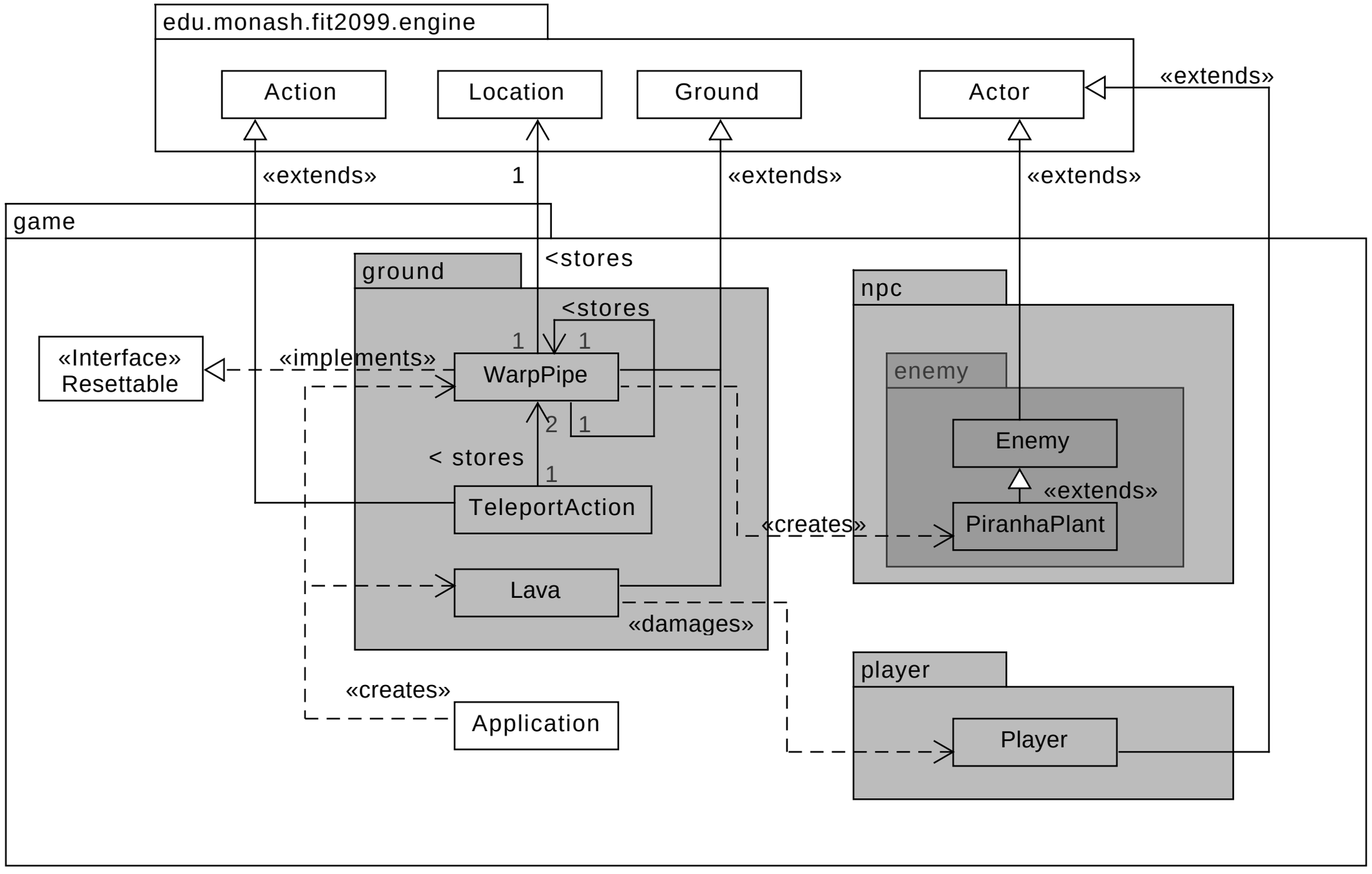
Assignment 3
Requirement 1



Assignment 3

Requirement 2

edu.monash.fit2099.engine

Actor

Ground

Item

Action

Location

«extends»

1

game

npc

Npc

PrincessPeach

«creates»

enemy

Enemy

Goomba

Bowser

EndAction

1 has *

«interface» Behaviour

«implement»

WanderBehaviour

FollowBehaviour

«implements»

PiranhaPlant

FlyingKoopa

Koopa

«has»

AttackBehaviour

«adds»

AttackAction

Application

ground

Fire

WarpPipe

Sprout

Sapling

Mature

item

SuperMushroom

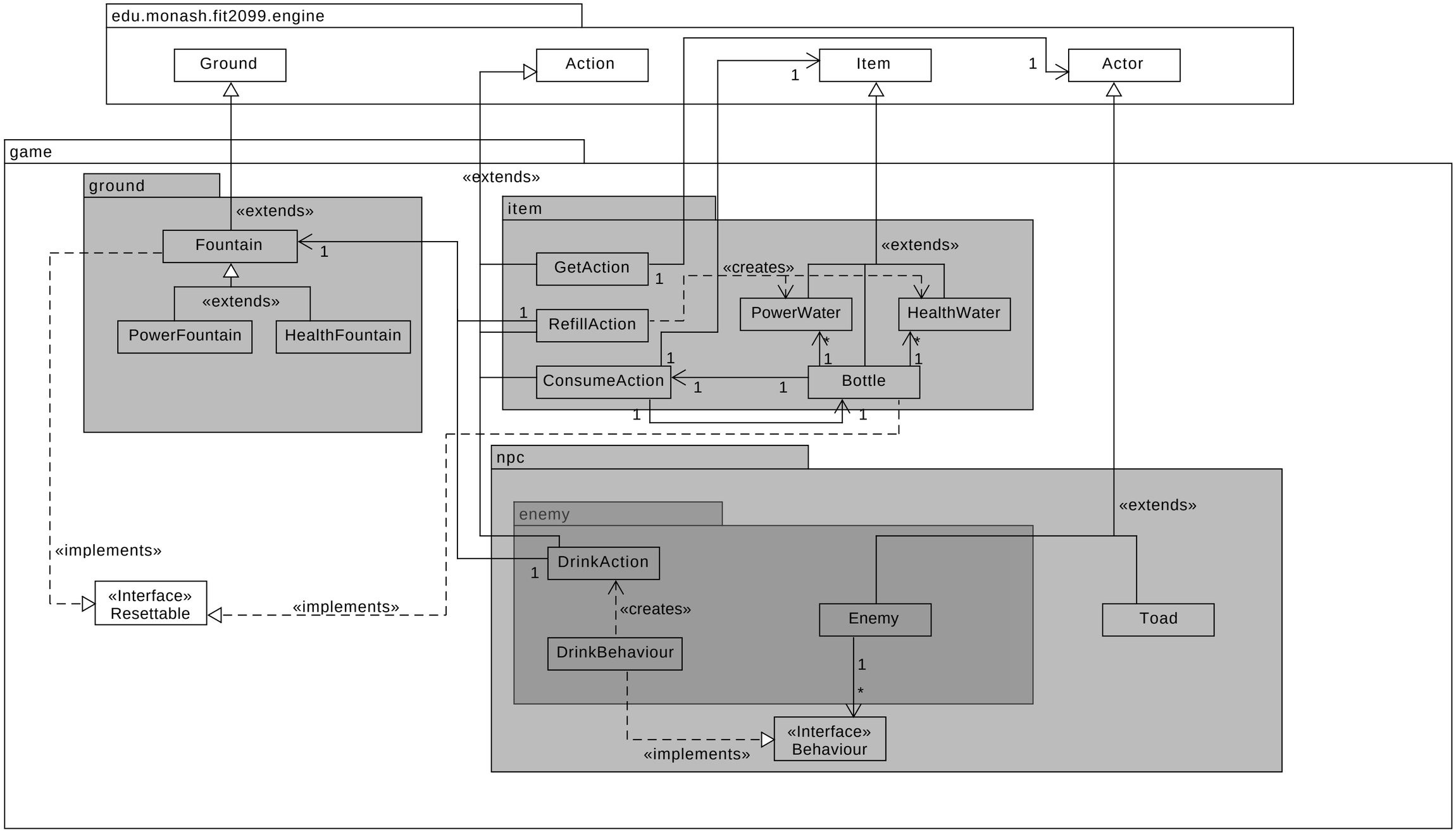
Key

«Uses»

