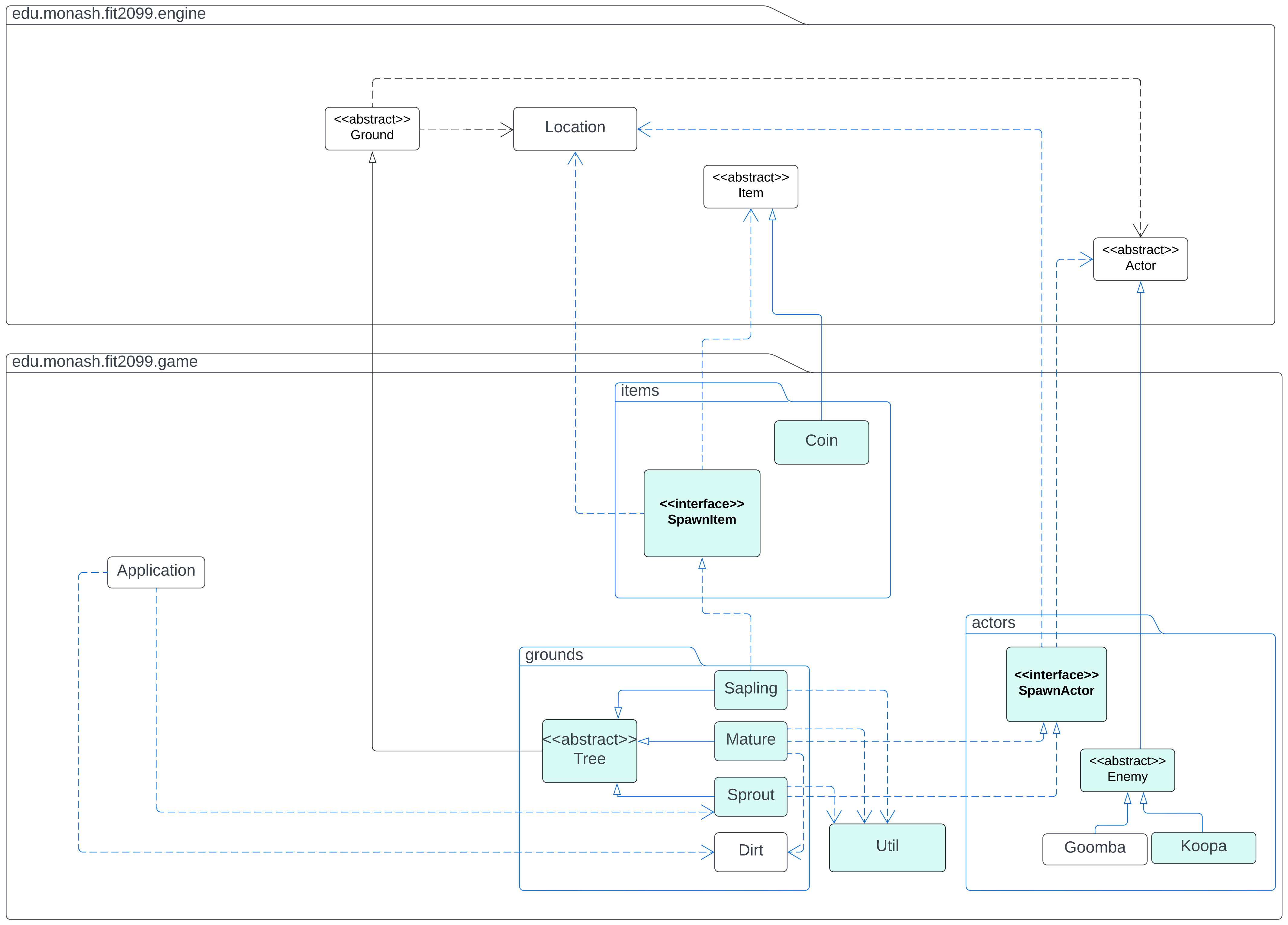
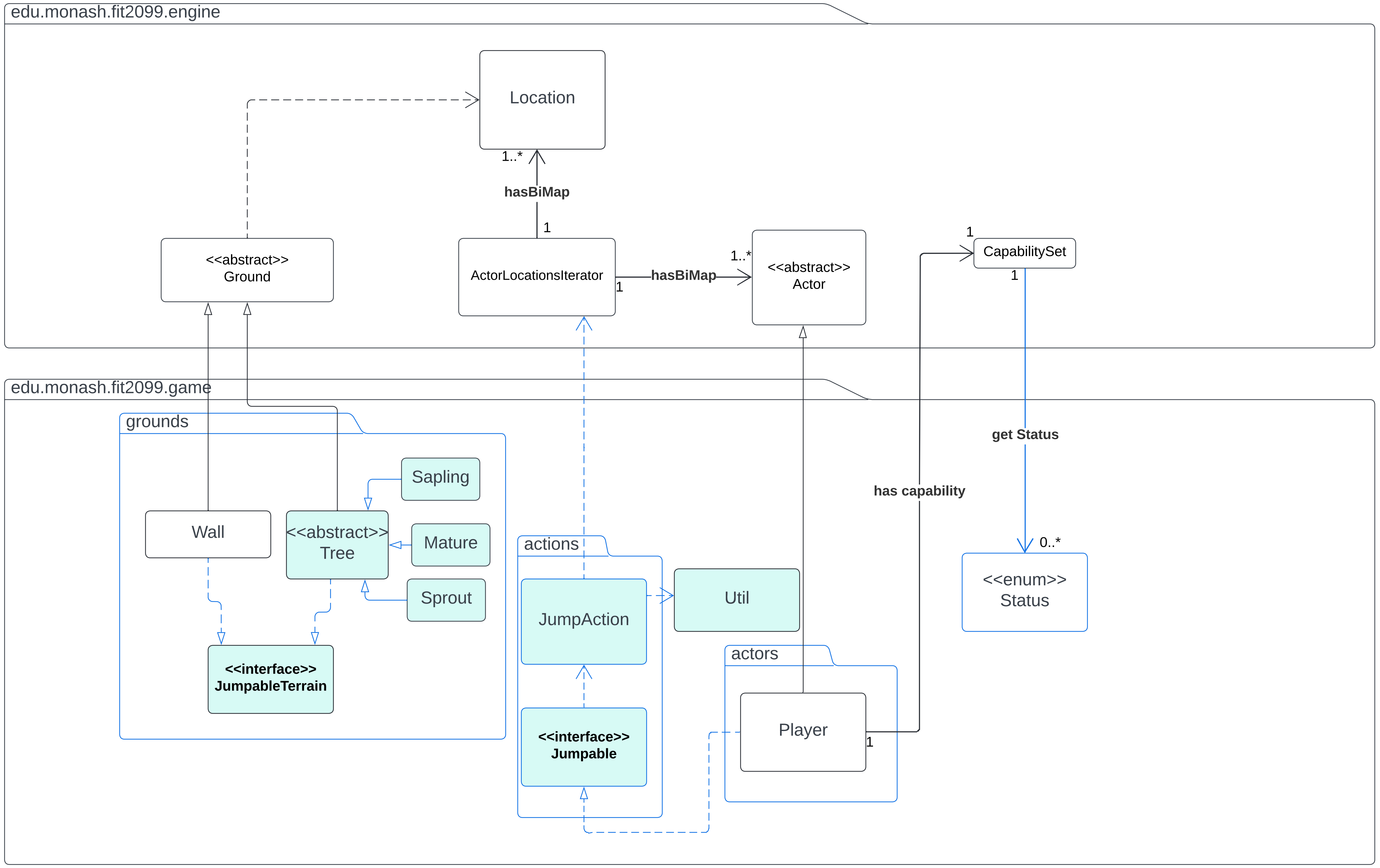
