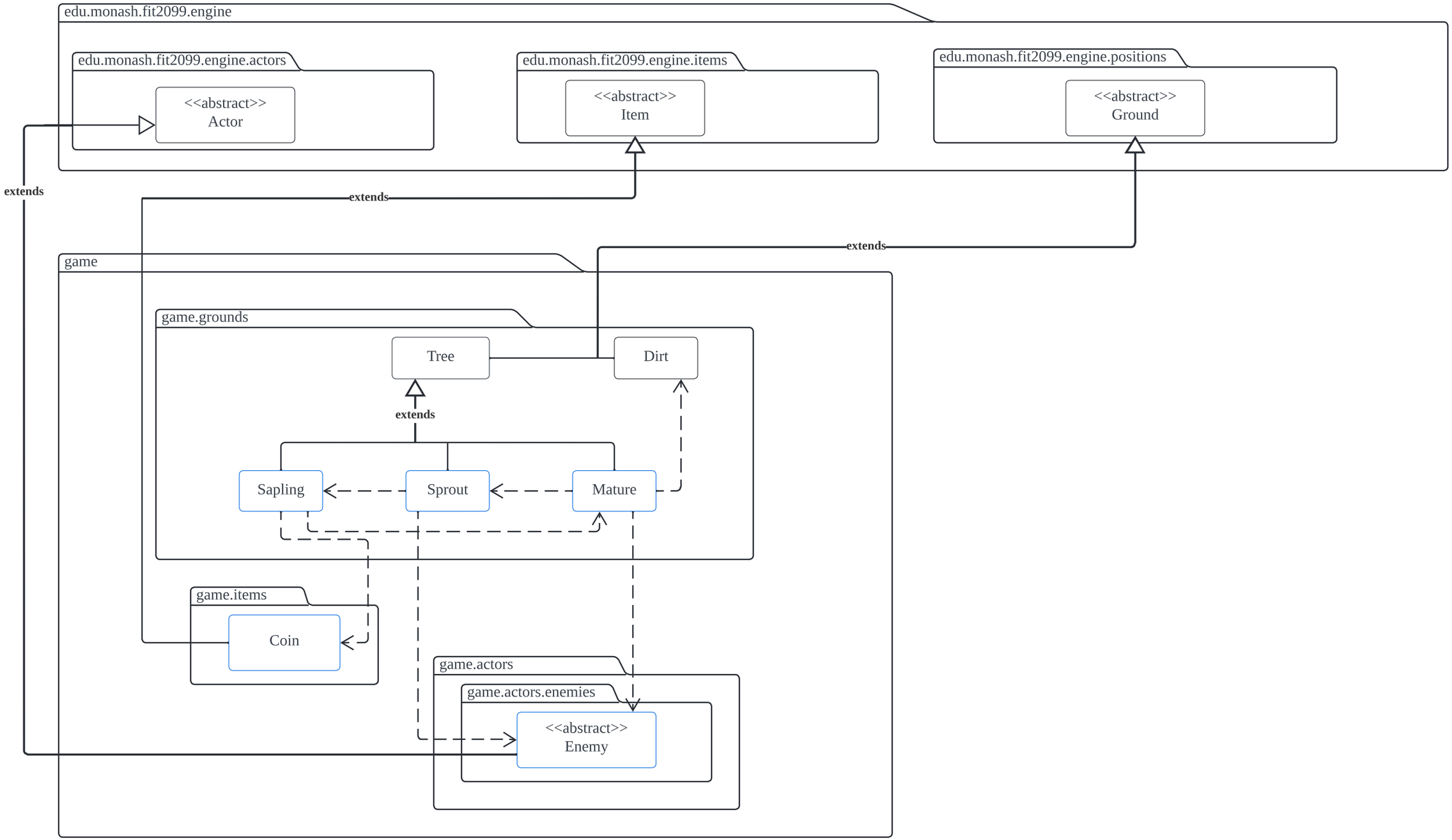
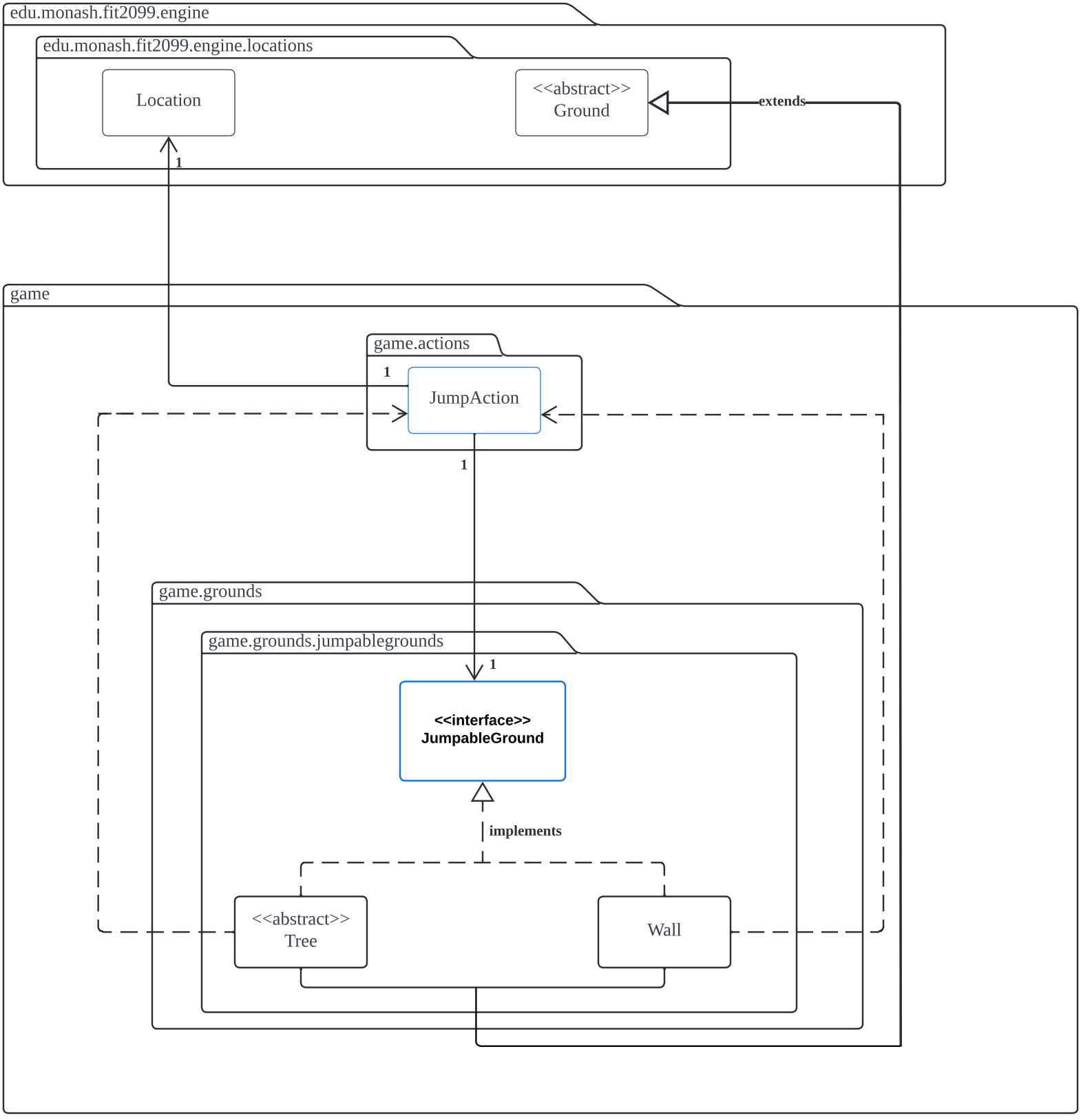