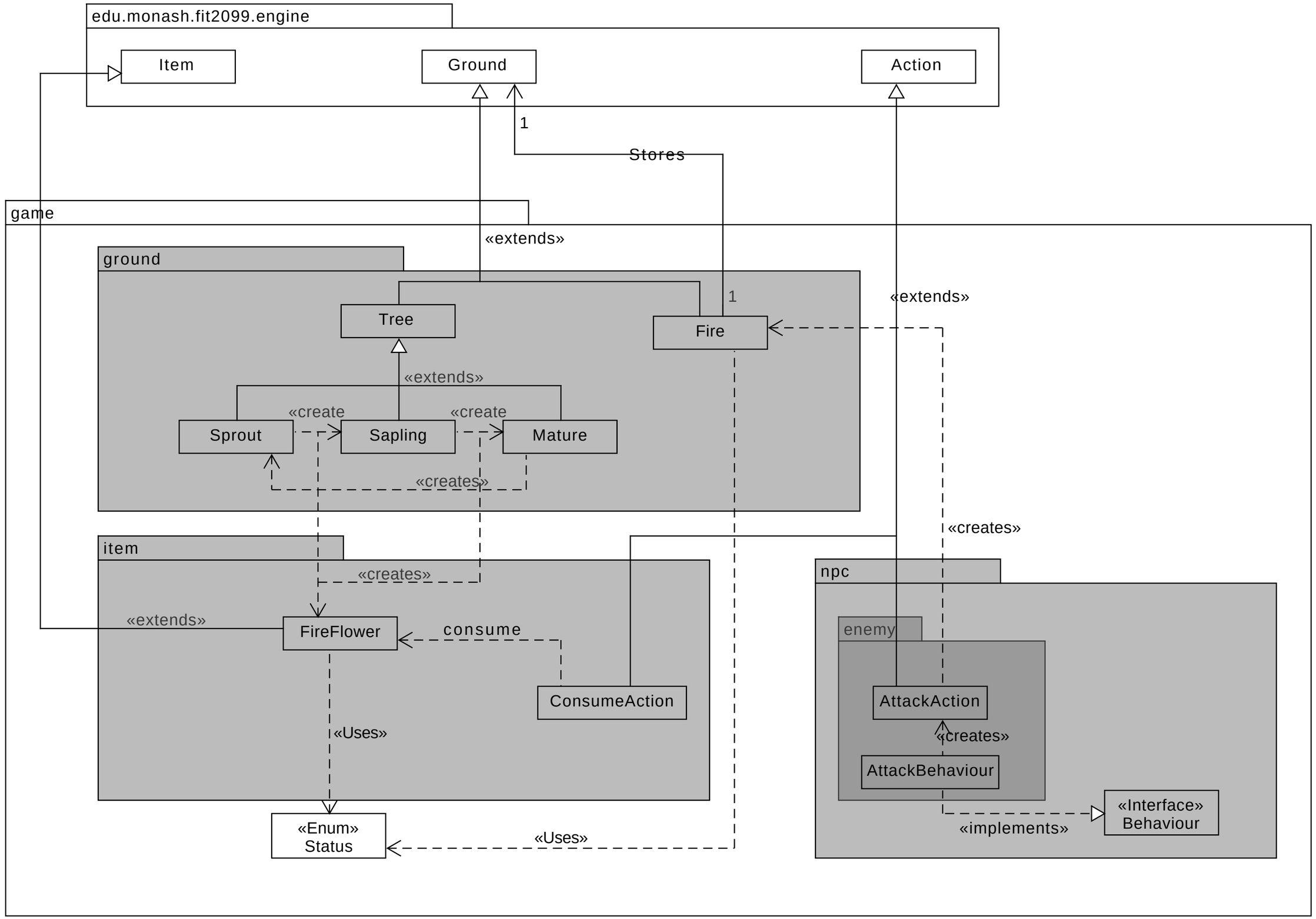
RetrieveAction

«Enum» Status

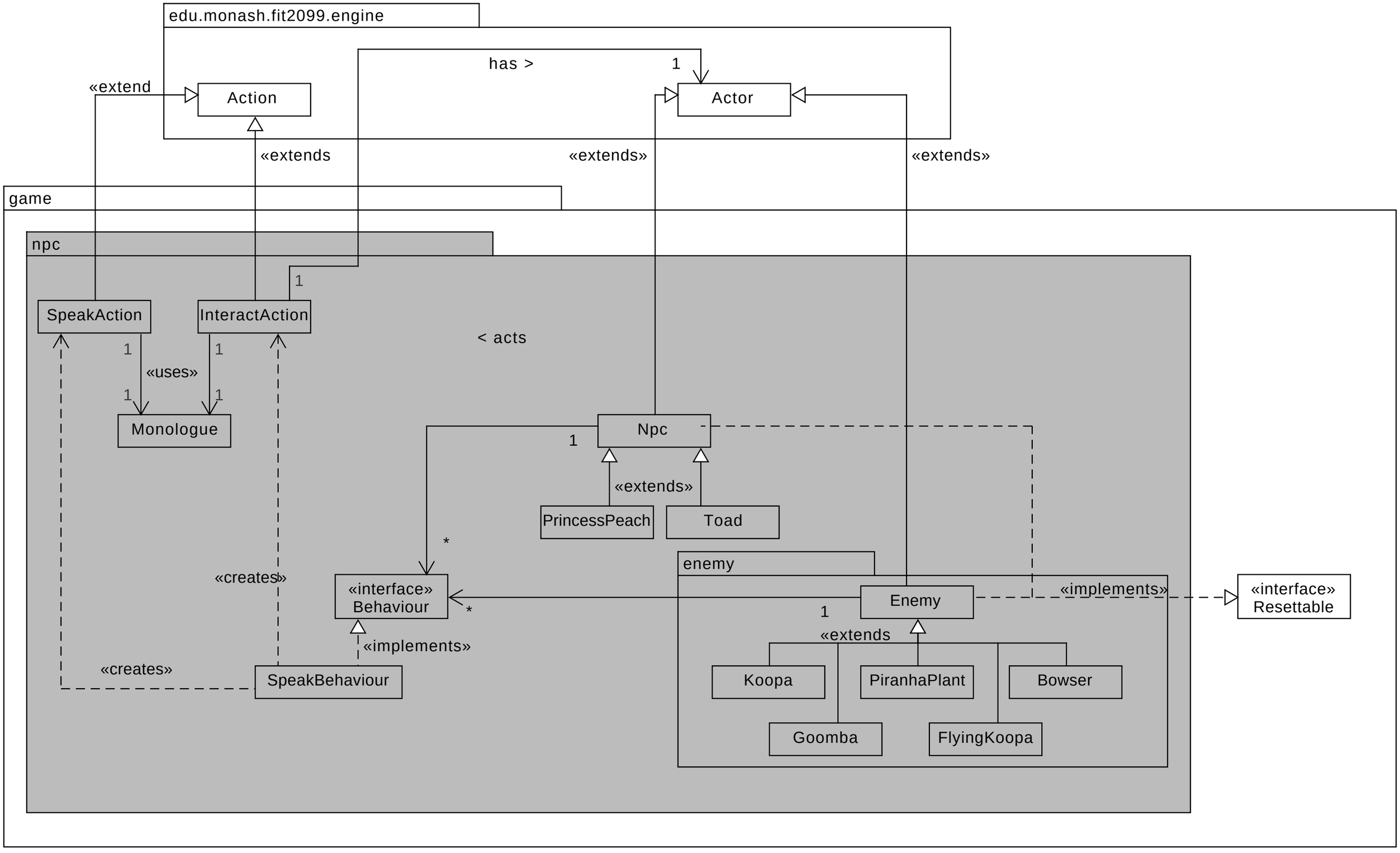
Assignment 3
Requirement 3



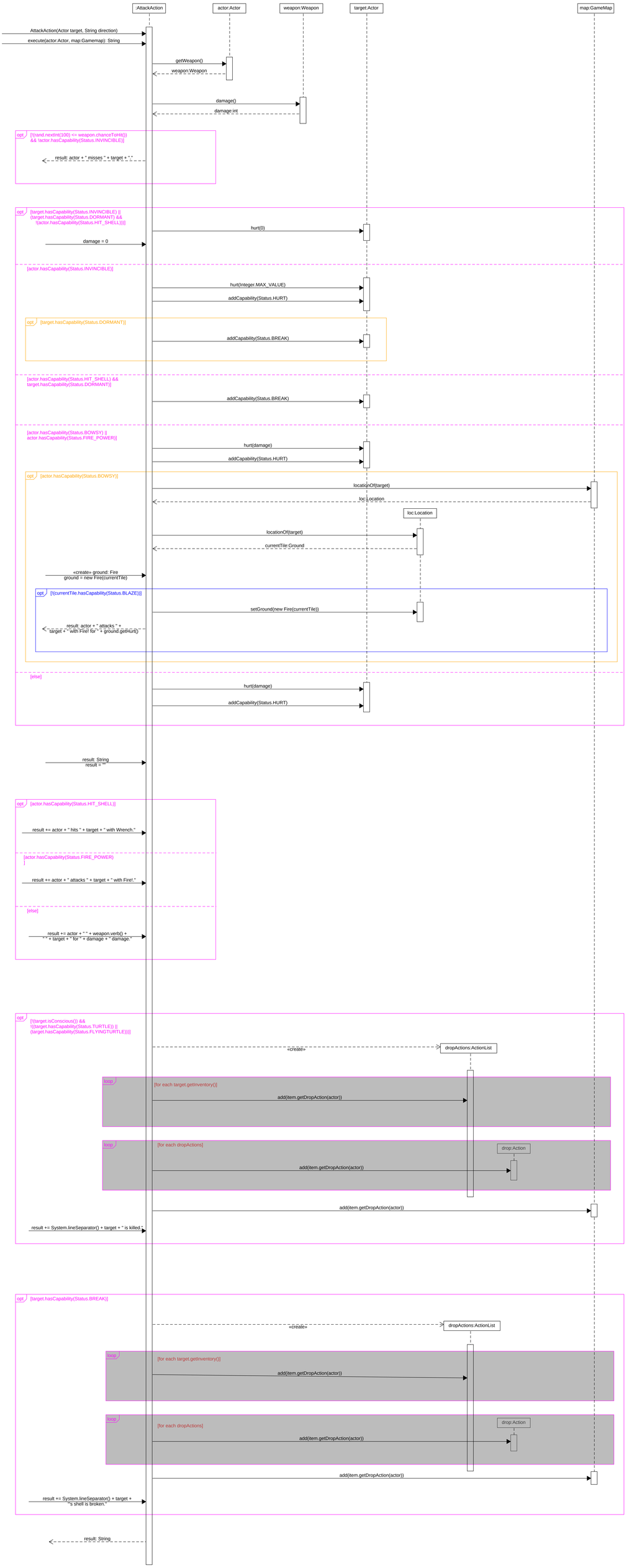
Assignment 3
Requirement 4



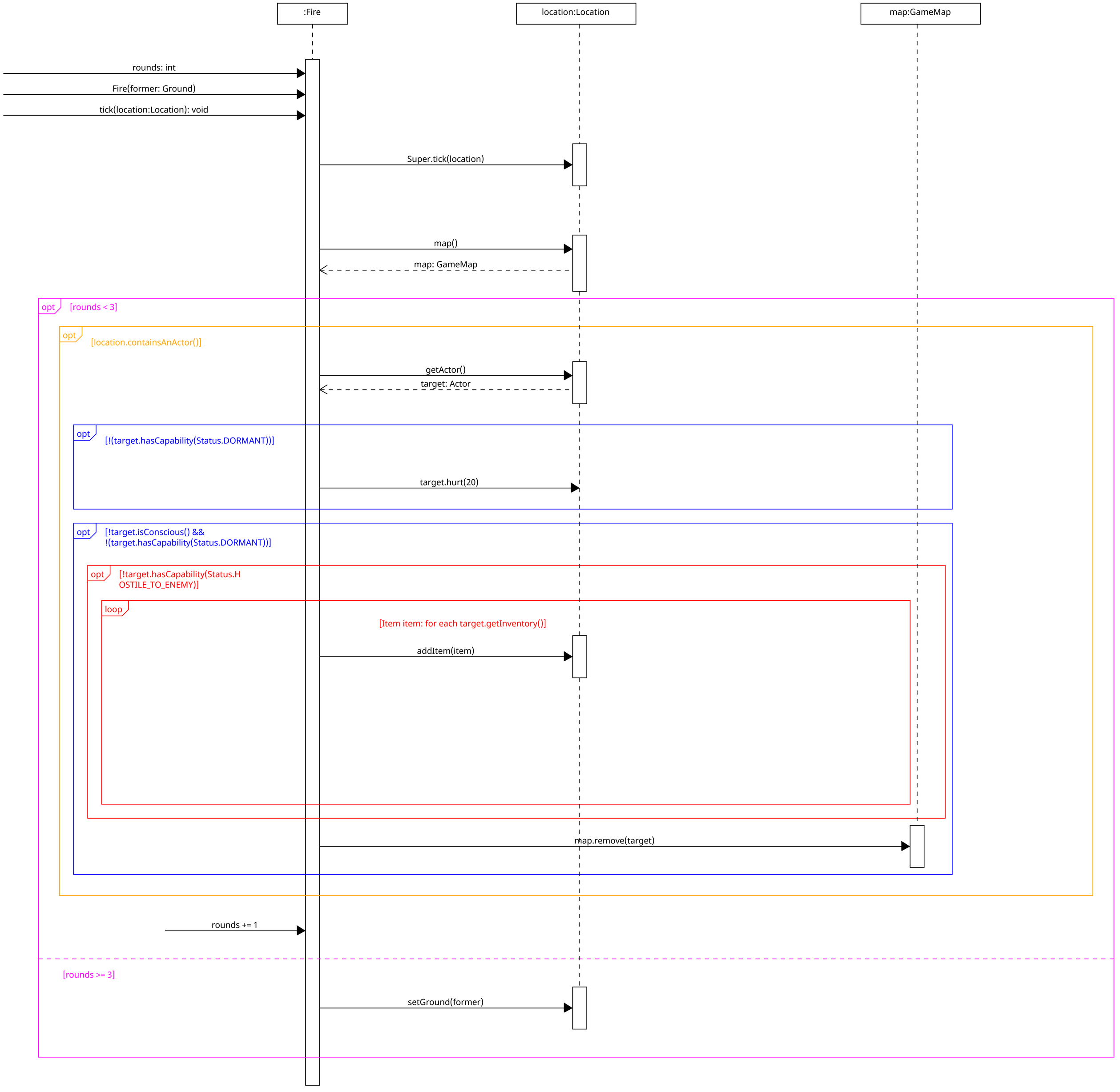
Assignment 3
Requirement 5



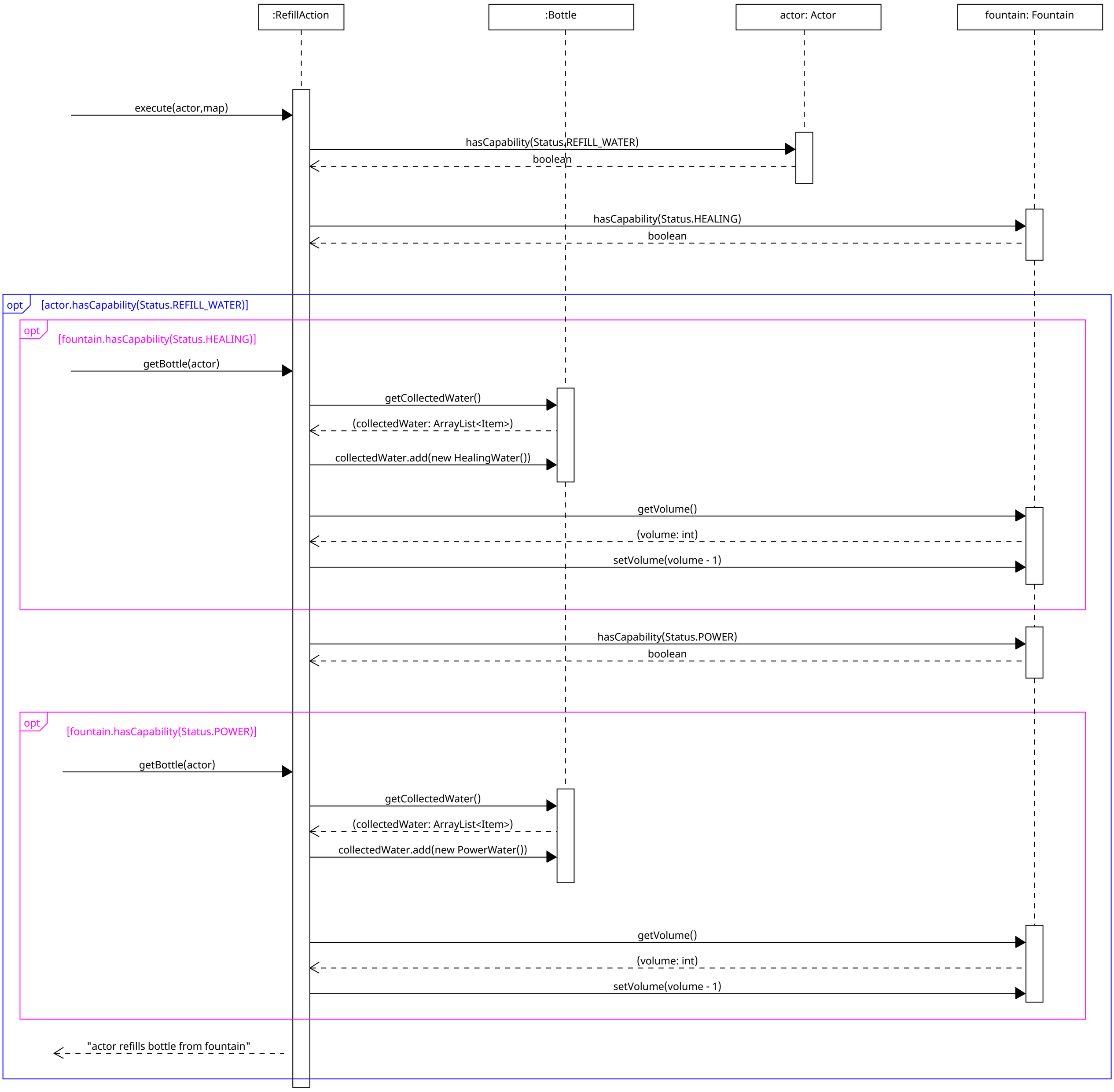
AttackAction.execute() Sequence Diagram



Fire.tick() Sequence Diagram



RefillAction execute() Sequence Diagram



Design Rationale Assignment 3

Sub-package Implementation

This is implemented in a way that avoids multiple crossing over between classes in different subpackages.

ground
This package includes all classes that extend *Ground* and also Action classes that are only applicable to ground objects such as TeleportAction and JumpAction.

item
This package includes all classes that extend *Item* and also Action classes that are only applicable to item objects such as ConsumeAction.

npc > enemy
This package includes multiple classes that extend *Actor*: Non-playable actors such as Toad and Princess Peach. It also includes action and behaviour classes that are only applicable to these actors such as SpeakAction and SpeakBehaviour. It also includes another subpackage **enemy**, which has the classes that represent non-playable characters that are classified as an enemy, such as Bowser and Goomba. Action and behaviour classes that are only applicable for enemies such as AttackAction and AttackBehaviour are under the **enemy** subpackage.

player
This package includes a class that represents a playable character which extends the *Actor* class and also Item or Action classes that are only applicable to this particular actor.

game (main package)
This package contains all of the above subpackages, and also classes that are more of general use, such as interfaces and enumerators, and the main driver for the entire project.

Requirement 1

WarpPipe, Lava extends *Ground*
All concrete classes that inherit the abstract *Ground* Class have the necessary methods to provide proper functionality for every ground in the game map.

TeleportAction extends *Action*
All concrete classes that inherit the abstract *Action* class have the necessary methods to execute a certain action and provide a text description before (menuDescription) and after execution.

New map created in Application - Lava Zone
Map is created as a list of Array Lists just like how the first game map is created. This map consists of only 3 ground types, which are **Lava, Dirt** and one **WarpPipe**. The actor can travel from one map to another through a **WarpPipe**.
Application --<<creates>>--> Lava

Lava ground objects throughout the new map are instantiated by passing a new Lava instance into the **FancyGroundFactory** constructor.

Application --<<creates>>--> WarpPipe
WarpPipe objects throughout both maps are instantiated individually manually in the application, this enables us to pass in the location of the Warp Pipe into its constructor. There are two constructors to create a new **WarpPipe** instance, one has an additional parameter **WarpPipe connectedPipe,** to pass into the constructor. This is for warp pipes that are located on the main game map as there will be only one possible destination (in other words one possible warp pipe to be connected to). However, for the warp pipe located in the second map (Lava Zone) has no such parameter to pass in when instantiating it as it would only know which warp pipe (on the main game map) it will be connected to after the actor executes the teleport action.

WarpPipe –<<stores>>–> WarpPipe
WarpPipe has an association with itself as it needs to save the warp pipe that is connected to itself in order to have access to the location of the connected warp pipe.

WarpPipe --<<creates>>--> PiranhaPlant
Each **WarpPipe** can only create one **PiranhaPlant** once (unless **ResetAction** is executed), and it is kept track with a boolean. Upon creating a Piranha Plant, it adds the capability ***HAS_PIRANHA***. The allowed actions (**TeleportAction, JumpAction)** are dependent on said capability. The player cannot jump onto Warpipe on the first round before PiranhaPlant has been spawn. The player may only jump on warp pipe upon killing the plant.

TeleportAction –<<stores>>–> WarpPipe
TeleportAction stores two **WarpPipe** instances, the current warp pipe actor is on, and the connected warp pipe which would be the destination to move said actor to after the execution of this action. After teleport action is executed, it sets up a connection between the current and connected warp pipe.

<u>Requirement 2</u>

Npc* extends *Actor
An abstract class for non-playable characters that are not categorized as an enemy, such as Toad.

Application ---<<creates>>---> FlyingKoopa
Application ---<<creates>>---> Bowser