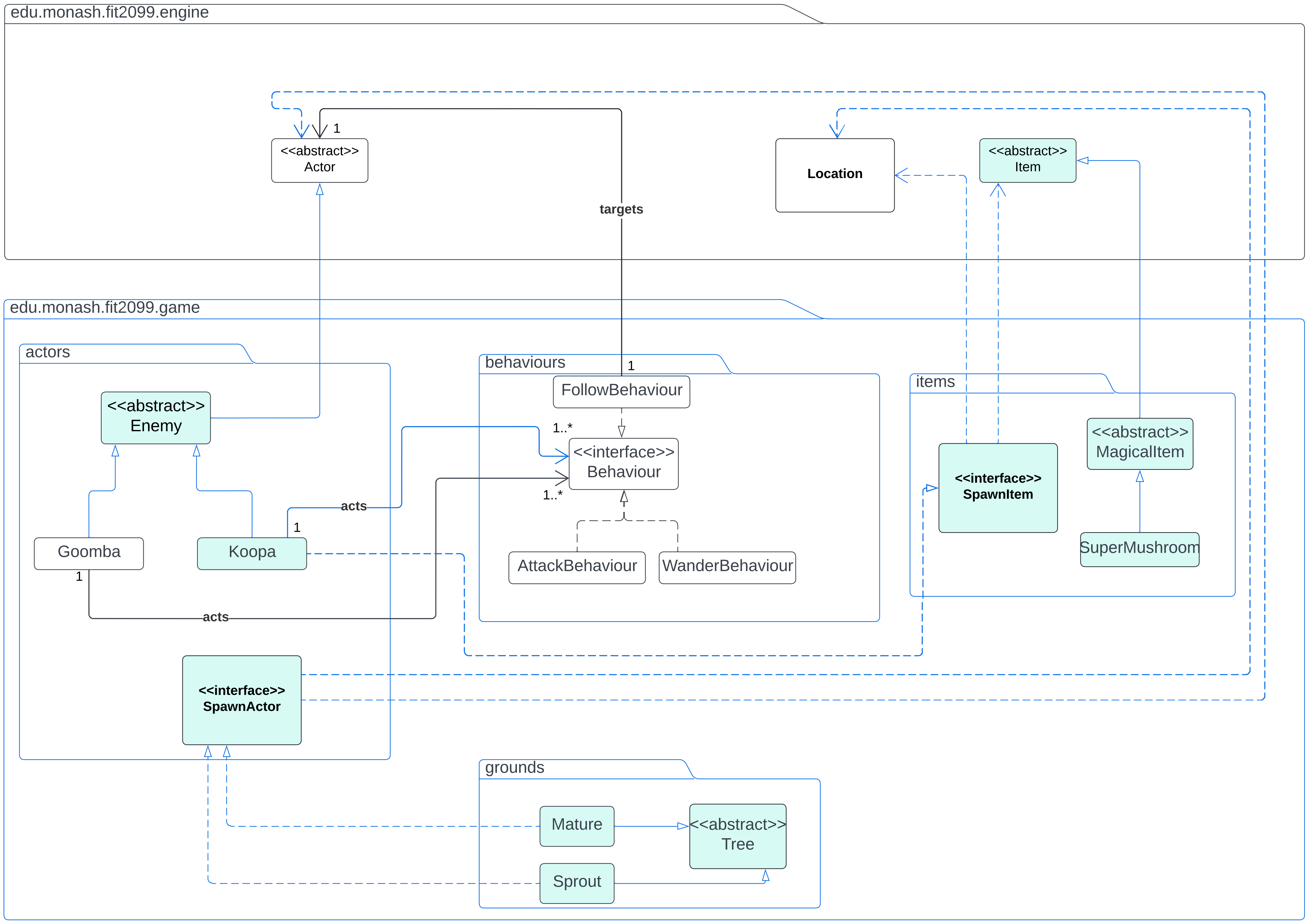