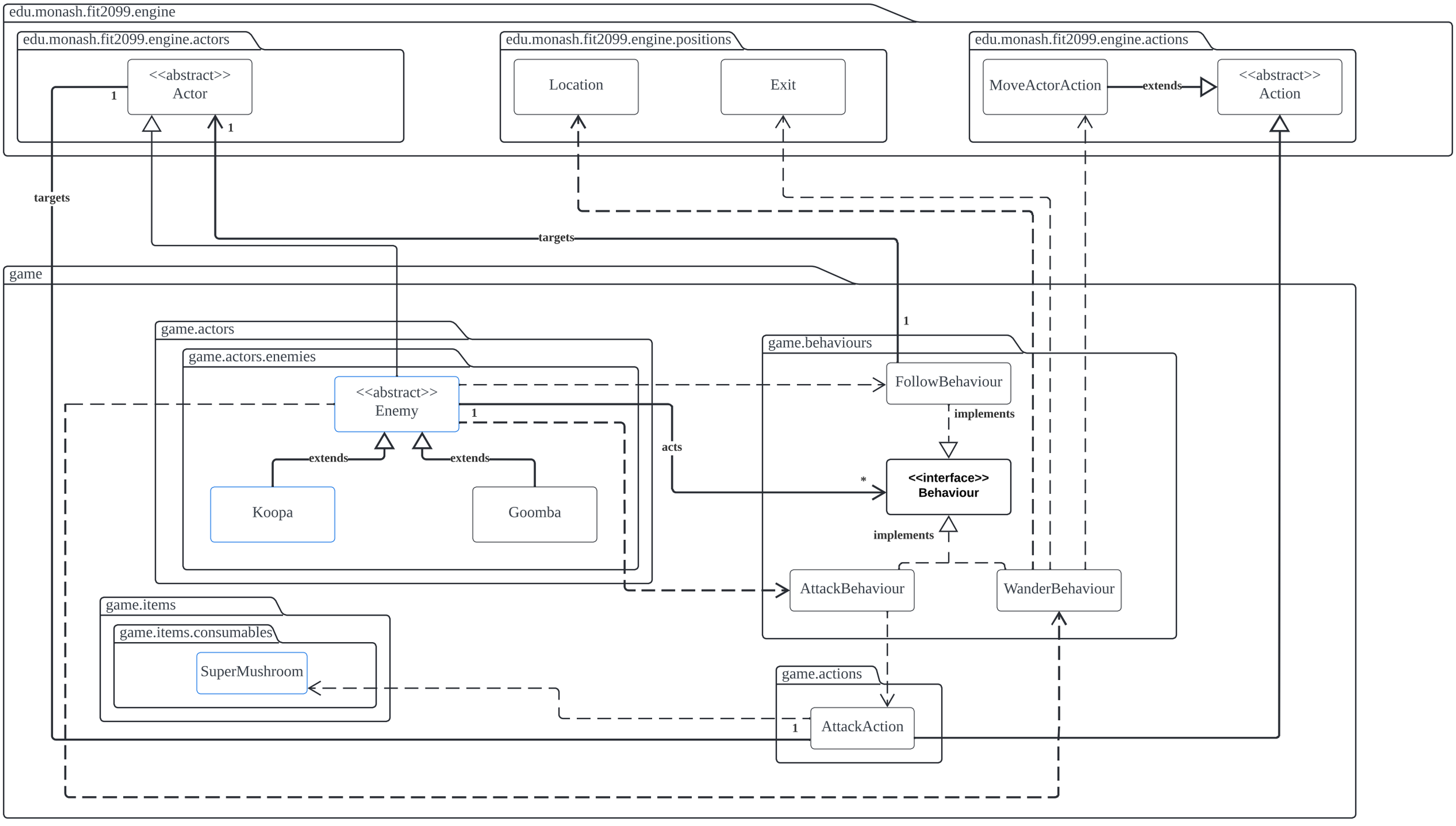
Req1


