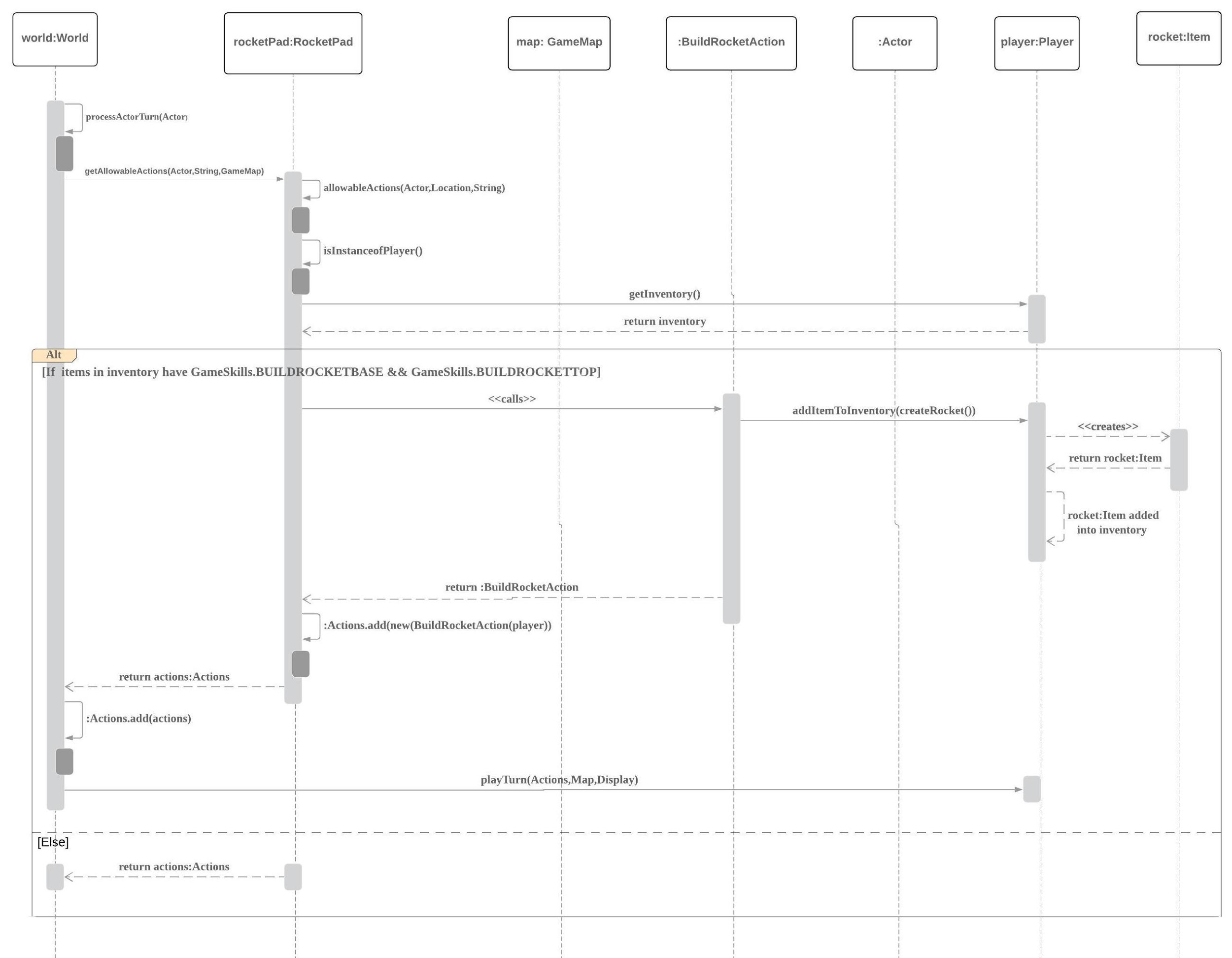
**Interaction diagram (for dropping a key item):**



The sequence diagram above illustrates how a Grunt object drops an Item object, key. This diagram is also applicable for other enemies including Goon and Ninja.