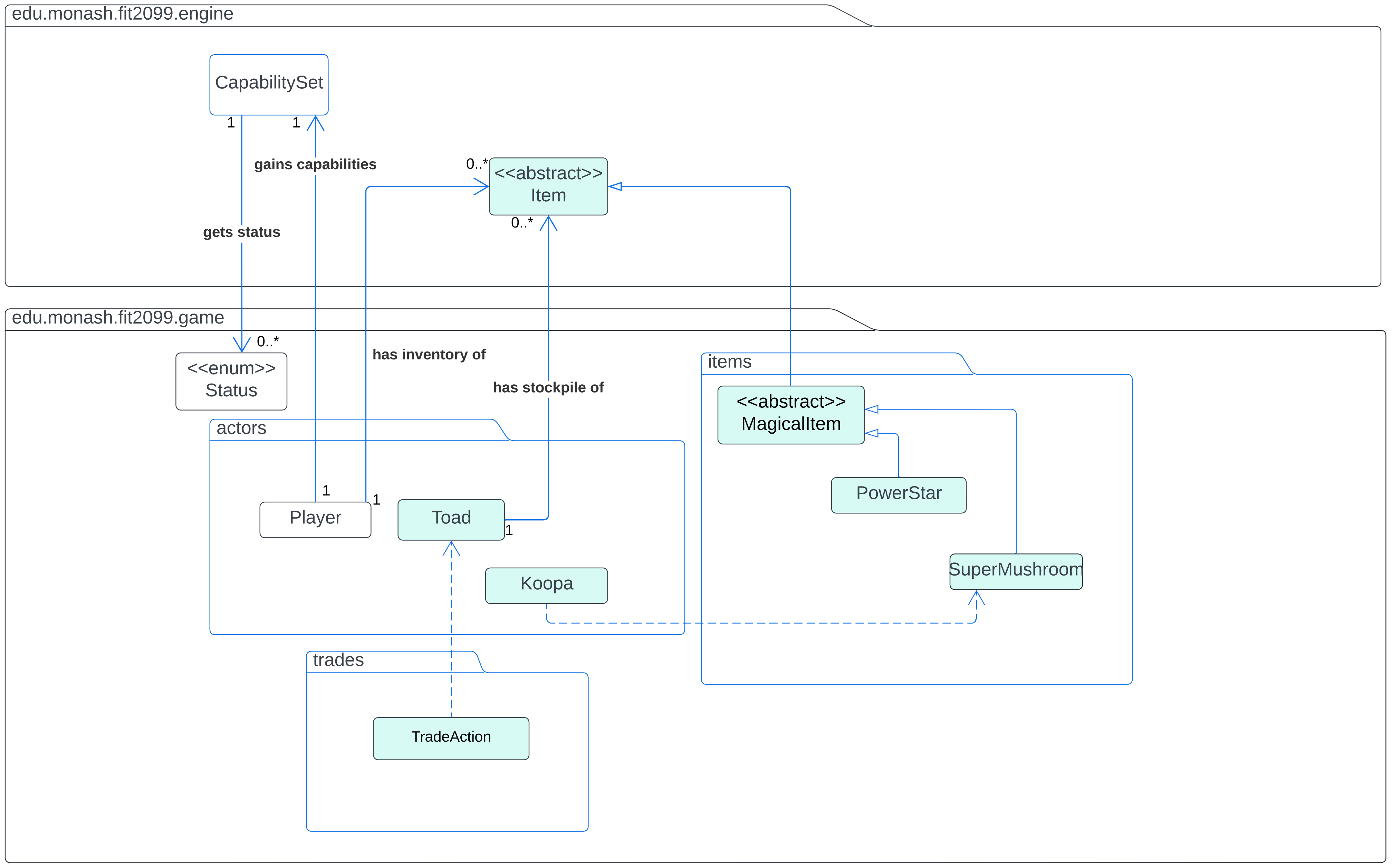
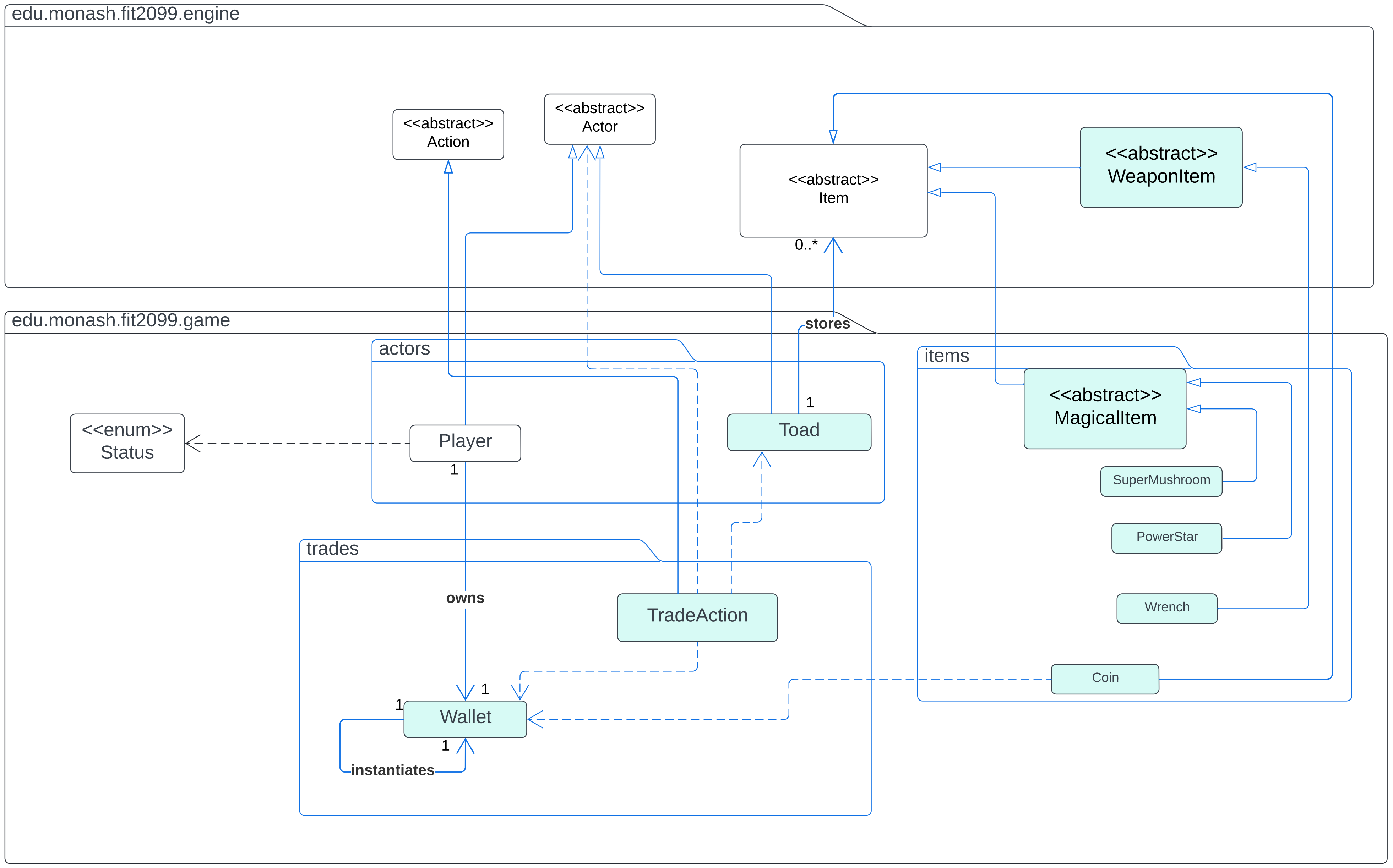