edu.monash.fit2099.engine
edu.monash.fit2099.engine.actors
<<abstract>>
Actor
edu.monash.fit2099.engine.items
<<abstract>>
Item
edu.monash.fit2099.engine.positions
<<abstract>>
Ground
extends
extends
extends
game
game.grounds
Tree
Dirt
extends
Sapling
Sprout
Mature
game.items
Coin
game.actors
game.actors.enemies
<<abstract>>
Enemy

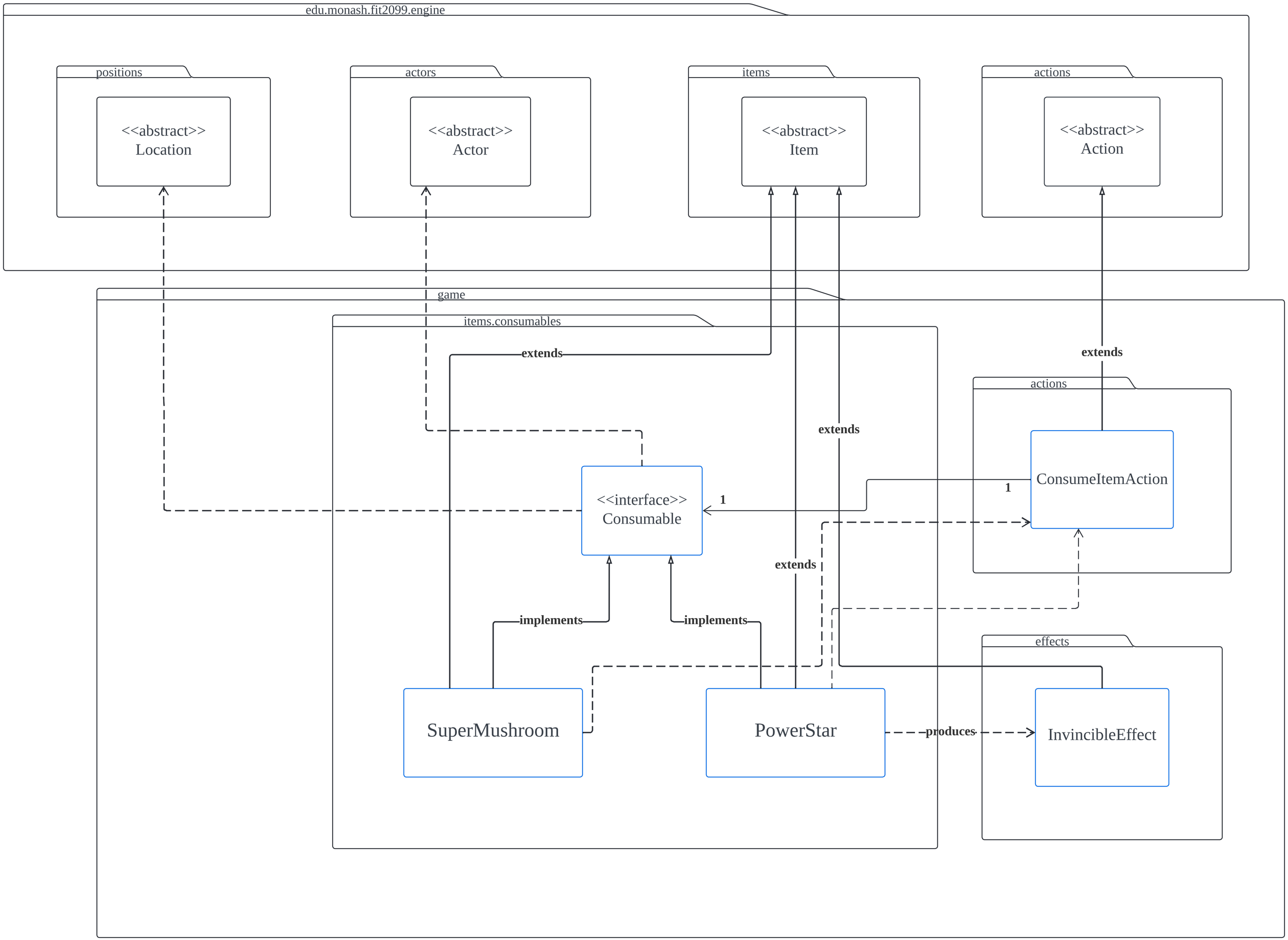
Req2


edu.monash.fit2099.engine
edu.monash.fit2099.engine.locations
Location
<<abstract>>
Ground
extends
1
game
game.actions
1
JumpAction
1
game.grounds
game.grounds.jumpablegrounds
1
<<interface>>
JumpableGround
implements
<<abstract>>
Tree
Wall