Application ---<<creates>>---> PrincessPeach

The application creates 2 new different Enemy, namely FlyingKoopa and Bowser, extending from Enemy. Both FlyingKoopa and Bowser have a hashmap of pairs as its attribute. This allows both enemies characters to possess different sets of behaviours to perform during their play turn.

Flying Koopa has almost the same functionalities compared to Koopa, the only additional advantage it gets is that it can fly over any obstructing grounds like walls or trees. As for Bowser, it does not move and attacks or follows only when Mario is on any of Bowser's surrounding adjacent locations. It drops a key for Mario to pick in order to interact with PrincessPeach and end the game. Another unique functionality it has is that he leaves fire on the ground of the target, dealing overtime damage for 3 turns.

Application creates PrincessPeach which extends from Npc, which is a new child class of Actor, for she is not an enemy and only serves as the ender of the application. Mario can interact with Princess peach but needs to retrieve the Key item prior to interaction to end the game.

Hence, this shows that both classes apply the Single Responsibility Principle, as they are just mere NPCs with their own set of behaviours and also providable actions to Mario. Besides that, not only these classes, but to all NPC type characters like Toad, PrincessPeach and all Enemy, it is clear that all of them follows the Dependency Inversion principle strictly as they gain their Behaviours from an external interface class instead of directly from the Application itself.

Application ---<<creates>>---> WarpPipe ---<<creates>>---> PiranhaPlant
Application ---<<creates>>---> Mature ---<<creates>>---> FlyingKoopa

Moreover, Application also creates WarpPipe and Mature tree that spawns or creates PiranhaPlant and FlyingKoopa respectively. PiranhaPlant only attacks and cannot perform any actions that revolve around moving its position. This also shows that Piranha Plant adheres to the Single Responsibility Principle.

Bowser ---<<creates>>---> Key

Bowser, when eliminated, will drop a Key. This Key item adds capability to Mario in order to show PrincessPeach that does possess said item to perform certain action. When this item is dropped, it can also prompt Mario to either pick it up or not as it will be shown on the map. This class also serves one purpose, proving that it also conforms to Single Responsibility Principle, which is to add a capability to actors.

RetrieveAction extends Action

This action inherits Action class, allowing it to access necessary methods to perform the action and provide a text description before and after execution.

Key item is what grants Mario, the player Actor, with a GOTKEY status to show that Mario has the Key. However, RetrieveAction is an action for Key where by default, whoever who holds the item or has it in the inventory will have this action run, and pass said Status to that actor. The approach is simplistic and direct as it only passes the item to the actor, giving it a certain status to it. With that, it is apparent that this action class aligns with the Single Responsibility Principle as it serves one and one purpose only, which is action to item.

EndAction extends Action

This action inherits Action class, allowing it to access necessary methods to perform the action and provide a text description before and after execution, as per the usuals.

Mario being near PrincessPeach is what triggers this prompt or action, and this action checks for player or Mario to see if they possess a key or not, and depending on whether it is true or false, it will run a specific action. The main goal of this action is to end the game when the user interacts with PrincessPeach and also owning a key, else prompting a simple message indicating that the player does not have a key yet. The approach is also rather straightforward as it checks for one specific condition in order to perform certain action(s). Therefore, it shows that this action is sticking to the Single Responsibility Principle, for all it does is an action to an actor.

AttackAction ---<<creates>>---> Fire

This relationship shows that AttackAction creates Fire, a ground type object, however, indirectly, Fire comes from Bowser. Everytime Bowser makes a successful attack, it will leave a fire on the ground. The conditions to when it leaves a fire, is that when Bowser attacks and the ground is still not a Fire ground yet, it will then replace the ground with a new Fire ground. This fire ground will last for 3 rounds, dealing 20 damage to all actors on it. Another conditional that could be included is that if ground is already a Fire ground, it can just extend its amount of rounds to stay there, however, we ignored this condition for simplicity.

