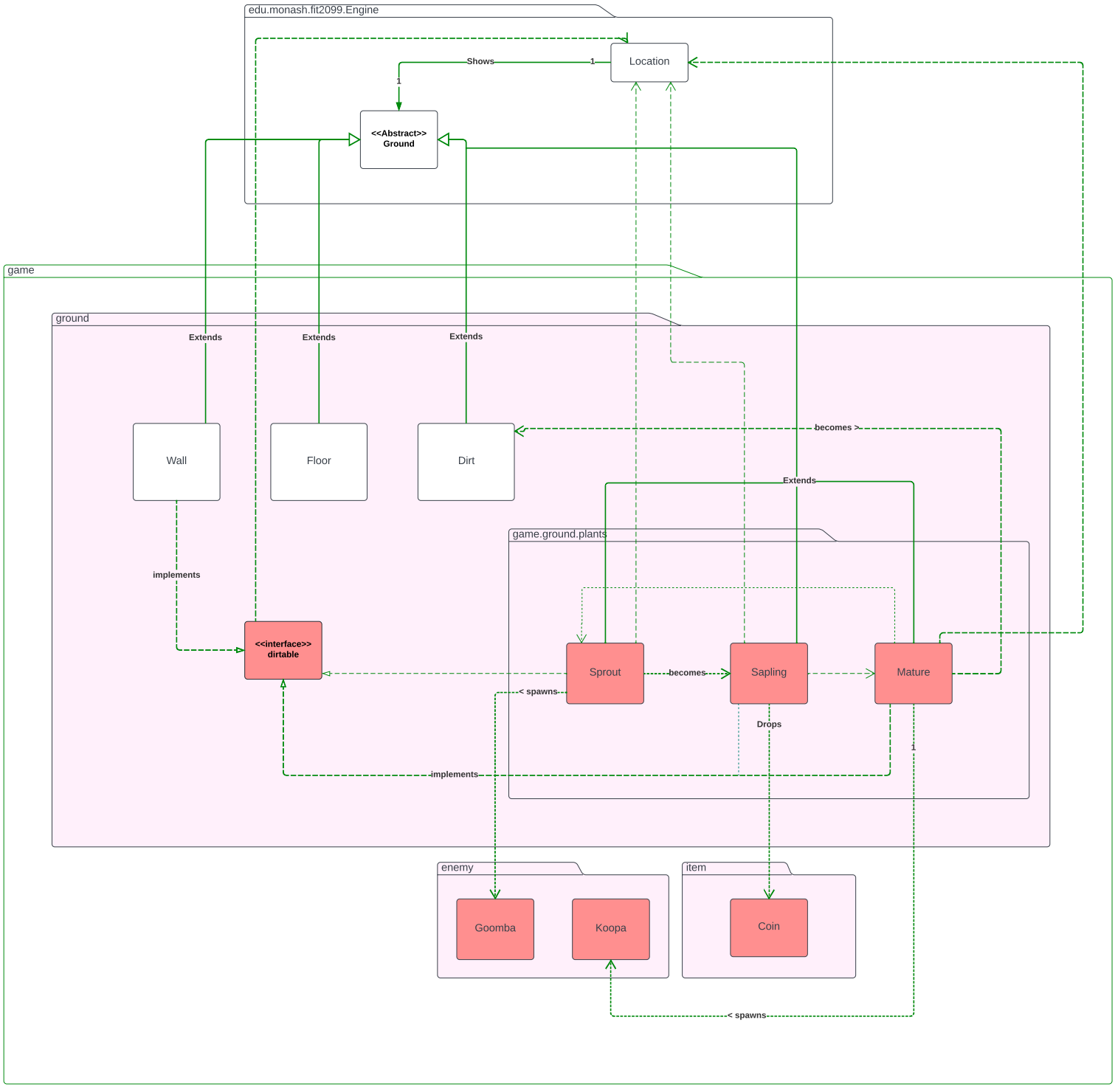