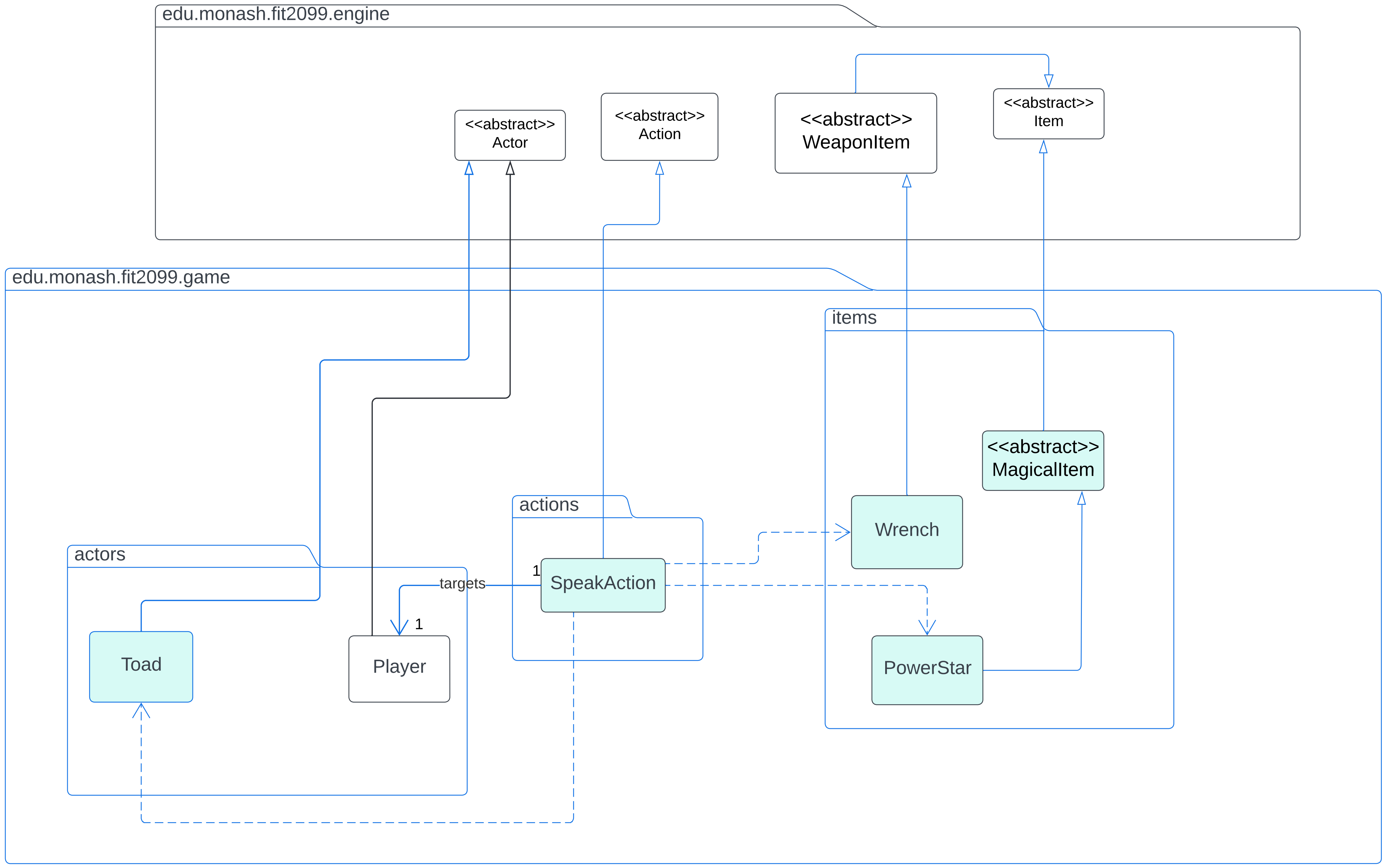