There are a total of 2 implementations. The first one would be by passing a Status to any of the affected ground. This would mean that whichever ground that is affected will go through specific changes under its particular class. This is not very ideal as it violates the "Don't Repeat Yourself" principle. This happens when you repeatedly need to alter every possible ground type class that could be affected by this fire effect. Besides, another scenario could cause some error, which is when a Tree withers, and when it does, it also halts all processes and gets removed, leading to an incomplete and illogical fire function.

The more ideal approach would be to implement a new Ground type class that takes in a ground type argument. The input is simply the very affected ground, by taking it in as an argument it halts all possible action that ground should do and lets Fire do its own thing for 3 turns. After 3 turns, it will replace itself with the former ground object so it can continue with its process from where it stopped (which is when it was taken in as an argument). The final design aligns with the Single Responsibility Principle as it serves only one purpose, which is to be initialiseApplication ---<<creates>>---> FlyingKoopa

Requirement 3

PowerFountain, HealthFountain extends Fountain extends Ground.
The implementation of the Fountain abstract class avoids the need to repeat the same codes. It allows us to define same similar methods that can be used by its child classes and helps in terms of extensibility.

Bottle, PowerWater, HealthWater extends Item.
Bottle is implemented as a separate class which has an ArrayList as its attribute that stores PowerWater and HealthWater objects. Accessing and modifying this ArrayList works like a stack, whereby the last added item into the list is first popped. Mario is only able to get the buff features of magic water by filling up the bottle from magic fountain and drinking from it(consumeAction). The Bottle

GetAction, RefillAction extends Action
The optional Challenges are implemented whereby, the player gets the bottle from toad (instead of trading using money) using the GetAction class. This is inline with the Single Responsibility Principle as we separate the features and do not have different implementations of obtaining items (with and without money) inside a single class. In order to obtain the magic water, player moves to fountain and uses the RefillAction to store water objects in the bottle. Since RefillAction has a one to one association with Fountain, we are able to create and store water objects inside bottle corresponding to the fountain the player is standing on.

DrinkBehaviour <<implements>> Behaviour
DrinkBehaviour <<creates>> DrinkAction
In order to allow Enemy child class objects to drink water from fountain, DrinkBehaviour is created whereby, the DrinkAction would be added to the hashmap of Enemy class. The DrinkBehaviour is set as the 3rd priority between follow/wander and speech behaviours. When an enemy uses DrinkAction, the sip account for 5 water volume.

Fountain, Bottle <<implements>> Resettable
When user hits reset, Child classes of Fountain will go back to their original volume and reset their recharge timers. The Bottle will also be cleared of all magic waters.

Requirement 4

FireFlower extends Item
(Sprout <<creates>> Sapling), (Sapling<<creates>>Mature) <<creates>> FireFlower
FireFlower inherits all the necessary methods from Item class for it to function effectively .
Using the imported random class for probabilistic creation, FireFlower is created 50% of the time during the growth process between sprout and sapling, and sapling and Mature. There is no need for the creation of new action to obtain its powers and refactoring the ConsumeAction allows us to avoid repetitive classes.

Fire extends Ground
Fire <<stores>> Ground
AttackAction <<creates>> Fire
Fire class inherits the necessary methods from Ground for it to appear on the map posing certain features of its own. The Fire lasts for 3 turns and since it has a one to one association with Ground class, it stores the ground of which the fire is set. Once the timer ends, the ground is restored to its former version. When an actor consumes the FirePlant, it gains a Fire_Power capability. In the AttackAction class, we reuse the same code that was implemented for Bowser since the fire attacks are the same and this avoids having repetitive code in different classes. The player's attack is changed to fire attack and lasts for 20 turns.

Requirement 5

SpeakAction, InteractAction extends *Action*
All concrete classes that inherit the abstract *Action* class have the necessary methods to execute a certain action and provide a text description before (menuDescription) and after execution.

SpeakAction: An action that caters to NPCs that are not to be interacted with which enables them to 'speak'.

InteractAction: An action that caters to NPCs that the player can interact with (in this case it is just Toad) and allows the NPC to 'speak' according to the conditions given.

***Npc*, *Enemy* –<<acts>>–> SpeakBehaviour**
SpeakBehaviour determines whether to return **InteractAction, SpeakAction** or **DoNothingActon.** This behaviour checks for the ***CAN_SPEAK*** capability which is added to the actor every two rounds (implemented in actor's playTurn() method), and decides to return one out of the three actions. This is implemented as a behaviour instead of just having the **SpeakAction** instance added to the actor's playTurn() method so that we can also implement the idea of priorities between a list of possible actions that an actor can perform.

SpeakAction, InteractAction –<<uses>>–> Monologue
An external **Monologue** class enables us to avoid the possibility of tight coupling between different classes. Let's say if the array list of possible sentences for each actor are stored in the respective actor's class, this would create multiple associations between **InteractAction, SpeakAction** and the classes that represent each actor. Here only **SpeakAction** and **InteractAction** require an association towards **Monologue** in order to retrieve the array list of possible sentences.

Monologue: A class that stores all the possible monologues for each actor in a hashmap *(<key: String actor's name, value: ArrayList<String> List of possible sentences).* It also provides a method to retrieve the array list by passing in the actor's name.

***Npc, Enemy* implements Resettable**
This just allows the number of play turns to be reset once reset action is executed

Work Breakdown Agreement

Members:

1. Matin Raj Sundara Raj (32124260)
2. Emily Chin Jing Yi (32457456)
3. Tan Sheng Huang (31206743)

We hereby agree to split the workload as the following:

1. Requirement 1:
    a. Initial Coding & Design doc update - Emily (Completion by: Saturday 14th May)
    b. Review by – Matin Raj

2. Requirement 2:
    a. Coding & Design doc update – Sheng Huang (Completion by: Monday 16th May)
    b. Review by – Matin Raj

3. Requirement 3:
    a. Coding & Design doc update – Matin Raj (Completion by: Saturday 14th May)
    b. Review by - Emily

4. Requirement 4:
    a. Coding & Design doc update – Matin Raj (Completion by: Thursday 19th May)
    b. Review by – Sheng Huang

5. Requirement 5:
    a. Coding & Design doc update – Emily (Completion by: Thursday 19th May)
    b. Review by – Sheng Huang

6. New Sequence Diagram – Sheng Huang (Completion by: Thursday 19th May)
    a. Draft and implementation
    b. Review by – Matin Raj

Finalization of code, review & modifications:

- To be done together (Completion by: Friday 20th May)
- Final submission (Completion by: Sunday 22nd May)

Signed by:

1. I accept this WBA – Matin Raj
2. I accept this WBA – Tan Sheng Huang
3. I accept this WBA – Emily Chin

FIT2099
Assignment 2

Work Breakdown Agreement

Members:

1. Matin Raj Sundara Raj (32124260)
2. Emily Chin Jing Yi (32457456)
3. Tan Sheng Huang (31206743)

We hereby agree to split the workload as the following:

1. Requirement 1:
    c. Coding & Design doc update - Emily (Completion by: Thursday 28th April)
    d. Review by – Matin Raj

2. Requirement 2:
    a. Coding & Design doc update – Sheng Huang (Completion by: Saturday 30th April)
    b. Review by – Matin Raj

3. Requirement 3:
    a. Coding & Design doc update – Sheng Huang (Completion by: Thursday 28th April)
    b. Review by - Emily

4. Requirement 4:
    a. Coding & Design doc update – Matin Raj (Completion by: Thursday 28th April)
    b. Review by – Emily

5. Requirement 5:
    a. Coding & Design doc update – Matin Raj (Completion by: Saturday 30th April)
    b. Review by – Sheng Huang

6. Requirement 6:
    a. Coding & Design doc update – Emily (Completion by: Saturday 30th April)
    b. Review by – Sheng Huang

7. Requirement 7:
    a. Coding & Design doc update – Together (Completion by: Saturday 30th April)
    b. Review by – Everyone

Finalization of code & commenting:

- To be done together (Completion by: Sunday 1st May)

Signed by:

1. I accept this WBA – Matin Raj
2. I accept this WBA – Emily Chin
3. I accept this WBA – Tan Sheng Huang

FIT2099
Assignment 1

Work Breakdown Agreement

Members:

1. Matin Raj Sundara Raj (32124260)
2. Emily Chin Jing Yi (32457456)
3. Tan Sheng Huang (31206743)

We hereby agree to split the workload as the following:

1. Requirement 1:
    a. UML Class Diagram - Emily (Completion by: Friday 8th April)
    b. Review by – Matin

2. Requirement 2:
    a. UML Class Diagram – Sheng Huang (Completion by: Friday 8th April)
    b. Review by – Matin Raj

3. Requirement 3:
    a. UML Class Diagram – Sheng Huang (Completion by: Friday 8th April)
    b. Review by - Emily

4. Requirement 4:
    a. UML Class Diagram – Matin Raj (Completion by: Friday 8th April)
    b. Review by – Emily

5. Requirement 5:
    a. UML Class Diagram – Matin Raj (Completion by: Friday 8th April)
    b. Review by – Sheng Huang

6. Requirement 6:
    a. UML Class Diagram – Emily (Completion by: Friday 8th April)
    b. Review by – Sheng Huang

7. Requirement 7:
    a. UML Class Diagram – Everyone
    b. Review by – Everyone

UML Sequence Diagrams

- To be done together (Completion by: Saturday 9th April)

Design Rationale

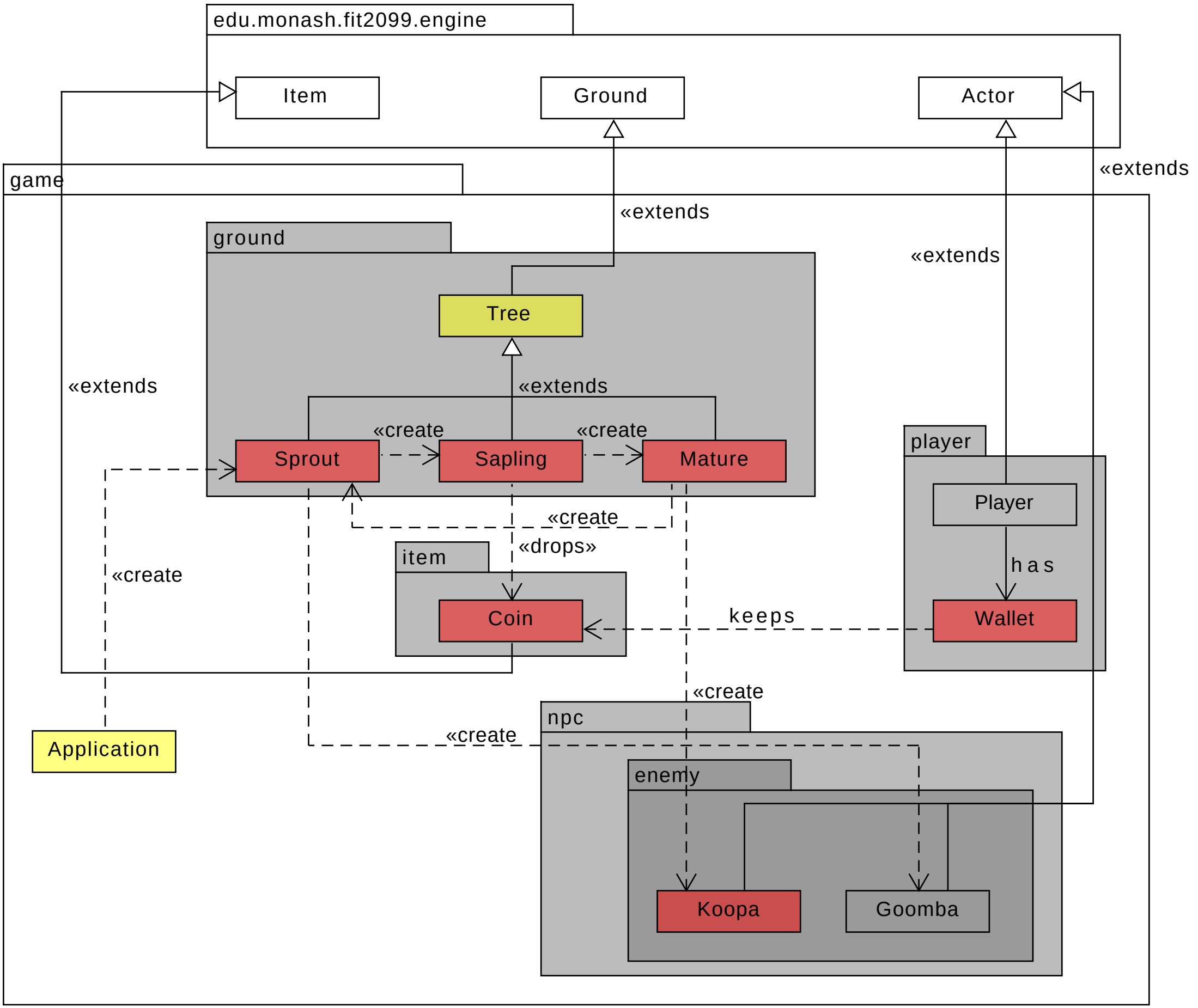
- To be done together (Completion by: Sunday 10th April)