Firstly, in the Application class which is the driver, it will instantiate a new Grunt object. The Grunt class will call its superclass (Enemy)’s constructor. In the Enemy class it will add the Item object, key into its inventory by calling the method addItemToInventory(createKey()). The createKey() returns an Item object named key and it is subsequently added into the Grunt object’s inventory. In the World class, the processActorTurn(Actor) method is called which calls the getAllowableActions(Actor,String,GameMap) method in the Actor class. It will then call the AttackAction class. This will call the Actor’s isConcious method to check whether the current Grunt object is conscious. If the Grunt object is not conscious, this implies that the Grunt object is knocked out. The Enemy class (which is the subclass of Actor) will then return its inventory which contains the key item. The key item will subsequently be dropped.

**Interaction diagram (for building a rocket item):**

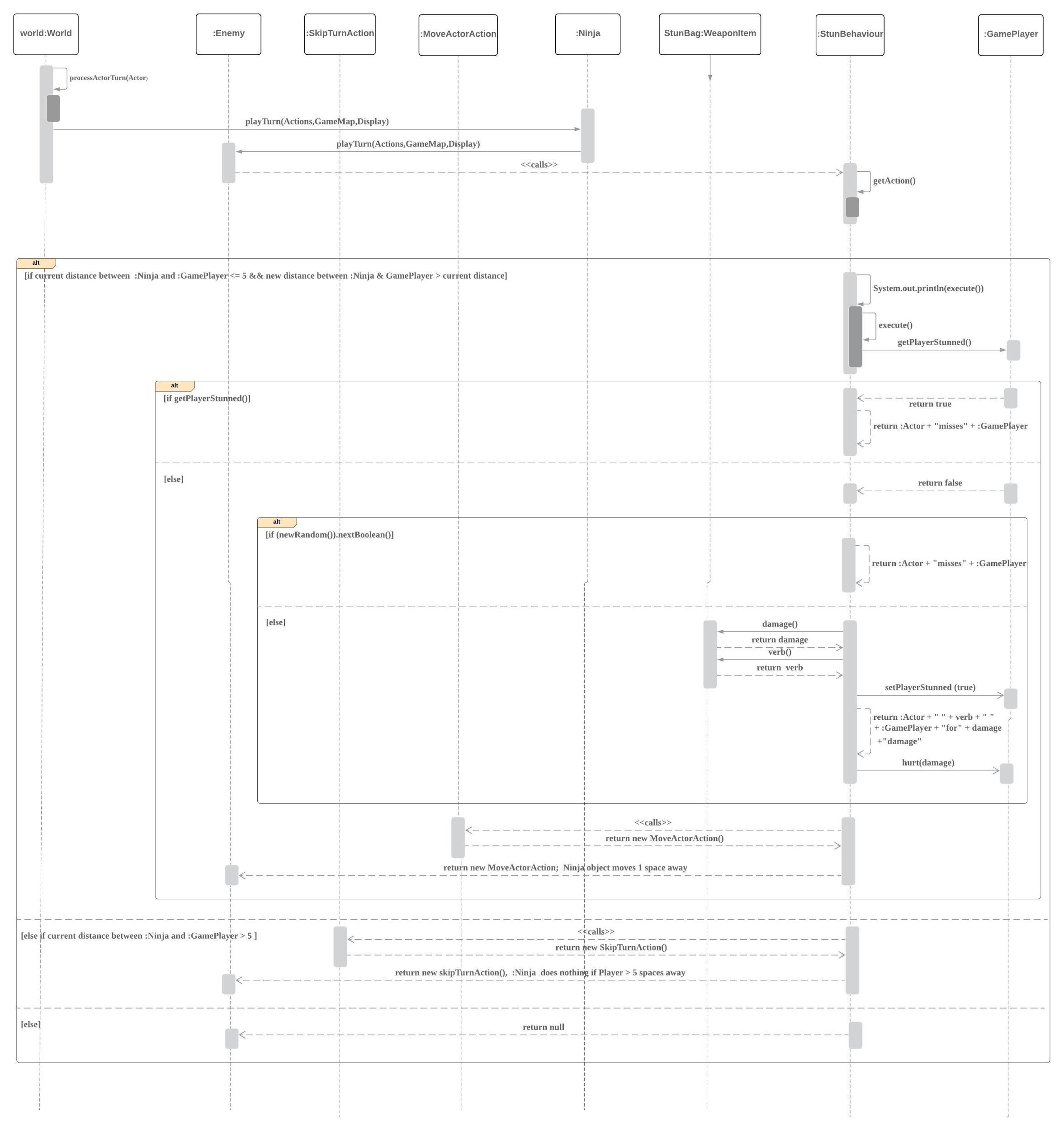