edu.monash.fit2099.engine
<<abstract>> Ground
Location
<<abstract>> Item
<<abstract>> Actor
edu.monash.fit2099.game
items
Coin
<<interface>> SpawnItem
Application
grounds
Sapling
<<abstract>> Tree
Mature
Sprout
Dirt
Util
actors
<<interface>> SpawnActor
<<abstract>> Enemy
Goomba
Koopa

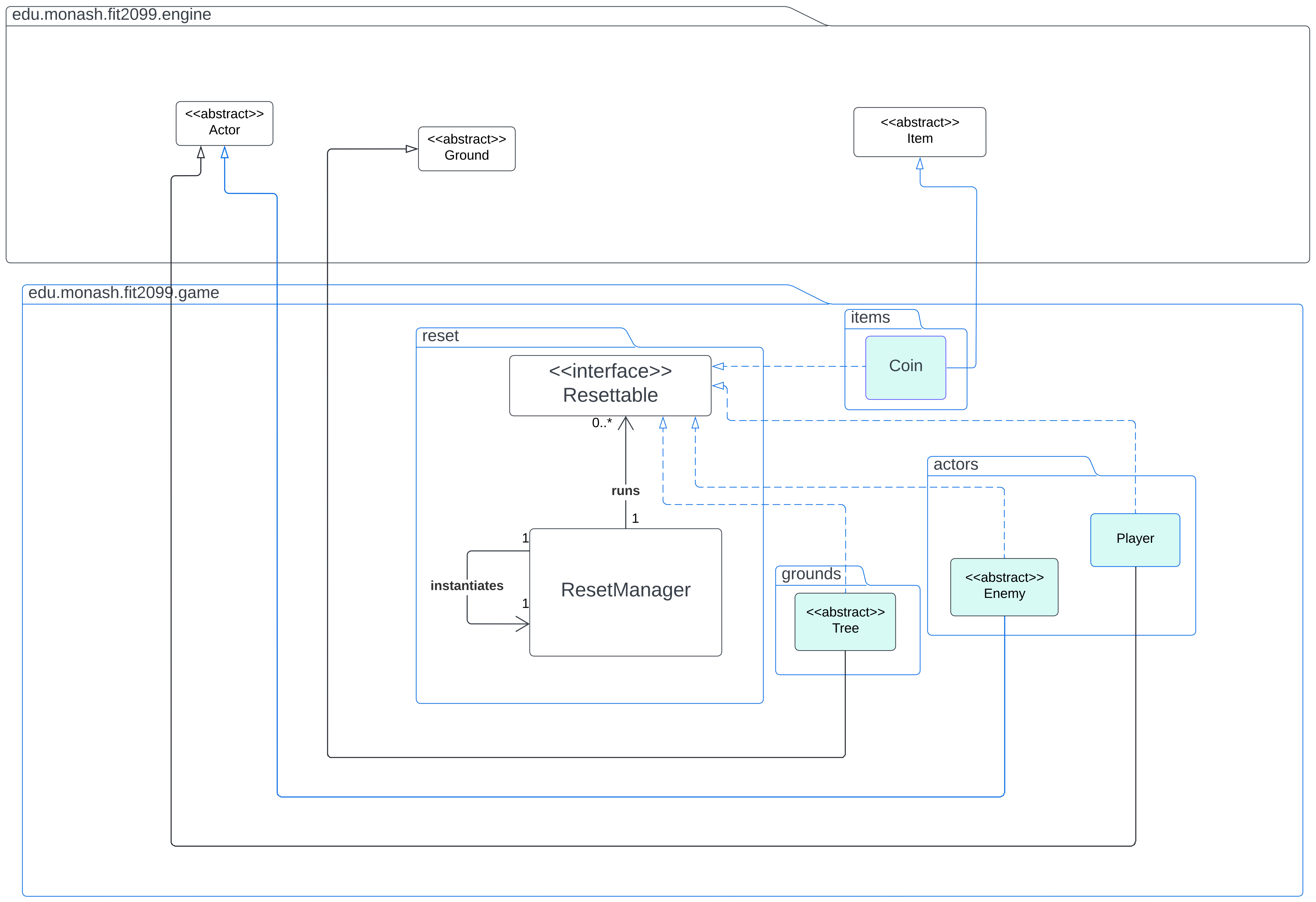

edu.monash.fit2099.engine
Location
1..*
hasBiMap
1
<<abstract>> Ground
ActorLocationsIterator
1
hasBiMap
1..*
<<abstract>> Actor
1
CapabilitySet
1
get Status
0..*
<<enum>> Status
edu.monash.fit2099.game
grounds
Sapling
Wall
<<abstract>> Tree
Mature
Sprout
<<interface>> JumpableTerrain
actions
JumpAction
<<interface>> Jumpable
Util
actors
Player
1
has capability

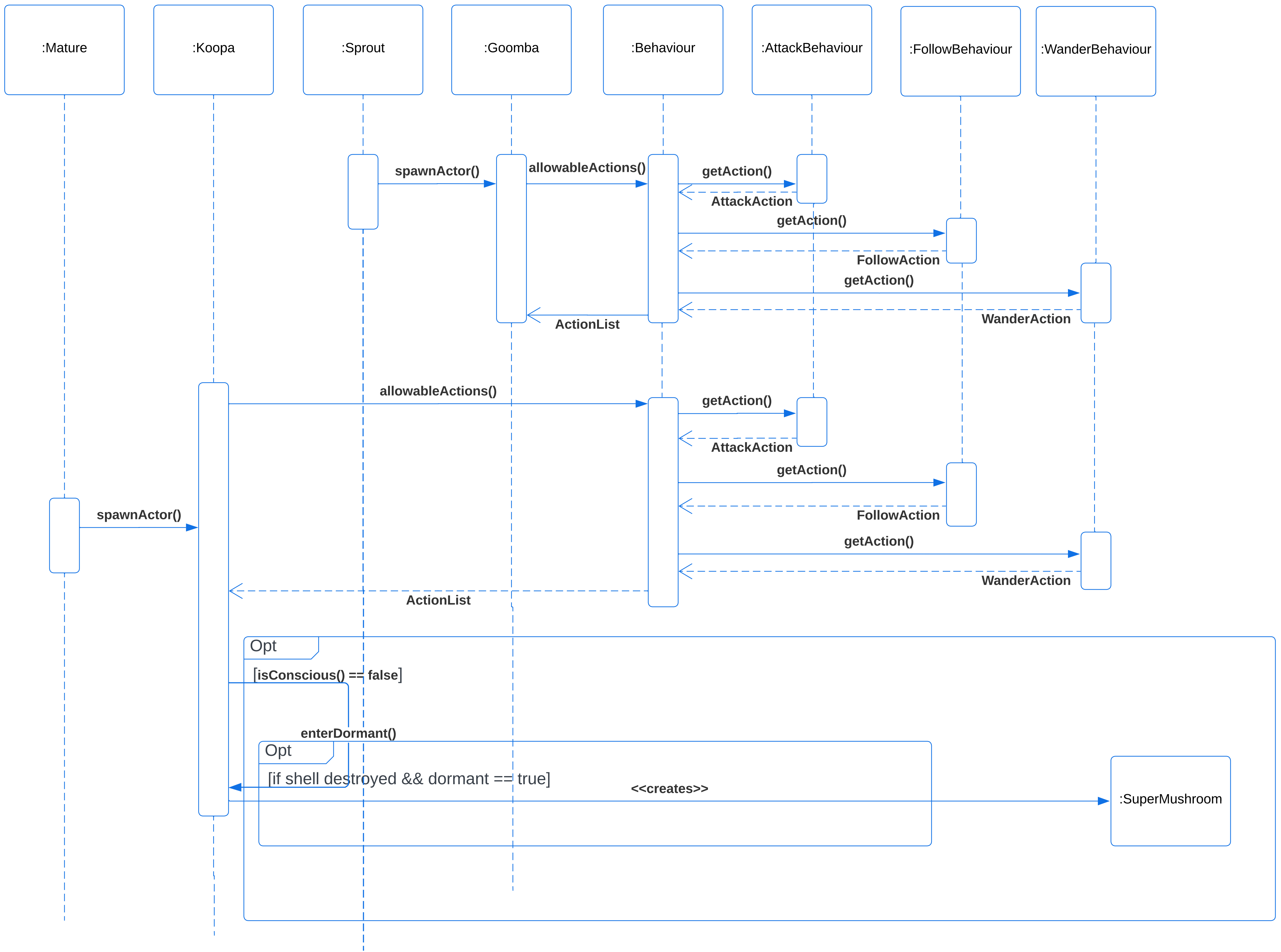

edu.monash.fit2099.engine
<<abstract>> Actor
1
targets
Location
<<abstract>> Item
edu.monash.fit2099.game
actors
<<abstract>> Enemy
Goomba
Koopa
1
acts
1
acts
<<interface>> SpawnActor
behaviours
1
FollowBehaviour
1..*
<<interface>> Behaviour
1..*
AttackBehaviour
WanderBehaviour
items
<<interface>> SpawnItem
<<abstract>> MagicalItem
SuperMushroom
grounds
Mature
<<abstract>> Tree
Sprout