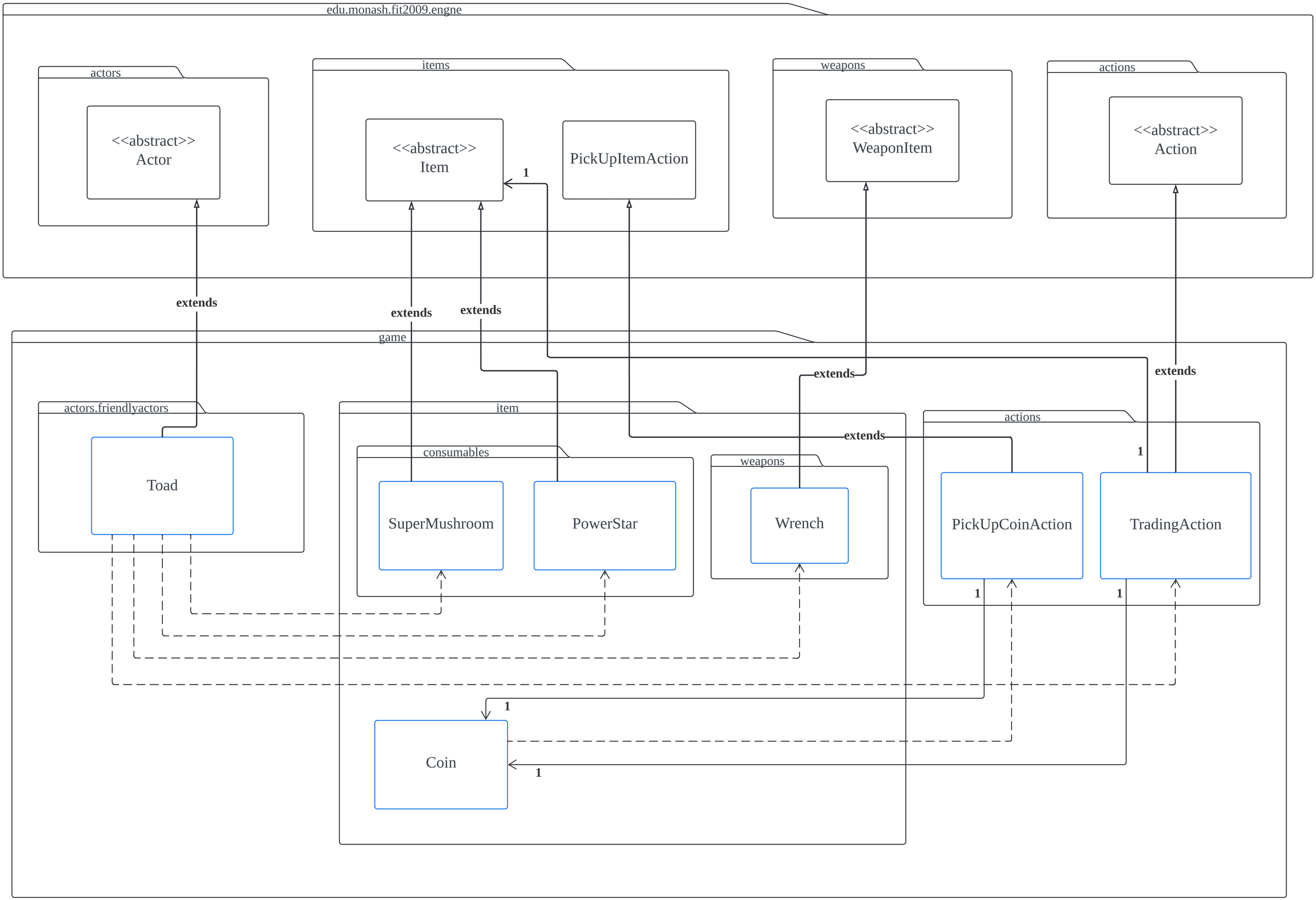
Req3


edu.monash.fit2099.engine
edu.monash.fit2099.engine.actors
<<abstract>>
Actor
edu.monash.fit2099.engine.positions
Location
Exit
edu.monash.fit2099.engine.actions
MoveActorAction
extends
<<abstract>>
Action
1
1
targets
targets
game
game.actors
game.actors.enemies
<<abstract>>
Enemy
1
extends
extends
Koopa
Goomba
acts
1
game.behaviours
FollowBehaviour
implements
*
<<interface>>
Behaviour
implements
AttackBehaviour
WanderBehaviour
game.items
game.items.consumables
SuperMushroom
game.actions
1
AttackAction