Signed by:

1. I accept this WBA – Matin Raj
2. I accept this WBA – Emily Chin
3. I accept this WBA – Tan Sheng Huang

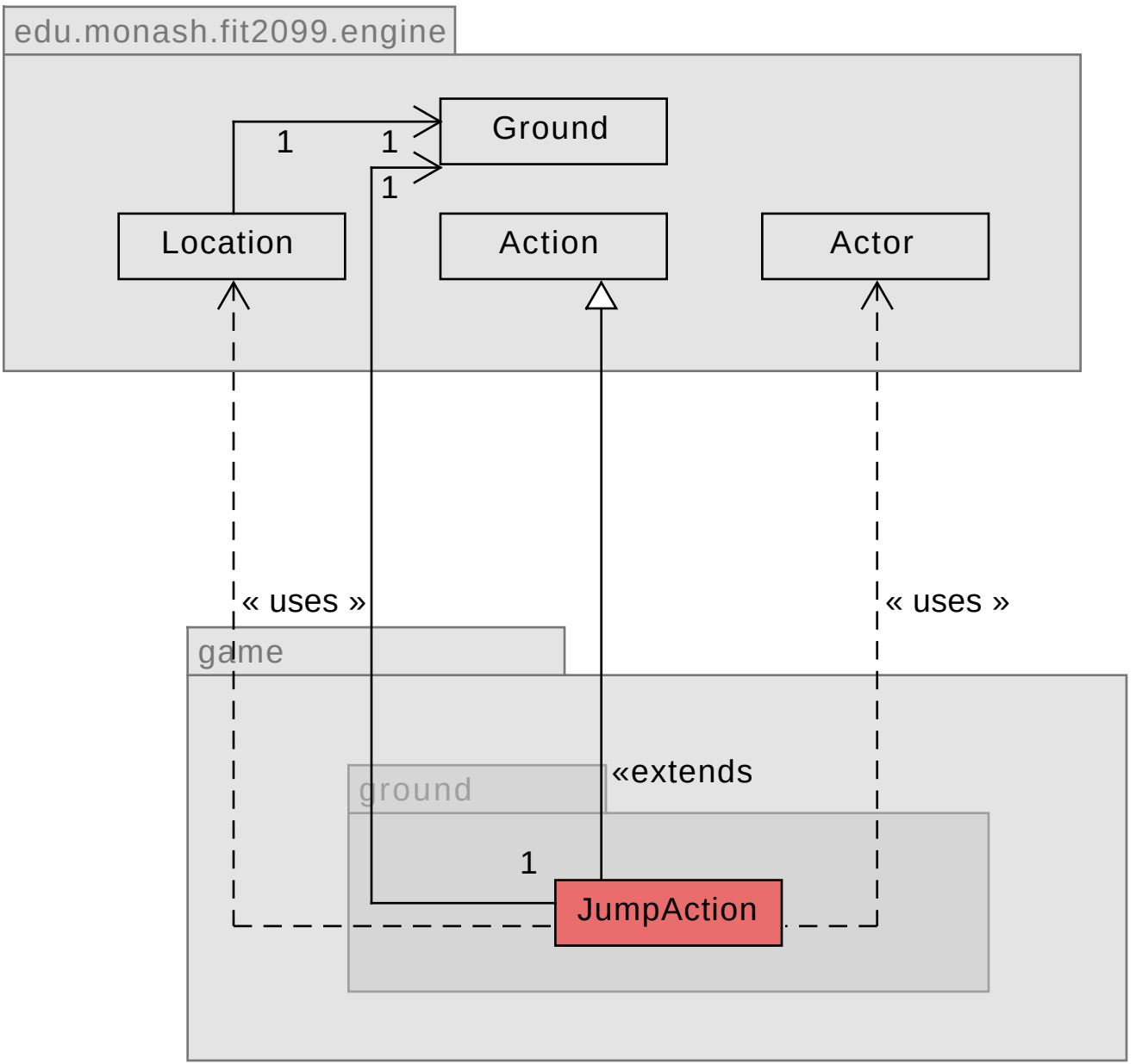
Assignment 2

Requirement 1

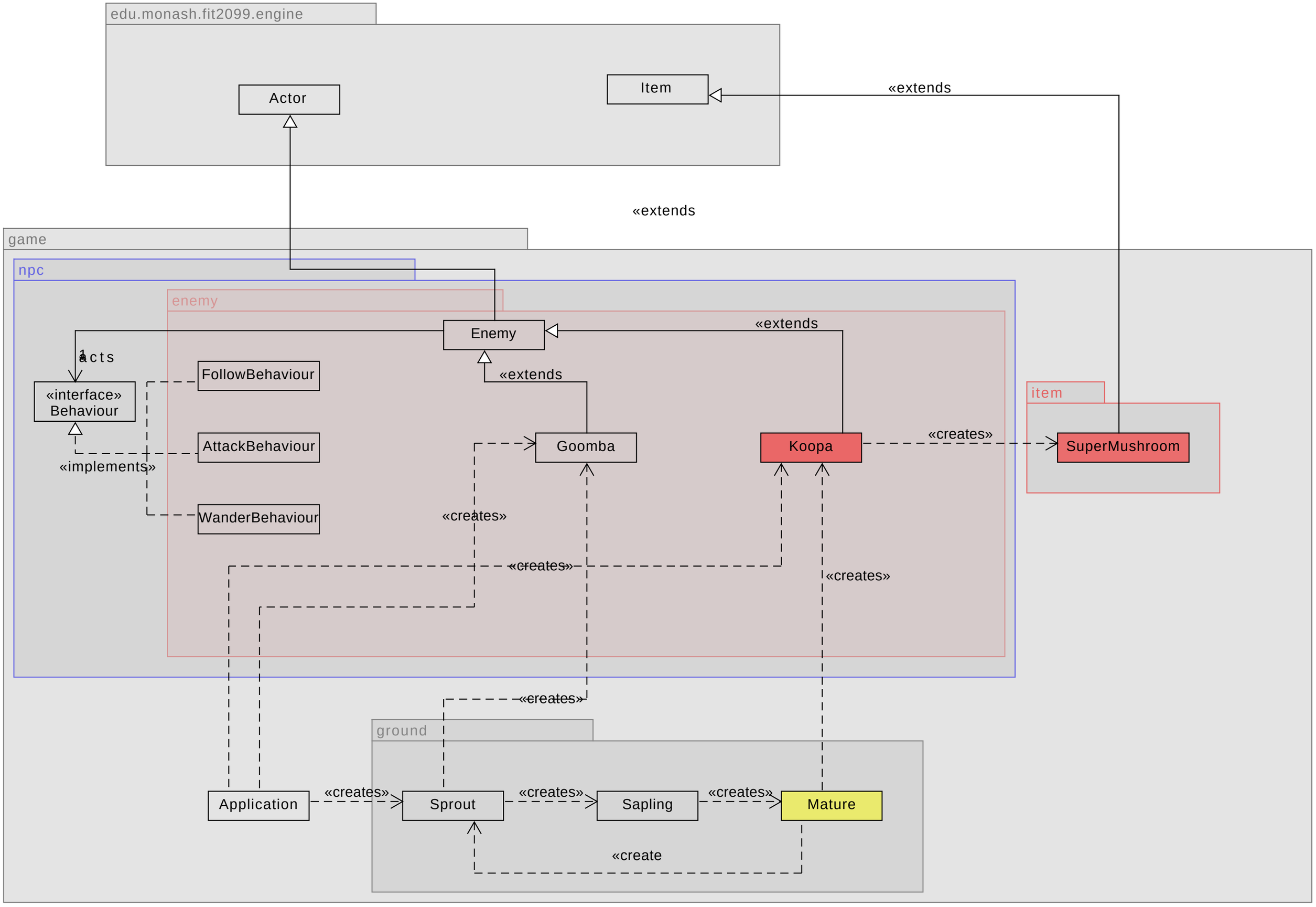


Assignment 2

Requirement 2

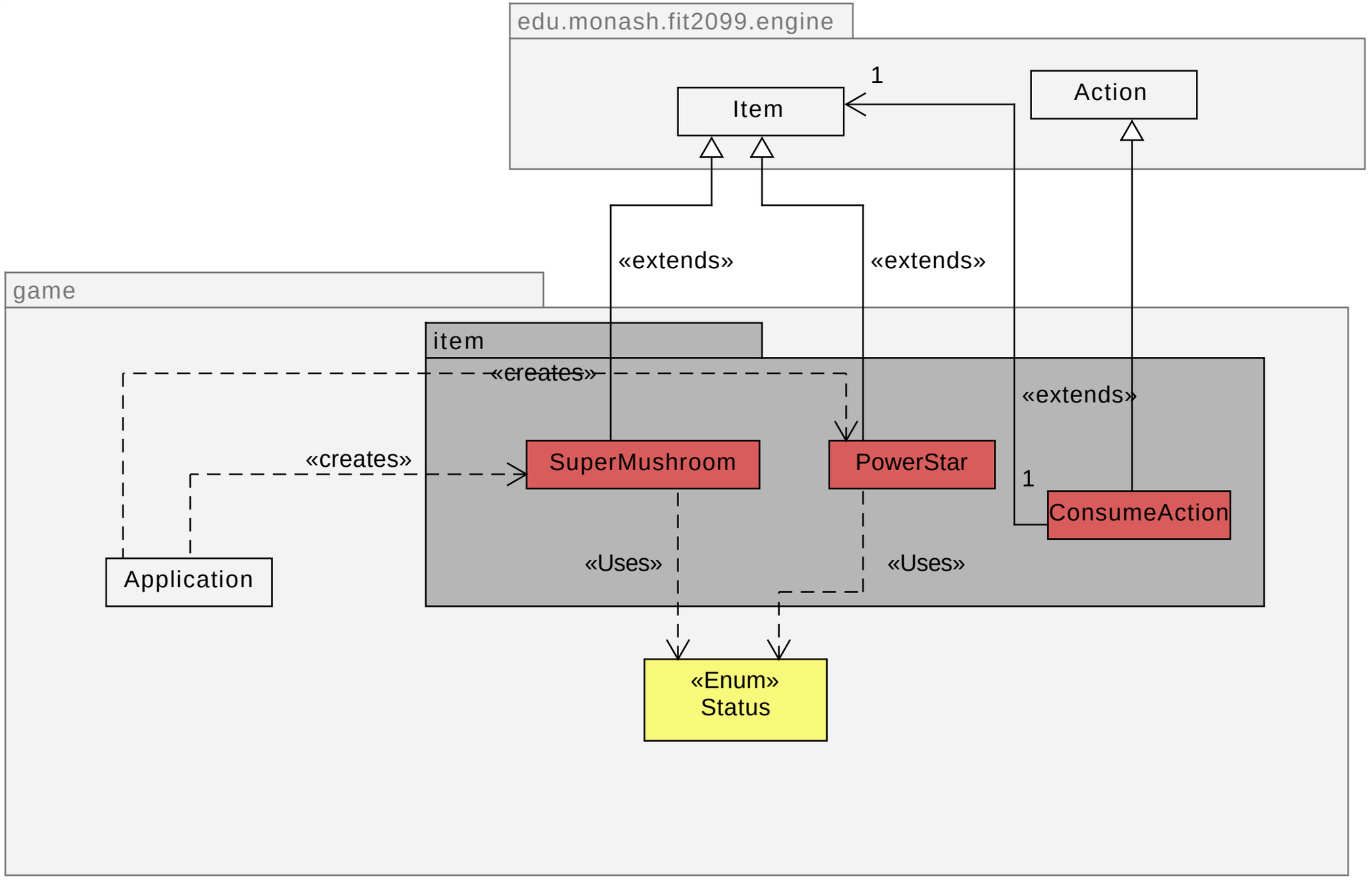


Requirement 3



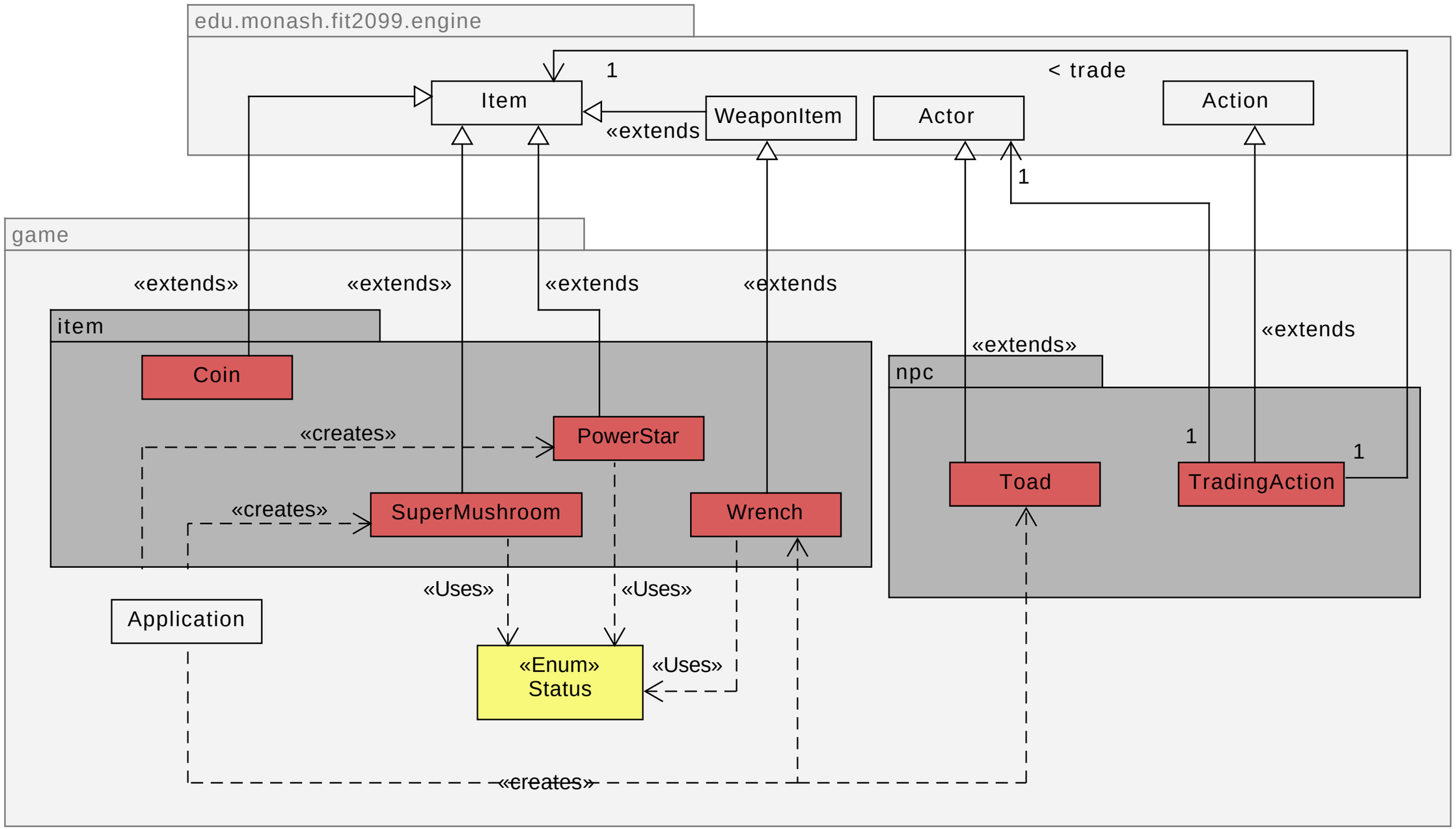
Assignment 2

Requirement 4

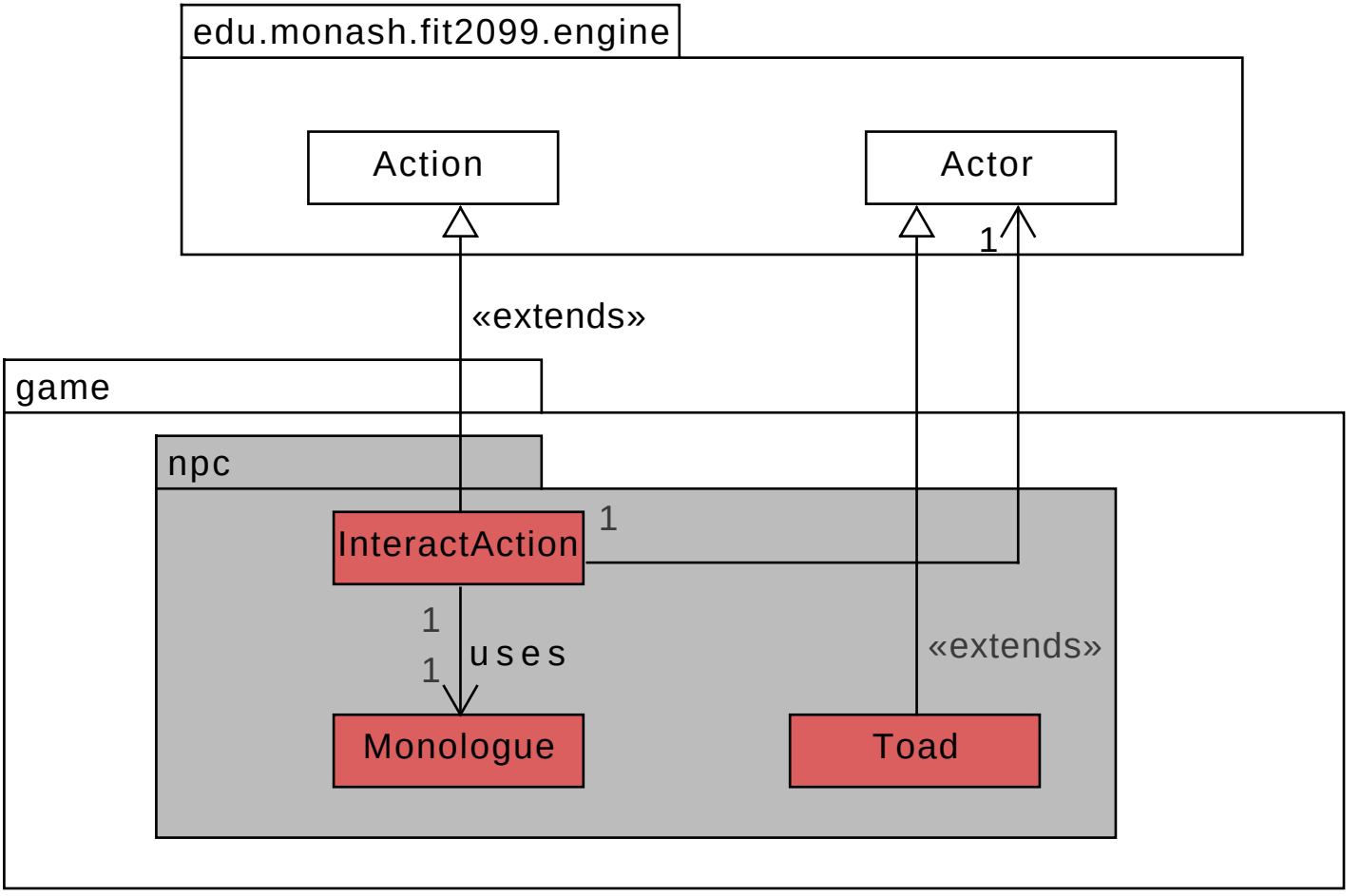


Assignment 2

Requirement 5

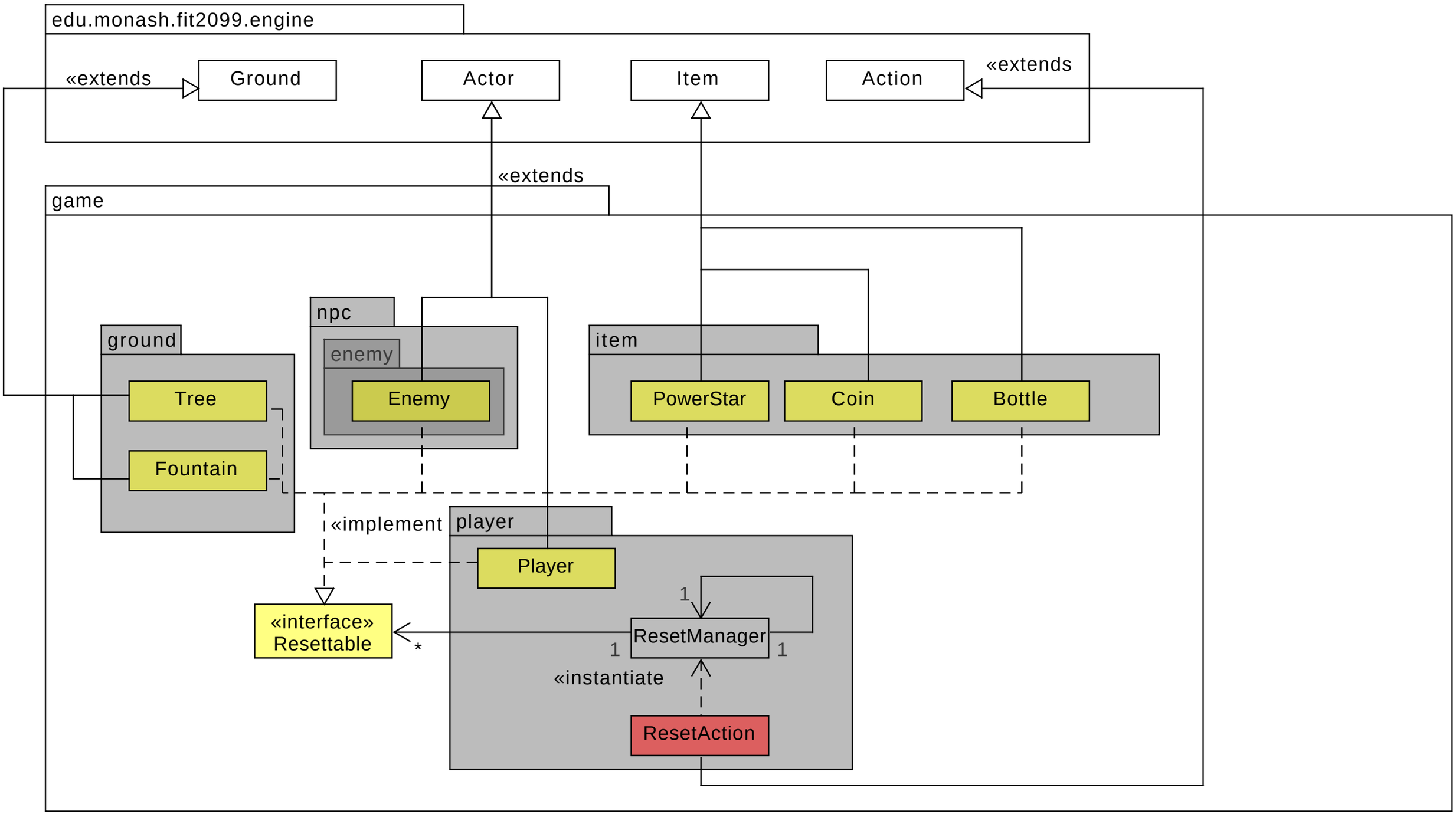


Assignment 2
Requirement 6

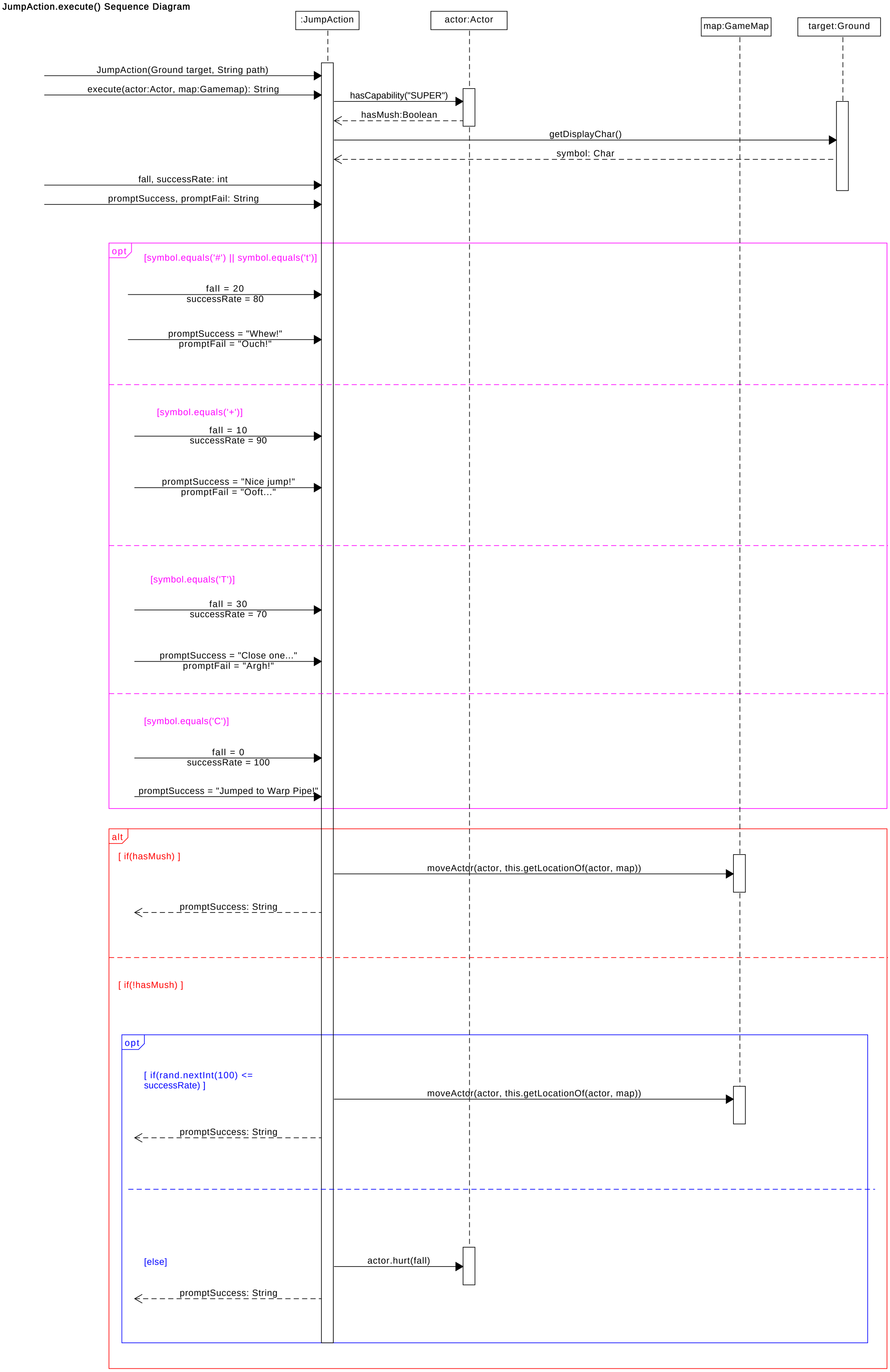


Assignment 2

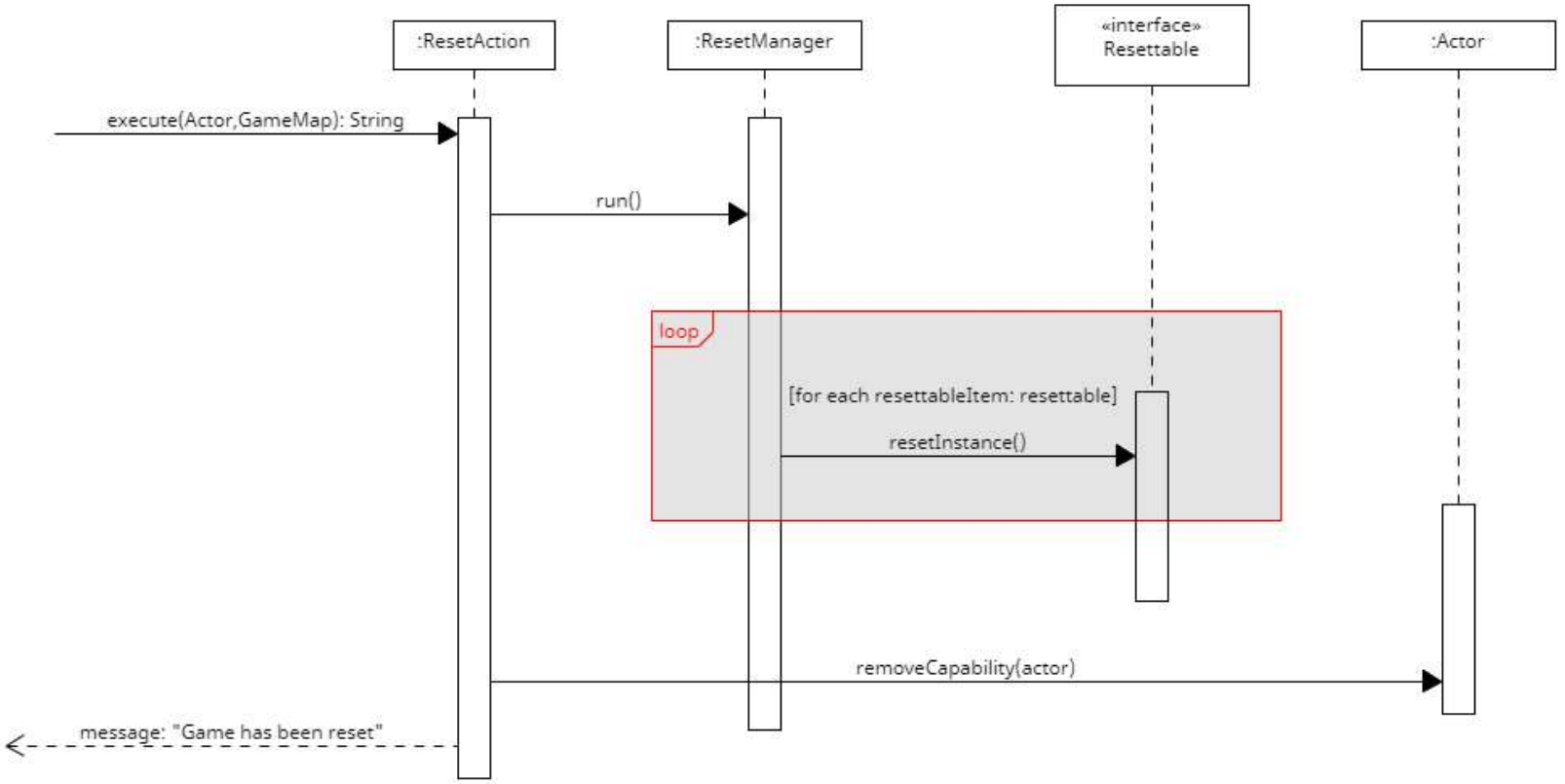
Requirement 7



JumpAction.execute() Sequence Diagram



ResetAction.execute() Sequence Diagram



Design Rationale

Requirement 1

Changes

Wallet Class

Originally, it was intended to have **Wallet** as an attribute inside the **Player** class instead of a class on its own. However, by having it as a class on its own, this adheres to the Single-Responsibility Principle, rather than having the **Player** class to keep track of the balance of coins.

Previous design

Coin --> <<stores>> --> Coin

The circular association between Coin objects was implemented with the intention to ease the process of resetting the game when implementing Requirement 7. However it was rather redundant after being able to understand and utilize the base code.

Tree* extends *Ground

All concrete classes that inherit the abstract *Ground* Class have the necessary methods to provide proper functionality for every ground in the game map.

Sprout, Sapling, Mature extend *Tree*

All concrete classes that inherit the abstract *Tree* Class have the necessary methods to grow, wither and spawn items or enemies.

Application --<<creates>>--> Sprout --<<creates>>--> Sapling --<<creates>>--> Mature

A tree in the game has three cycles with unique abilities at each stage. A naïve alternative implementation for this requirement would be similar to that in demo.conwayslife 's Tree class whereby, all of the logic is coded within the same class. The inclusion of special abilities at each stage means we would have a convoluted and un-modularised code.