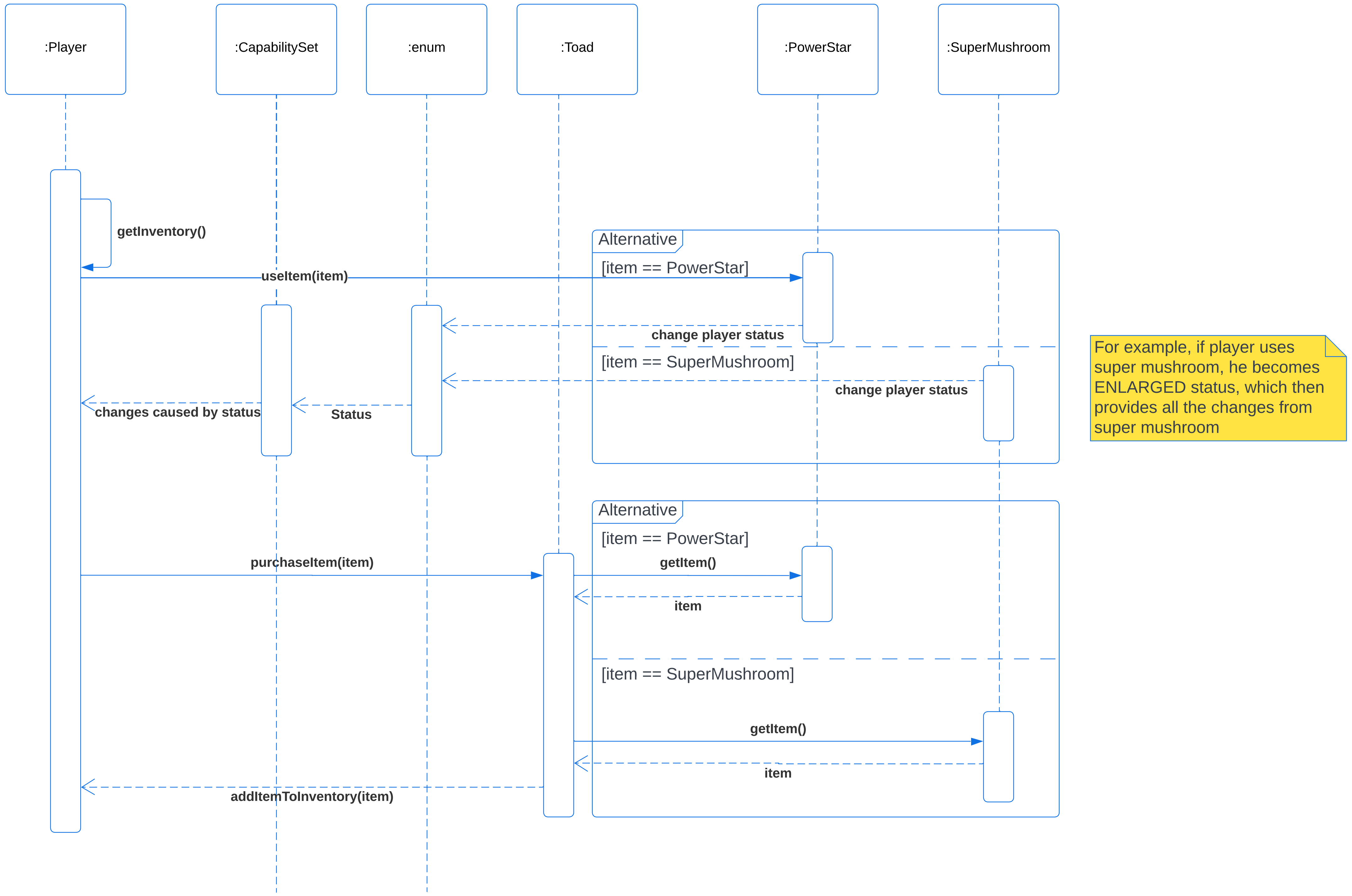

edu.monash.fit2099.engine
CapabilitySet
1
1
gains capabilities
0..*
<<abstract>> Item
0..*
gets status
edu.monash.fit2099.game
0..*
<<enum>> Status
has inventory of
has stockpile of
items
<<abstract>> MagicalItem
actors
PowerStar
1
1
Player
Toad
1
SuperMushroom
Koopa
trades
TradeAction


edu.monash.fit2099.engine
<<abstract>> Action
<<abstract>> Actor
<<abstract>> Item
0..*
<<abstract>> WeaponItem
edu.monash.fit2099.game
stores
actors
Player
1
Toad
1
<<enum>> Status
items
<<abstract>> MagicalItem
SuperMushroom
PowerStar
Wrench
Coin
trades
owns
TradeAction
Wallet
1
1
1
instantiates