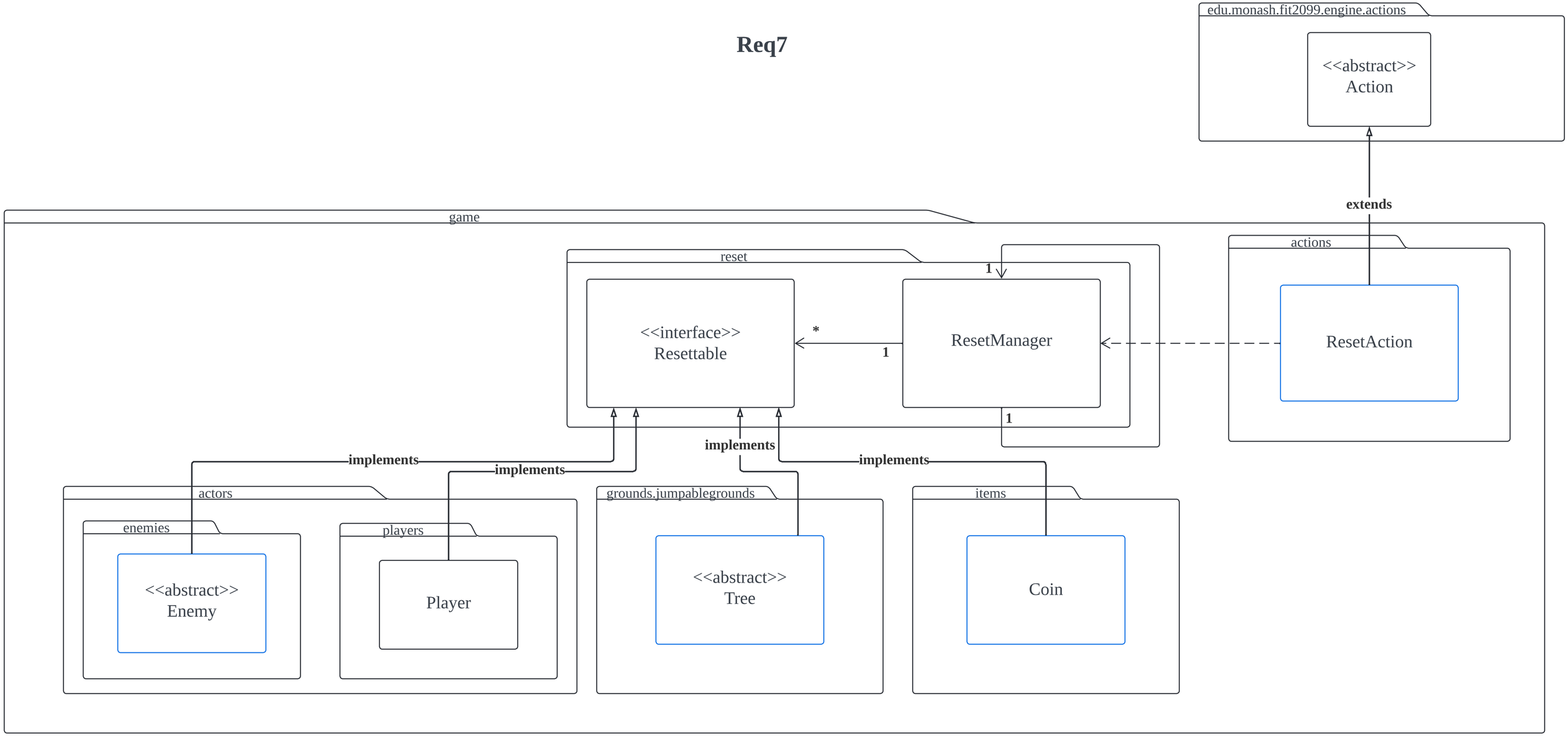
Req4


edu.monash.fit2099.engine
positions
<<abstract>>
Location
actors
<<abstract>>
Actor
items
<<abstract>>
Item
actions
<<abstract>>
Action
game
items.consumables
extends
extends
extends
actions
ConsumeItemAction
1
1
<<interface>>
Consumable
implements
implements
SuperMushroom
PowerStar
produces
effects
InvincibleEffect