Hence, instead of having **Application** create ***Tree*** directly, we have **Application** create **Sprout** and make ***Tree*** an abstract class since **Sprout, Sapling** and **Mature** are indeed Trees that share the same basic attributes and functionalities. This also allows for different implementations of certain shared functionalities like spawning enemies. We have thus managed these objects through separation of concerns, in adherence to the SRP (Single-Responsibility Principle) which states that each class should have a specific responsibility. By making the ***Tree*** class abstract, it avoids being a GOD class and conforms to both the Dependency Inversion Principle and the Open-closed Principle where in case of any changes that are necessary to be made to either subclass, there will be no need to make changes in the parent class. This increases maintainability and encourages extensions of new features to the Trees.

Requirement 2

JumpAction extends *Action*
Since JumpAction class inherits the Action class, it would have the necessary methods to perform the action and provide a text description before and after execution.

Ground objects (ie. Wall, Tree) are the ones that grant Mario, the player Actor, the action to jump onto them (be on the same location as the Ground objects). One of the possible approaches was to implement JumpAction as a method in the Player class. It would certainly work fine since this action is only used by the Player. However, it would mean that maintaining that action and its possible future extensions would be hard. Hence, it is better to extend Action and implement it as a separate class. By having the Actor class as an attribute in JumpAction (one-to-one association), we can allow this action to be carried out by other actors in the future if we need to. Besides, code management would be much cleaner. Our final design aligns with the Single Responsibility Principle as it serves only one purpose or functionality, which is an action to an actor.

As a general explanation of what this class does, it provides the player a way to move up to the high ground, which is also called jump. This can only be instantiated/added when the Player is near a Ground object of type Tree or Wall. When near, Player will have the option to execute the JumpAction. Depending on the obstacle ahead, the Player will be tested with different jump rates, meaning that jumping up also has a certain success rate. However, if Player has the 'SUPER' capability, obtained after eating the Super Mushroom, the Player will be able to jump freely with 100% success rate and no fall damage. Finally, when the Player successfully moves to the new position, it should not replace the Ground object.

Previous Design
The GameMap registers Player as an existing actor on that location, so when Player moves away, whichever Ground object that was there previously would be there again as usual.

Changes
The new JumpAction() method does not register Player to the location, it should just plainly shift the position of the Player.

In terms of actual changes, the initial implementation of this class was supposed to be done within the Player's class. However, it was not possible as it does not have a method to instantiate this action to itself, therefore, the alternative method was to instantiate JumpAction() in the Ground objects that can obstruct players, namely the Tree and Wall.

Requirement 3

Changes

Goomba, Koopa extends Enemy extends Actor

The initial plan was to let Goomba and Koopa to just extend its functionalities from Actor class since it was mostly using only Actor, however it was then made known that an Enemy class could make things more convenient in an event where certain class or methods are required to be implemented to only Goomba and Koopa, for example, Resettable class. This class still adheres to the Single Responsibility Principle.

Previous Implementation

Goomba, Koopa extends *Actor*

By inheriting Actor class, Goom and Koopa now have access to methods that allow them to function as characters (actor).

As Goomba and Koopa are supposed to be individual objects that have different characteristics and traits (which require different implementation of methods), it is only logical to create them separately as different classes. This allows us to maintain them individually. Besides, they do not share similar functionalities when it comes to non-inheritable functions. It does not justify the creation of an abstract Enemy class as the majority of the codes involve those of the Actor anyway. If an Enemy class is used, then further changes to some implementations or even additions are necessary, which would not adhere to the DRY principle. Therefore, we are able to reduce potential dependencies on other classes by inheriting Actor instead. Hence, this shows that both classes apply the Single Responsibility Principle.

Application ---<<creates>>---> Goomba
Application ---<<create>>---> Koopa

The application creates both Goomba and Koopa at the start of the game. The existing Application class needs to be modified to accommodate this. Koopa, just like Goomba, has a hashmap of <Integer, Behavior> pair as its attribute (hence, the association relationship to behavior interface). This allows Koopa to have more complex actions such as the FollowBehavior which allows to follow the player around the map.

Sprout ---<<create>>---> Goomba
Mature ---<<create>>---> Koopa

Moreover, Application also creates Sprout. Sprout experiences growth and follows a cycle and has a certain percentage of possibly spawning Goomba in its location at every turn. Similarly, Koopa may be spawned by Mature tree.

Requirement 4

SuperMushroom, PowerStar extends *Item*

SuperMushroom and PowerStar are best implemented as items in the game by inheriting the Item class as they are meant to be portable and picked up or dropped by the actor. They both now have access to the necessary methods to be displayed on the map and hold their own separate characteristics.

ConsumeAction extends *Action*

Since our Player needs a way to interact with the items and gain powers, it is better to implement it as a separate action than to code the logic in the Player's class. This would significantly improve readability and ensures the separation of concerns. Inheriting the Action class also means that we will be able to display the action option through the menuDescription() method and return a string after the execution. The main logic can be housed inside the execute() method, where we could call Player's methods such as resetMaxHp() and increase their capabilities. Instead of having a direct association with only SuperMushroom and PowerStar classes, we opt to have the Item class as one of the attributes (one-to-one association relationship), as it allows us to have a general implementation to other items that may require similar action in the future. Maintaining and expanding the game functionality would be much easier. This ConsumeAction would then be added to the allowableActions list of SuperMushroom and PowerStar.

SuperMushroom, PowerStar --<<Uses>>--> Status

Status is an enum class that contains statuses which can be used as a means of indicating actions or abilities that can be attached or detached. By using this class, we can know when the Player has consumed an item. For example, in the execute() method, if the Player consumes a SuperMushroom, then we can add a capability stating that the Player is 'SUPER'. This indication of the Player's status can be used to determine the outcomes of other actions such as JumpAction where the player could jump freely when he has the 'SUPER' capability (consumed a mushroom). For PowerStar, we can use an enum to indicate whether it has been consumed or not. This would help us to reset the life counter as Power stars have a fixed life span of 10 turns until it disappears or fades from the Player. While this may not seem as straightforward, it reduces the amount of attributes need to check for the statuses of the actor or items. This is in line with the Single Responsibility Principle where classes should not have too many responsibilities.

Requirement 5

Previous design:
Coin extends Item
Coin is an item that the Player could collect which would increase their wallet balance. The coin class would be responsible for creating coins that hold a certain value. We have decided to keep track of all Coin objects that are on the ground by using an ArrayList. This is because, it would be easier to have the location of all these coin objects when resetting the game as they would need to be removed.

Changes in Coin Class:
As mentioned in Requirement 1, there is no need to store the location of all coin objects to implement Requirement 7 (Resetting the game) as we can implement the removal of all Coin objects on the ground by overriding the tick() method which is already implemented in the ***Item*** class.

Wrench uses <<Enum>> Status
We use the status enum to give capability to wrench since it is capable of destroying shells. Thus the actor who has a wrench in their inventory will get the capability too.

TradingAction association with Item, Actor
TradingAction class now has a one to one association with Item and Actor objects as they as used as attributes in its class. This is so we are able to keep track of the actor who possesses the item which would allow use to add or remove capabilities accordingly. Next, the Item is stored to keep track of the type of Item that is being traded such as PowerStar, SuperMushrrom and Wrench.

Wrench extends Weapon*Item*
WeaponItem extends *Item.* Hence, it is wise for us to implement Wrench as a class that extends the WeaponItem class, instead of the Item class. This is because, in order for us to use the Wrench as a weapon against enemies, we would have to implement certain attributes that would provide damage. Since these extra capabilities are already provided by the abstract WeaponItem Class, we inherit that instead.

TradingAction extends *Action*
By inheriting the Action class, TradingAction has access to the necessary methods that facilitate the execution of the trade logic while also displaying the menu description and text of the action that transpired during the game.

The TradeAction is responsible for the trade logic whereby, in the execute method, it checks if the Player has enough money in their wallet for the item that they would like to purchase. If it is enough, then the item is added to the Player's inventory, if not a message is printed. By having a dedicated trade class, we can perform similar logic to any item that the Player buys. An alternative way would be to have this method included in each item that could be traded which involves a lot of repeated codes and it will not be easy to maintain or make changes. By having a general trade action that has the Item class as its attribute (one-to-one association), we are able to implement this action to any item if needed. Besides that, we

have opted for a simple wallet system where the Player's money is kept as an integer in an attribute. This is because the transaction of coins does not require the object itself and creating a separate wallet class would be redundant for its simple intended usage.

Requirement 6

Changes

Monologue Class

An addition of the **Monologue** class to store all the possible monologues that **Toad** will display according to certain conditions in an array list. The list of possible monologues was supposed to be stored in **Toad** class itself, but this does not accommodate for future extensions, for example adding new possible monologues for Toad to display when interacted with.

InteractAction → <<uses>> → Monologue

InteractAction has an instance of the Monologue class in order to filter out the monologues to randomly choose from under certain conditions.

Previous design

InteractAction extends *Action*

All concrete classes that inherit abstract *Action* Class have the necessary methods to execute a certain action while providing a written text before, through menuDescription() method, and after the execute() method, by returning a string of the outcome.

InteractAction:

A general explanation of how this class works is that it provides a function where the **Player** can interact with Toad. ***Toad*** extends the actor class which allows it to inherit its methods. **InteractAction** would be added to Toad's list of allowable actions for the player to execute. The **execute()** method in this class will check whether the actor or in our case, the player has a Wrench in their inventory or whether they have consumed a Power Star. it will then return a String from an array list that stores the four possible sentences that Toad will respond with when the player chooses to interact with Toad. We could add more sentences if required, by using the add() method for ArrayList compared to manually increasing an array's size. Alternatively, this functionality could be implemented directly inside the **Toad** class or inside the Player's playTurn() method, but the action would only be applicable to Player interacting with Toad. Not to mention, that it would violate the Single Responsibility Principle where each class should have a focused responsibility. Implementing the InteractAction as a separate class and having an association with the *actor* will definitely make it easier to maintain and accommodate for extensions to have other actors interact in the future.

Requirement 7

Changes

***Tree,* Goomba, Koopa, Player, Coin, PowerStar implement Resettable:**

The resetInstance() implemented in the respective classes that implements the Resettable interface is just so that the items / players have the status to be resetted (which in other words means that ResetAction has been executed by Player). The actual implementation of how the item/actor is resetted is in either tick() for Items or playTurn() for Actors.

The **ResetManager** will then call resetInstance() for all Resettable Objects that are stored in the resettableList when **ResetAction** is executed. The main game loop will then handle the actual resetting of Items/Actors.

Previous design

***Tree,* Goomba, Koopa, Player, Coin --<<implement>>--> Resettable**

All classes that implement the Resettable interface will have the necessary methods to provide the functionality to reset the objects of each respective class.

ResetAction extends *Action*

All concrete classes that inherit the abstract *Action* class have the necessary methods to execute a certain action and provide a text description before (menuDescription) and after execution.

ResetAction:

A general idea of how this class works is that it creates an instance of the ResetManager in order to access the array list of Resettable Items. Subsequently, in the **execute()** method, it can call **run()** on all the items in the Array List.

***Tree,* Goomba, Koopa, Player, Coin implement Resettable:**

Instead of all the above classes having a direct relationship with **ResetManager**, we have decided to make use of the **Resettable** interface in accordance with the Liskov Substitution Principle. This is because all the classes above have different requirements when it comes to implementing the reset function in their respective classes. The functionality of ResetManager remains unchanged even though more classes require a reset function in them.