The interaction diagram above illustrates how an Item object, rocket is built. It will first check if the current Actor is an instance of Player. Since the GamePlayer class is a subclass of Player, it is an instance of Player. If the Actor is an instance of Player, it will check if the Player object contains both of the following items:

1. One Item that has the GameSkills.BUILDROCKETBASE (namely Item rocketEngine)
2. One Item that has the GameSkills.BUILDROCKETTOP (namely Item rocketBody)

If the Player’s object’s inventory contains both Items, it calls for the BuildRocketAction where a new rocket Item will be created and subsequently added into the GamePlayer object’s inventory. The newly instantiated BuildRocketAction is returned then added to the list of allowable actions. The game continues with playTurn(Action,Map,Display).

**Interaction diagram (for ninja to stun the player) :**



The interaction diagram above illustrates how the stun attack is performed.

It first checks for the following two conditions:

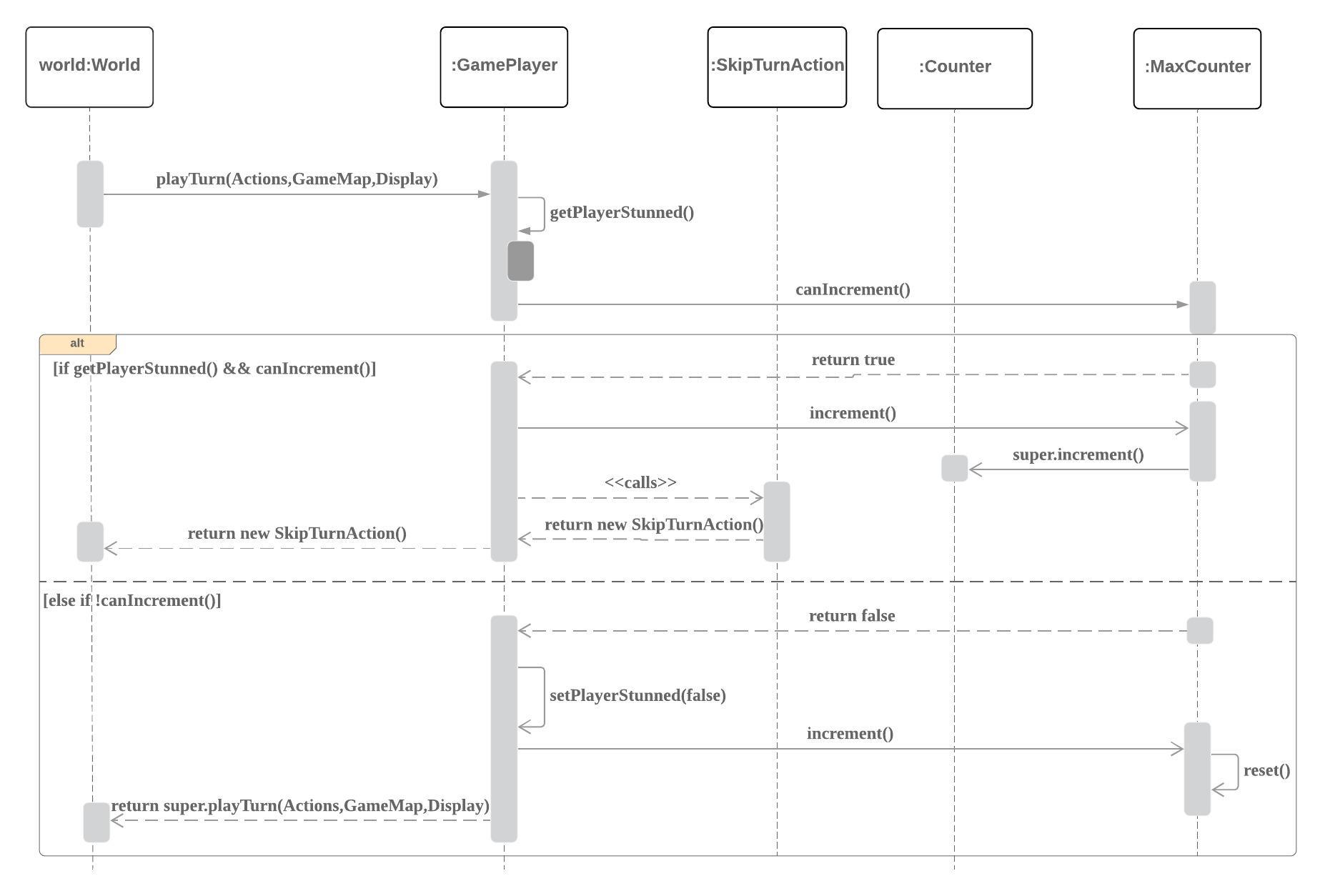
1. The current distance between :Ninja and :GamePlayer is less than or equals to 5