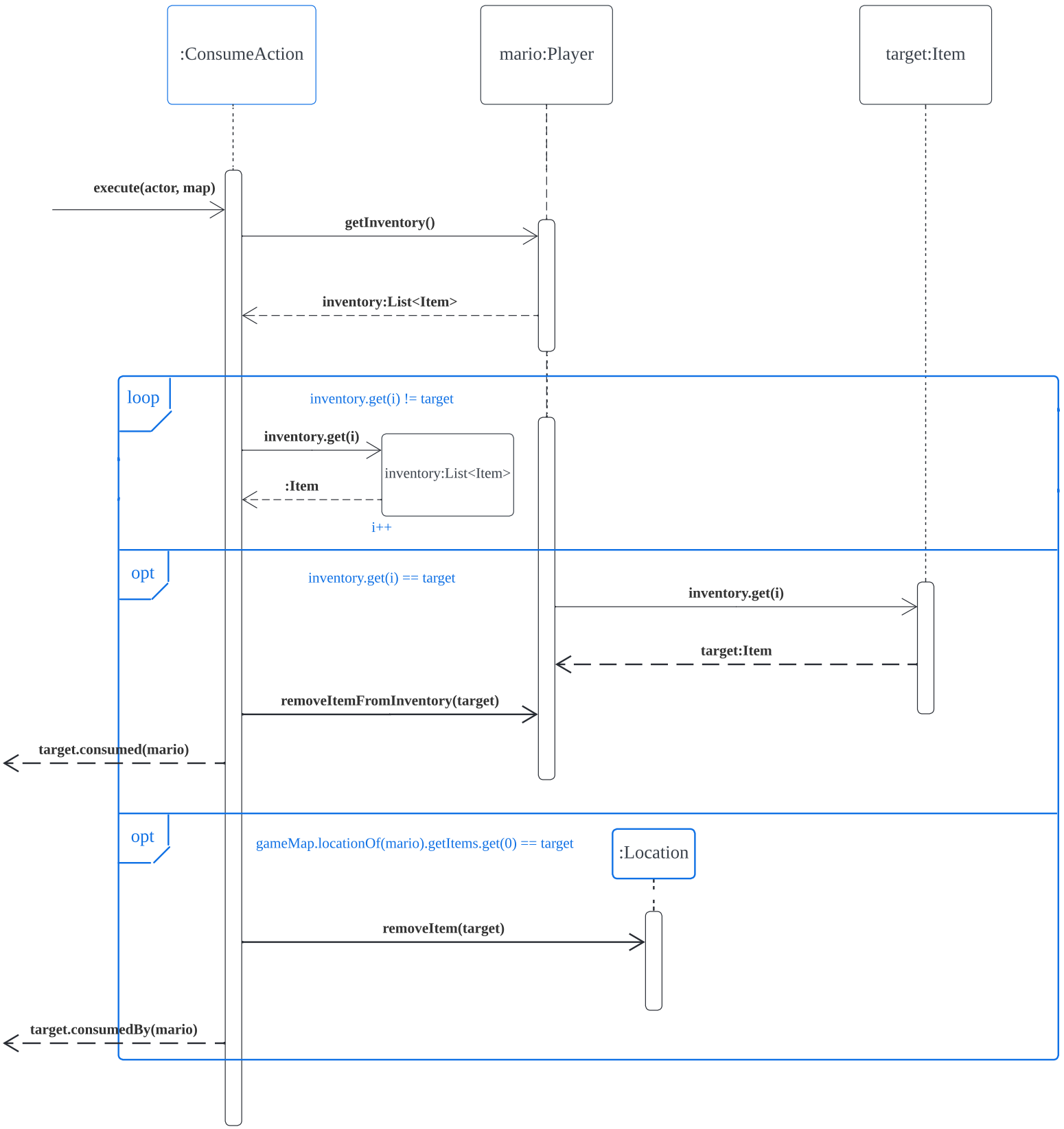
Req5


edu.monash.fit2009.engne
actors
<<abstract>>
Actor
items
<<abstract>>
Item
PickUpItemAction
weapons
<<abstract>>
WeaponItem
actions
<<abstract>>
Action
1
extends
extends
extends
extends
extends
game
actors.friendlyactors
Toad
item
consumables
SuperMushroom
PowerStar
weapons
Wrench
actions
PickUpCoinAction
TradingAction
1
1
1
1
1
Coin