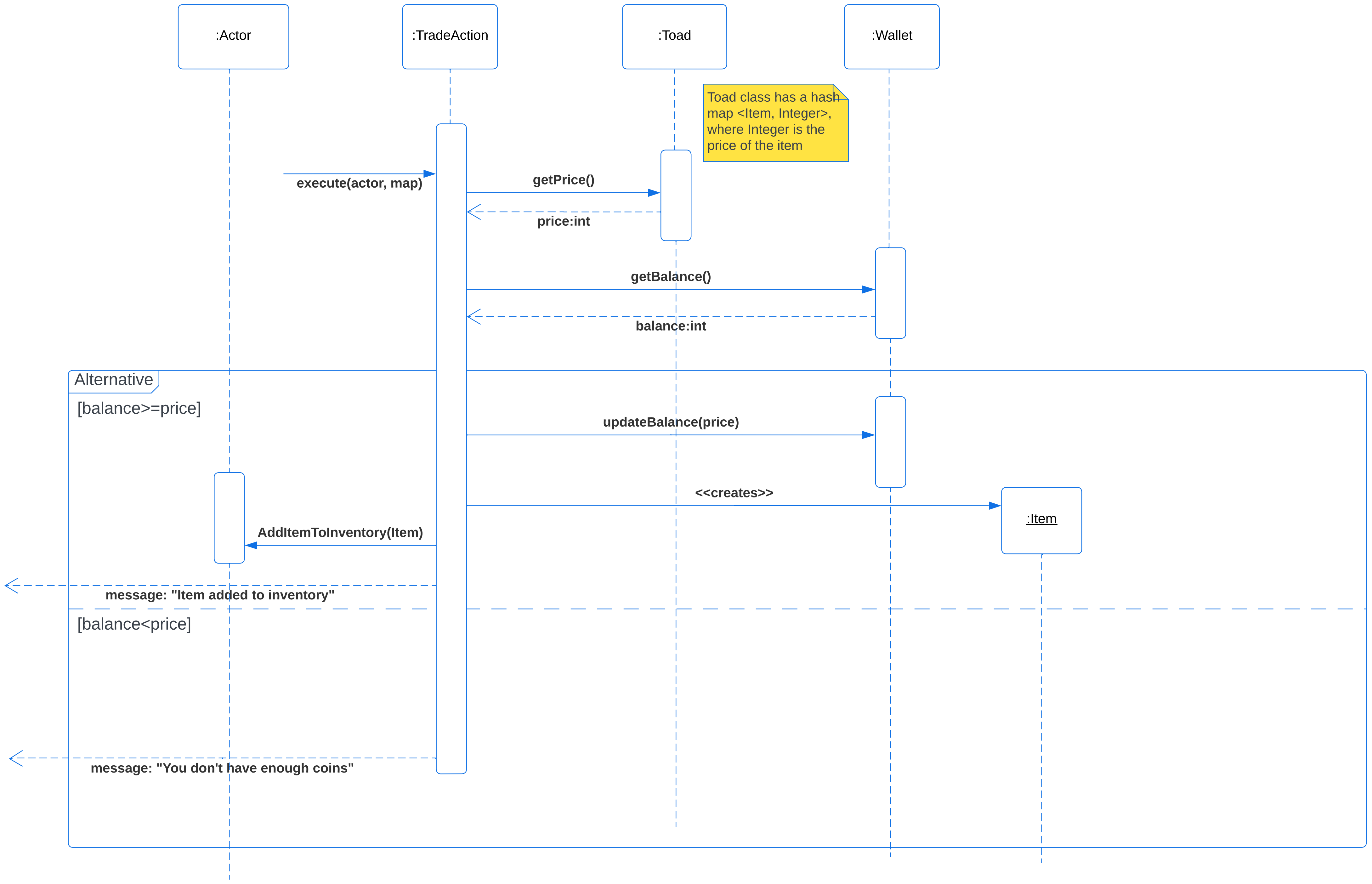

edu.monash.fit2099.engine
<<abstract>> Actor
<<abstract>> Action
<<abstract>> WeaponItem
<<abstract>> Item
edu.monash.fit2099.game
items
<<abstract>> MagicalItem
Wrench
PowerStar
actions
SpeakAction
actors
Toad
Player
targets
1
1


edu.monash.fit2099.engine
<<abstract>> Actor
<<abstract>> Ground
<<abstract>> Item
edu.monash.fit2099.game
reset
<<interface>> Resettable
0..*
runs
1
1
instantiates
1
ResetManager
items
Coin
actors
Player
<<abstract>> Enemy
grounds
<<abstract>> Tree


:Mature
:Koopa
:Sprout
:Goomba
:Behaviour
:AttackBehaviour
:FollowBehaviour
:WanderBehaviour
spawnActor()
allowableActions()
getAction()
AttackAction
getAction()
FollowAction
getAction()
WanderAction
ActionList
allowableActions()
getAction()
AttackAction
getAction()
FollowAction
spawnActor()
getAction()
WanderAction
ActionList
Opt
[isConscious() == false]
enterDormant()
Opt
[if shell destroyed && dormant == true]
<<creates>>
:SuperMushroom



For example, if player uses super mushroom, he becomes ENLARGED status, which then provides all the changes from super mushroom


:Actor
:TradeAction
:Toad
:Wallet
Toad class has a hash map <Item, Integer>, where Integer is the price of the item
execute(actor, map)
getPrice()
price:int
getBalance()
balance:int
Alternative
[balance>=price]
updateBalance(price)
<<creates>>
:Item
AddItemToInventory(Item)
message: "Item added to inventory"
[balance<price]
message: "You don't have enough coins"

Design Rationale

1. **Enemy is an abstract class**
   Since enemies are a common thing in a game such as Mario, it would make sense to implement an abstract class just for the various enemies. This would prevent enemies needing to remake identical methods for enemies with similar properties. For example, it is aggressive towards the player, which would violate the Do Not Repeat Yourself principle. Furthermore, the code would be much easier to implement and maintain in the long run in case more enemies are added in the game.

2. **Item is an abstract class**
   There are numerous items in game, but all share something in common, they can be picked up by the player and be used for certain buffs or specific purposes (breaking koopa shells). Hence, it would be a great idea to create an abstract class to create methods for all of their similarities and allow each item to inherit from the abstract class and add on their differences. Furthermore, it would make it easier for the player class as now player can store all items (present and future) in an arraylist of item class objects. Therefore, the code would be easily maintained in the future as players can always store newly added items into the same arraylist (no new arraylists are needed for each new item).

3. **MagicalItem is an abstract class**
   Magical items are special power ups that can provide unique buffs to the player, they can be either bought from Toad via trading, or picked up by the player from the ground. For now, magical items have similar properties to regular items (the methods in the abstract item already include methods such as tick to keep track of item despawning), however when more magical items and new properties specific to magical items are added in the future, the MagicalItem abstract class would allow easy code maintenance for all future magical item changes.

4. **Player has an association with CapabilitySet**
   A player can be affected by multiple statuses at the same time, and each status provides different capabilities or disabilities. To allow easier implementation, a player receives its stat changes, abilities or debuffs from CapabilitySet which then calls from enum to gain the inflicted status(es) towards the player. CapabilitySet processes all the changes towards a player from a specific status, compiles all the changes then returns it towards the player. This is how Player will be affected by incoming statuses.

5. **Koopa has a relationship with SuperMushroom**
   I believe Koopa should have a relationship with SuperMushroom because whenever a Koopa is killed (shell destroyed), it will spawn a SuperMushroom for the player to pick up in its place. Hence, the way I would like to implement it is whenever a koopa is killed, it will create a new instance object of SuperMushroom to replace it.

6. **Spawn Interface.**
   Under requirement 1, we have Sprout that spawns Goomba's(Actor) and Sapling that spawns coins (Item). Thus, we decided to implement a spawn interface instead of coding a spawnCoin method inside the Sapling class. The spawn interface would allow us to spawn any items if we wished to spawn different items in the future. If we chose the route of a spawnCoin method, then we would need to modify the class and add a spawnSuperMushroom method if we wanted to be able to spawn Super Mushroom in the future, thus violating the Open-Closed Principle.