2. The new distance between :Ninja and :GamePlayer is greater than the current distance

If both of the conditions above are true, the program continues by checking if the GamePlayer is stunned. If GamePlayer is already stunned, Ninja misses the attack this round. On the contrary, if GamePlayer is not stunned, Ninja will attempt to stun attack the player at a 50% success rate and calls the MoveActorAction class to move one space away from Player.

In the event where the current distance between the Ninja object and GamePlayer object is more than 5 squares apart, Ninja calls the SkipTurnAction class and does nothing.

**Interaction diagram (for GamePlayer and counter) :**



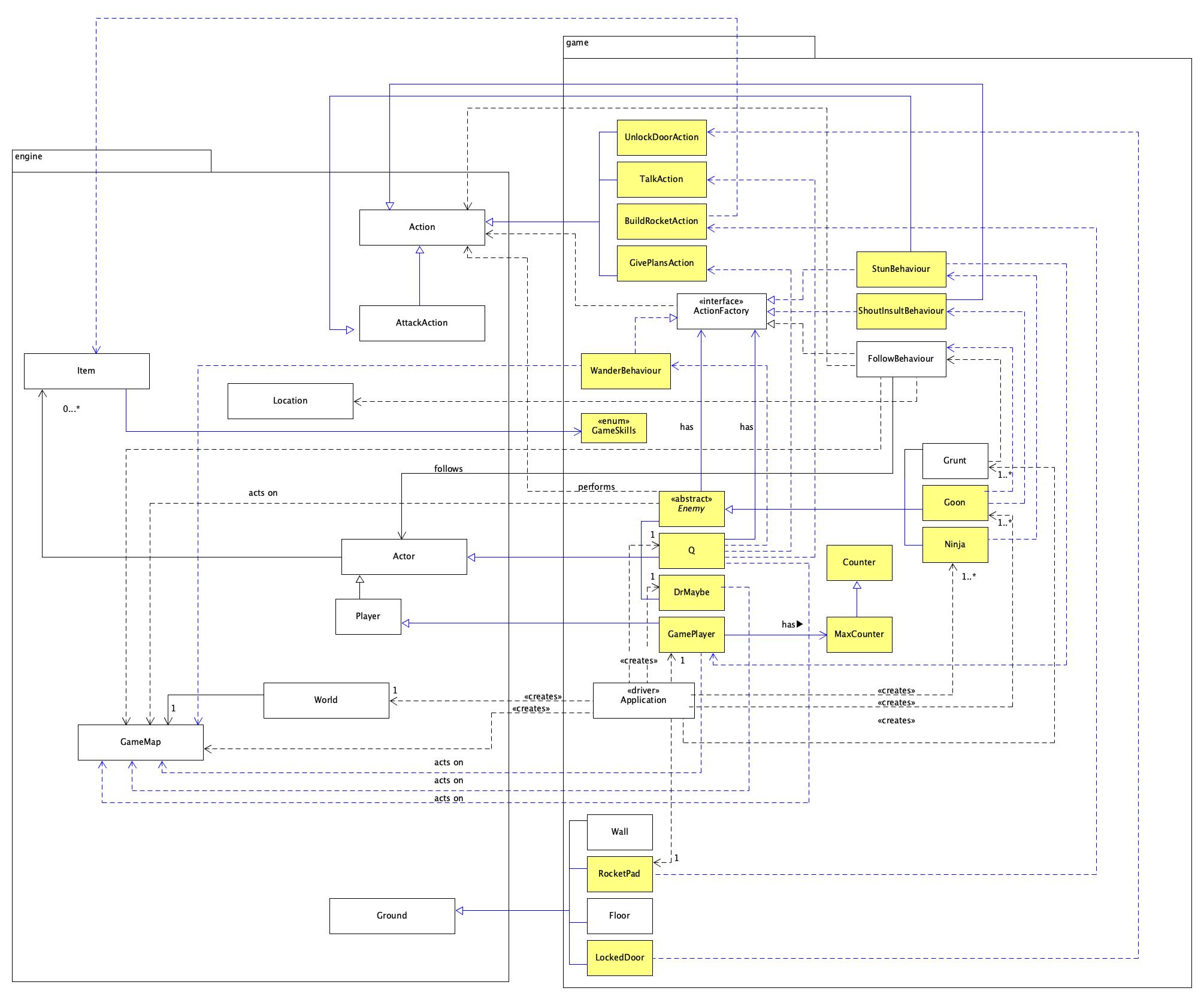
The interaction diagram above illustrates how the GamePlayer class and MaxCounter class work together to keep the player stunned for 2 turns. In the playTurn method in the GamePlayer class, it will check:

1. If the player is stunned (getPlayerStunned method)
2. If the counter can be incremented (canIncrement method)

If both of the conditions above are true, the counter will be incremented and GamePlayer object will skip the next turn by calling the SkipTurnAction class.

If the counter cannot be incremented further, it means that the counter has reached its max value and the GamePlayer has been stunned for 2 turns. Hence, it will set the playerStunned boolean back to false by calling the method setPlayerStunned, and resets the counter. The game continues and the GamePlayer object is not stunned anymore. Hence, it can resume performing other actions than waiting.

**Class diagram on how new classes interact with the existing system:**



*Notes: The yellow blocks indicates new classes.*

*The blue lines indicate new relationships between the classes*

Based on the class diagram above, there are two packages shown. The two packages include the engine package and the game package. The new classes are added into the game package.

The LockedDoor class inherits from the GroundClass and calls the UnlockDoorAction class which inherits from the Action class.

An abstract Enemy class has been added that inherits from the Actor class. The Grunt, Goon and Ninja class inherit from the Enemy class. The Enemy class has an ActionFactory attribute. Grunt uses the FollowBehaviour class, Goon uses the FollowBehaviour and ShoutInsultBehaviour class and Ninja uses the StunBehaviour class.

The ShoutInsultBehaviour class inherits from Action and implements ActionFactory. The StunBehaviour class inherits from the AttackAction class and implements ActionFactory. It also depends on the GamePlayer class. The GamePlayer class inherits from the Actor class and has a MaxCounter attribute which is a subclass of the Counter class.

The Q class inherits from the Actor class and calls the TalkAction class and the GivePlansAction class. Q has an ActionFactory attribute and also uses the WanderBehaviour class. The WanderBehaviour class implements the ActionFactory interface and depends on the GameMap class.

The DrMaybe class inherits from Actor and it acts on GameMap.

The RocketPad class inherits from the Ground class and calls the BuildRocketAction class which inherits from the Action class. The BuildRocketAction class creates a new Item object called rocket. The Item class has enum GameSkills.