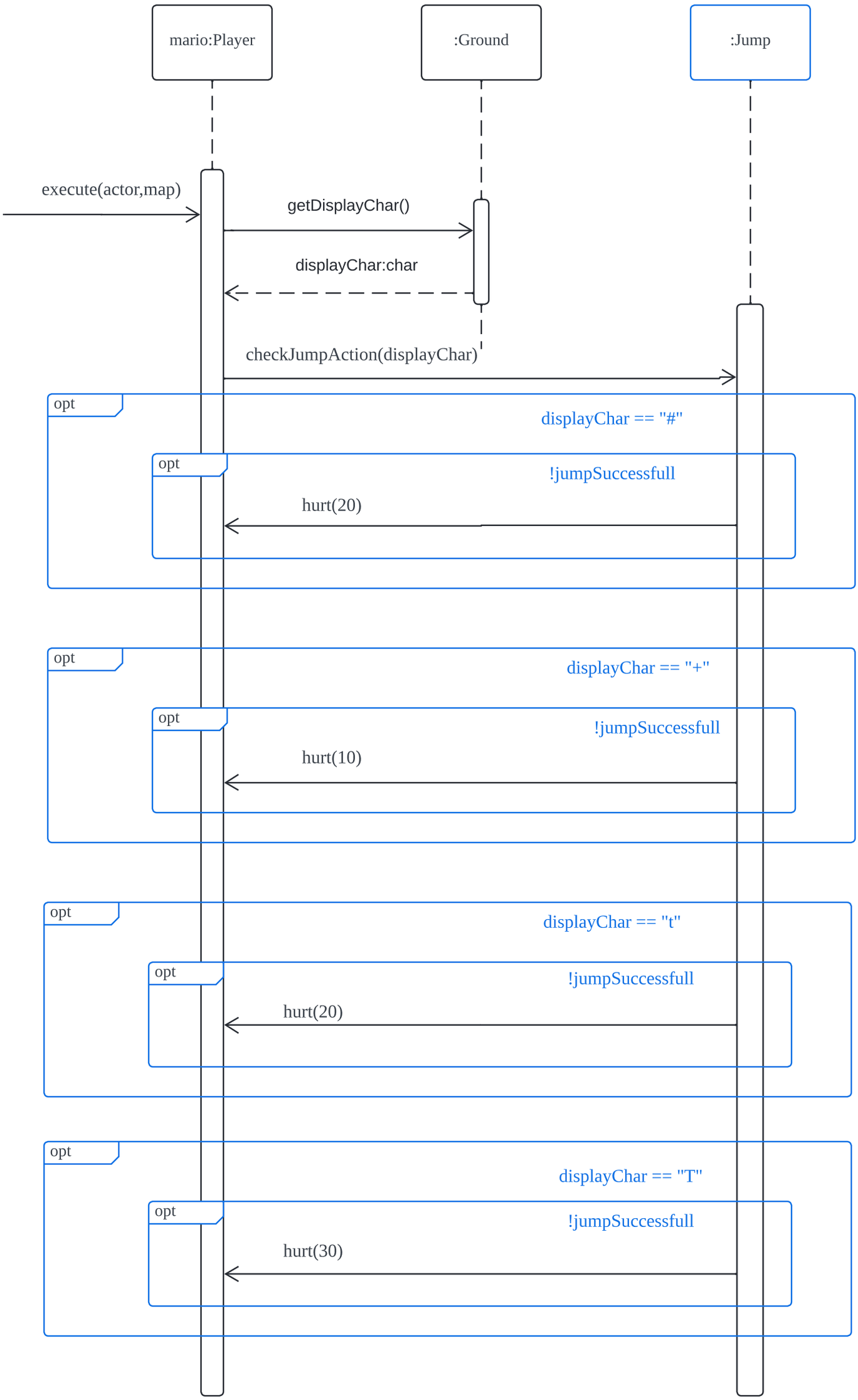
Req7


edu.monash.fit2099.engine.actions
<<abstract>>
Action
extends
game
reset
<<interface>>
Resettable
*
1
ResetManager
1
1
actions
ResetAction
implements
implements
implements
implements
actors
enemies
<<abstract>>
Enemy
players
Player
grounds.jumpablegrounds
<<abstract>>
Tree
items
Coin

Consume - Interaction Diagram



Jump - Interaction Diagram



FIT2009 Assignment

Design Rationale

Lab 11 Team 4

Teammate:
Subhan Saadat Khan - 32268513
Mingze Zheng - 32433786

REQ1

As mentioned in the requirement the Tree has 3 stages Sprout, Sapling and Mature. This indicates that these stages will act as child/base classes where the parent is Tree class.

Thus, following the SRP Principle that each class should have one responsibility, I have created 3 new classes denoting each stage of the tree.These classes will only be responsible for the features and functionalities that any stage possesses.

By creating 3 new classes I have also fulfilled the requirement of OCP Principle as any new method for any Tree stage can be added in the respective stage class easily without changing the way the existing code is used.

These new classes inherit the Tree class as the fundamental behaviour of a tree remains the same at any stage.

Furthermore, the Sprout and Mature Class have dependency with Abstract Class Enemy as stated in the requirement Sprout has a chance to spawn Goomba and Mature has a chance to spawn Koopa. This way I followed the LSP Principle as I used instances of Goomba & Koopa (subclass of Enemy class) where code was expecting the instance of Enemy class (base class).

The Sapling Class has a dependency with the new class Coin as stated in the requirement that sapling has a chance to drop a coin.
The Sprout,Sapling and Mature class have dependencies among themselves as it was stated in the requirement that Sapling grows from Sprout, Mature grows from Sapling and Mature grows back Sprout.
Finally the Mature Class has a dependency with the Dirt Class as a Mature has a chance to die and become Dirt.

REQ2

This requirement gives us an opportunity to use abstraction since each jumpable ground, i.e. Tree and Wall, has a unique implementation of being jumped by the player. In this case, I decided to implement the JumpableGround abstraction with an interface. Following the DIP Principle I have kept the interface small and concise as all classes that implement the interface will only need to provide a concrete implementation for a single method.

I created a JumpAction class that extends Action to represent all interactions or movements that the player should be able to perform while jumping in the game. It extends Action class as it is responsible for handling all actions that a player performs.

The JumpAction Class has an association relation with JumpableGround as it needs to know which jumpable ground will be jumped in this action. By doing this I have followed the LSP Principle as I used instances of Tree & Wall (subclass of JumpableGround class) where code was expecting the instance of JumpableGround class (base class). Moreover, it has an association relation with the Location Class of the engine as it needs to know the Location of the jumpable ground as it is required by the *jumped* method of the JumpableGround interface.

As Tree and Wall are the grounds that need to be jumped by the Player thus these classes will implement the JumpableGround interface to override the *jumped* method. Using this I have followed the DIP Principle as complex classes depend on interface. Furthermore, Tree and Wall also has dependency relation with JumpAction as they will create a new Jump Action in the *allowableActions* method.

REQ3

In this requirement we were supposed to implement two enemies. Since all enemies have moreover the same fundamental attributes, thus I created a new Abstract class called Enemy. The Enemy class will serve as a base class for common attributes of enemies.

Thus, following the SRP Principle that each class should only have one responsibility, I have created 2 new classes denoting each enemy Goomba & Koopa.These classes will only be responsible for the features and functionalities possessed by the particular enemy.

By creating 2 classes I have also fulfilled the requirement of OCP Principle as any new method for any enemy can be added in the respective enemy class easily without changing the way the existing code is used.

As Goomba and Koopa are enemies of Player thus they inherit from the Enemy abstract class. Since we know an enemy is also an actor thus the Abstract class Enemy will inherit the Abstract class Actor.

Moving on, as stated in the requirement that an enemy follows a certain set of behaviours. Since all enemies follow the same set of behaviours, therefore to minimise redundancy and repetition of code I decided to implement Behaviours inside Enemy abstract class (base class) rather than inside Goomba & Koopa individually as that would have violated the DRY Principle.