7. **SpawnItem and SpawnActor are Interfaces.**
   Continuing from above, we observe that not all objects spawn actors or spawn items. Thus, it would be a much better design to break down the spawn interface into spawnItem interface and spawnActor Interface. This is in line with the Interface Segregation Principle.

8. **Added Util Class to compute hit rate.**
   Throughout the game, we have many classes that involve a mechanism of performing some action, such as Sprout having 10% chance to spawn Goomba, jump success rate, attack hit rate and more. Instead of doing the computations inside the respective classes, we have decided to create a util class that does the job for us. Thus, we have reduced duplicate code which is in alignment with the DRY principle. We have also implemented the Single Responsibility Principle (SRP) by making the computation of hit rates separate from the respective classes.

9. **Tree is an abstract class. Sprout, Sapling and Mature inherit from Tree.**
   We have decided to make Tree an abstract class because we do not want to instantiate any Tree objects. The Tree class acts as an abstract class and provides the foundation for each type of Tree, which are Sprout, Sapling and Mature. Tree can have an abstract grow() method since the 3 types of trees all have a grow mechanism that are also different from each other.

10. **JumpableTerrain interface and Jumpable Interface.**
    In the game, we have different types of grounds such as Dirt, Wall and Tree. To differentiate the terrains that can be jumped over, we have implemented a JumpableTerrain interface. Grounds that implement this interface can be jumped over, and the interface can have methods such as getFalldamage(). In the future, if we had more types of grounds and wanted them to be able to be jumped over, those grounds could simply implement the JumpableTerrain interface, thus making code that can be extended easily.

    Next, we have the Jumpable interface. Instead of just coding a jump method inside Player, we decided to implement an interface so that any actor can jump. For example, to allow enemies to be able to jump, the Enemy class just needs to implement the Jumpable interface. Jumpable interface can have a jump method that allows an actor to jump.

The two interfaces complement each other nicely as well. To perform a jump, we first check if the ground implements the JumpableTerrain interface. Then, we just check if the actor implements the Jumpable interface. The idea of implementing an interface rather than just coding a method is in line with the Open-Closed Principle. We can make an enemy be able to jump without having to modify the enemy class!

11. **TradeAction is executed when the actor buys the items from the Toad.**
    TradeAction extends the Action abstract class because TradeAction has common functionality as the Action class which is to execute the action chosen by the actor. By extending the Action abstract class, the TradeAction class can reuse the methods in the Action class and override the abstract methods to provide concrete implementation of each of them which would reduce the redundancy of code in the the class as well as improve the readability of the code. In order for the TradeAction to get the price of the specific item, the Toad class has a hashmap <Item, Integer> that stores the item and its respective price. After getting the price, the TradeAction can perform the trading of the item, by getting the balance from the actor's wallet to perform the transaction. The balance will then be updated into the actor's wallet and the bought item will be added into the inventory of the actor.

12. **Wallet class is used to store the balance of coins the player has.**
    Wallet is separated as its own class because it has its single responsibility to only deal with the balance of the player, e.g. update the balance when the player picks up the coin from the ground, or when the trading with toad occurs. This enforces the Single Responsibility Principle (SRP) where one class has only one responsibility. If we do all the balance updates in the player class, the player class will become a large class which has many responsibilities to carry, thus it violates the design principle of single responsibility. The accessibility of this wallet class should be restricted from certain classes, for instance the enemy should not have access to the wallet system, otherwise the security issue will be raised up when the game is running. Other than that, If the player is first instantiated, a static factory method of getInstance() will be invoked to create a wallet for the player. Thus, each player will have only one wallet to store his balance.

13. **Speak Action extends the Action to enable the interaction between Toad and Player.**
    Speak Action extends the Action abstract class because Toad will have an action to speak with the Player. When the player is holding the Wrench or Power Star, the Toad will be restricted from speaking a specific sentence. The sentence that is speaked by the Toad is dependent on the items that the player is holding at the moment. So, the speak action has a relationship with the items, more specifically Wrench and PowerStar to allow certain sentences to show on the console of the player. The reason why the speak action is not implemented as a method in the Toad class is because, in the future if the Toad is no longer has the ability to speak to the Player, instead there is another actor that is able to speak, we may need to modify the code by extracting the method out from the Toad class, which violates the Open Closed Principle (OCP). By having the Speak Action class, the actor that wishes to

speak with the player can simply add the speak action into its own allowable action list, which indeed is much more maintainable in the long run.

14. **Reset Manager stores a list of Resettable instances and it does the task to reset the game.**
    Reset Manager is a singleton manager that does all the reset tasks to the instances that implement the Resettable interface. Only one instance of reset manager will be instantiated throughout the game and the player can only reset the game once. So for any classes that are resettable, they will be appended to a list of resettable instances and being reset by the Reset Manager when the hot key for reset ’r’ is chosen by the player. In order to allow the player to reset the game only once, when the reset manager first runs the reset tasks, there could be a counter to keep track of the number of reset being invoked. If the counter is more than one, reset is not allowed anymore throughout the game.

15. **Coin item extends Item abstract class**
    Coin item extends Item abstract class because the coin represents a physical item on the ground. The player could pick up the coin from the ground and the amount of the coin will be added into the wallet of the player. The coin will not be included in the player’s inventory.

FIT 2099 Assignment 1 Work Breakdown Agreement

Lab 06 Team 2:

1. Lo Kin Herng (32023995)

2. Tang Kai Kit (31862071)

3. Ong Kai Yun (31861369)

We hereby agree to work on FIT 2099 Assignment 1 according to the following breakdown:

Class Diagrams (To be completed by 4th April, Monday):

~ Lo Kin Herng works with REQ1 and REQ2

~ Tang Kai Kit works with REQ3 and REQ4

~ Ong Kai Yun works with REQ5, REQ6 and REQ7

~ Initial commit: 1st April

~ Reviewers: All 3 members

Interaction Diagrams (To be completed by 6th April, Wednesday):

~ To be done by Tang Kai Kit and Ong Kai Yun

~ To be reviewed by Lo Kin Herng

Design Rationale (To be completed by 8th April, Friday):

~ To be done by all 3 members

Author: Lo Kin Herng

Date: 2nd April

Acknowledgement:

I accept this WBA. Signed by Lo Kin Herng

I accept this WBA. Signed by Ong Kai Yun

I accept this WBA. Signed by Tang Kai Kit