There is an association relation between Enemy abstract class and Interface Behaviour as Enemy has an attribute that stores Behaviour objects inside a TreeMap. By doing this I have followed the LSP Principle as I stored instances of FollowBehaviour,WanderBehaviour & AttackBehaviour (subclass of Behaviour interface) into the TreeMap where code was expecting the instance of Behaviour interface (base class).

Enemy abstract class has dependency with FollowBehaviour,WanderBehaviour & AttackBehaviour because by default whenever an enemy is created it will possess Attack and Wander behaviour, it will possess follow behaviour once enemy is engaged in a fight.

The AttackBehaviour creates a new AttackAction when Enemy decides to attack thus having a dependency relation. Furthermore, AttackAction creates a super mushroom when Player breaks Koopa's thus having a dependency relation.

REQ4

For the design of magical items, from the definition and description of super mushroom and power star, it is obvious that we need to create two new classes that inherit the Item class. Therefore, we create the two classes SuperMushroom and PowerStar and make them inherit class Item. In these two classes, we will implement the details separately to follow the SRP Principle.

To consume magical items, we need a new action. Obviously, the new action is a subclass of the abstract class Action. We create a new class called ConsumeItemAction which inherits Action. To improve the expansibility of the design. We can create a new interface called Consumable. Any class that implements this interface will be able to be the target of ConsumeItemAction. This reflects the advantage of the DIP Principle.

In ConsumeItemAction, we need to use the item that will be consumed and the consumer of the item. As some other actions do, I choose to declare Item as an instance variable in ConsumeItemAction. And the consumer will be a parameter of the overridden method execute(). To make ConsumeAction can be selected by the player, we need to override the getAllowableActions method in the two classes.

From class Item, we can see that the attributes and methods in the Item class contain the common attributes and methods that SuperMushroom and PowerStar need. From the functionalities of SuperMush and PowerStar, we may need to add new statuses to Enum Status to make the implementation easier. For SuperMushroom, we will add TALL as the status after the player consumes a SuperMushroom. And we will add another status called INVINCIBLE for the player who consumed PowerStar. To make sure the player can get INVINCIBLE status after consuming PowerStar and PowerStar will not affect the player's status, we need to create a new class InvincibleEffect to do that. This follows the SRP Principle to make sure the program will run as we imagine.

REQ5

As we can see from the description, a coin is also a type of item. It has the attributes that an item should have. So we make the new class Coin a subclass of Item. One thing we need to notice is that when we pick up coins from the map, coins should be added to the wallet instead of many Coin objects in the inventory. So we can create a new action called PickUpCoinAction to do this. It is a subclass of PickUPItemAction.
And we need to provide methods to increase the worth of coins and override Coin's PickUpItemAction to fulfil our requirement. The design above follows the LSP Principle.

"Toad is a friendly actor", so Toad should inherit the Actor class from the description. We need to purchase items from the toad, as we can imagine, which is also an action. So we need to create a new class called TradingAction (it is also a subclass of action since trading is a type of action). As other actions do, we will make Item an instance variable of TradingAction, and buyer (Player) will be a parameter of the

method execute(). So the relationship between TradingAction and Item is association and the relationship between TradingAction and Player is dependency. To make sure the player can select TradingAction when a Toad is surrounding, we need to override allowableActions in Toad. And inside allowableActions we will add three TradingActions to sell Wrench, SuperMushroom, and PowerStar. This design follows the OCP Principle.

REQ7

If we want to reset the game, from the base code and the description, the best design is to make the classes that need to be reset implement the interface Resettable. So in our design, abstract class Enemy, class Player, abstract class Tree, and class Coin will implement Resettable. This follows the DIP Principle.

Then we will need to finish the methods in class ResetManager. In the ResetManager, there is a list that is used to keep track of objects implementing Resettable. Objects that implement Resettable will be added to the List after initialising. To use ResetManager, we need to reset an action that can be selected by the player. Therefore, we will create a new action called ResetAction. In the ResetAction class, we will get the instance of ResetManager to control the reset. This reduces the coupling between ResetManager and ResetAction, which follows the OCP principle.

Work Breakdown Agreement for FIT2099 Assignment 1

Lab 11 Team 4:

- Subhan Saadat Khan - 32268513
- Mingze Zheng - 32433786

We will separate the Class Diagram into 2 parts:

1. REQ1: Let it grow! , REQ2: Jump Up, Super Star! , REQ3: Enemies
2. REQ4: Magical Items, REQ5: Trading, REQ7: Reset Game

Subhan Saadat Khan will be responsible for Part 1 and its design rationale, Reviewer: Mingze Zheng, Completion: Saturday 9th April.

Mingze Zheng will be responsible for Part 2 and its design rationale, Reviewer: Subhan Saadat Khan, Completion: Saturday 9th April.

We will separate the Interaction diagrams into 2 parts:

1. REQ1: Let it grow! , REQ2: Jump Up, Super Star! , REQ3: Enemies
2. REQ4: Magical Items, REQ5: Trading, REQ7: Reset Game

Subhan Saadat Khan will be responsible for Part 1, Reviewer: Mingze Zheng, Completion: Saturday 9th April.

Mingze Zheng will be responsible for Part 2, Reviewer: Subhan Saadat Khan, Completion: Saturday 9th April.

Signed by (type “I accept this WBA”)

I accept this WBA - Subhan Saadat Khan
I accept this WBA – Mingze Zheng