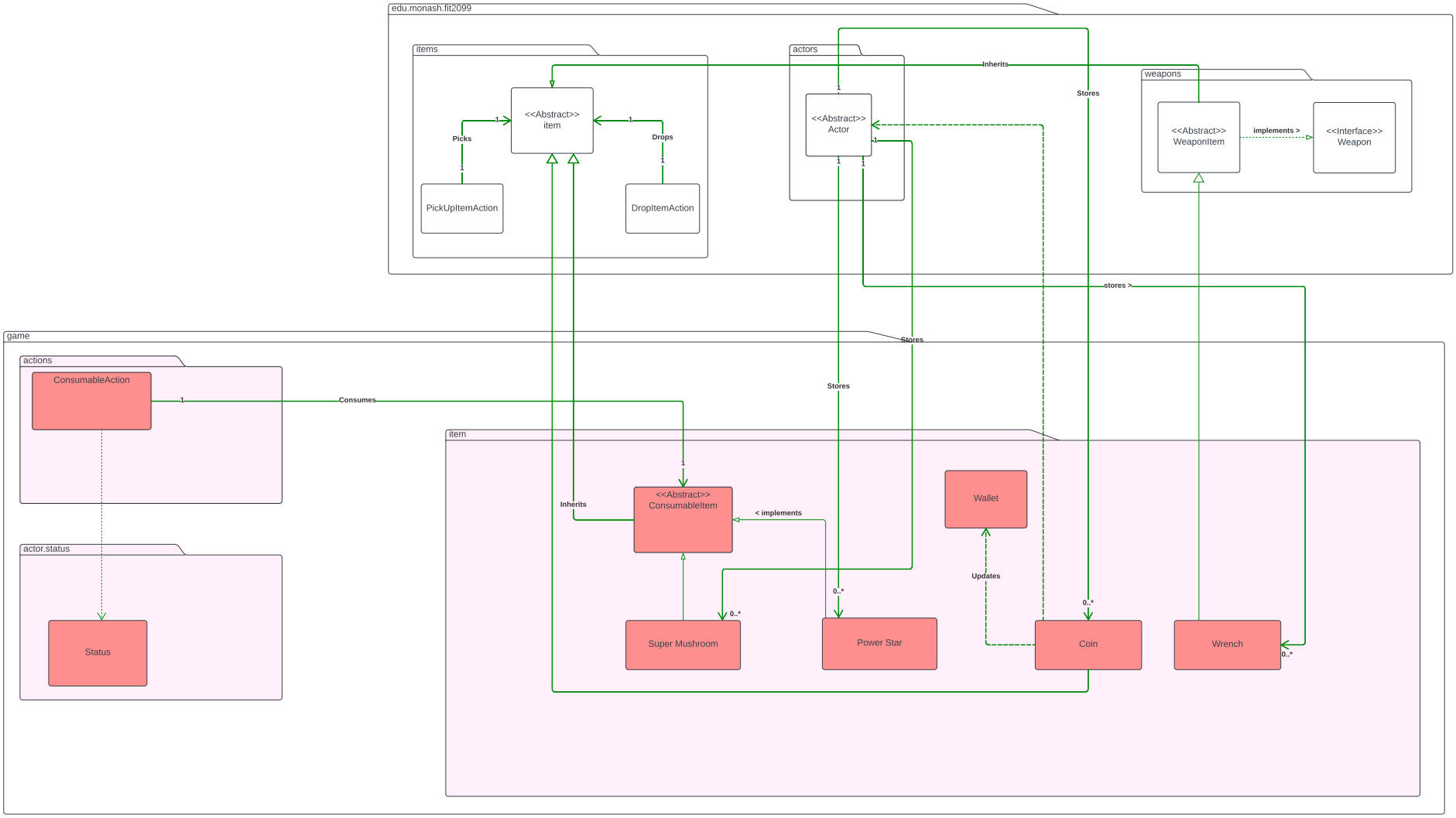
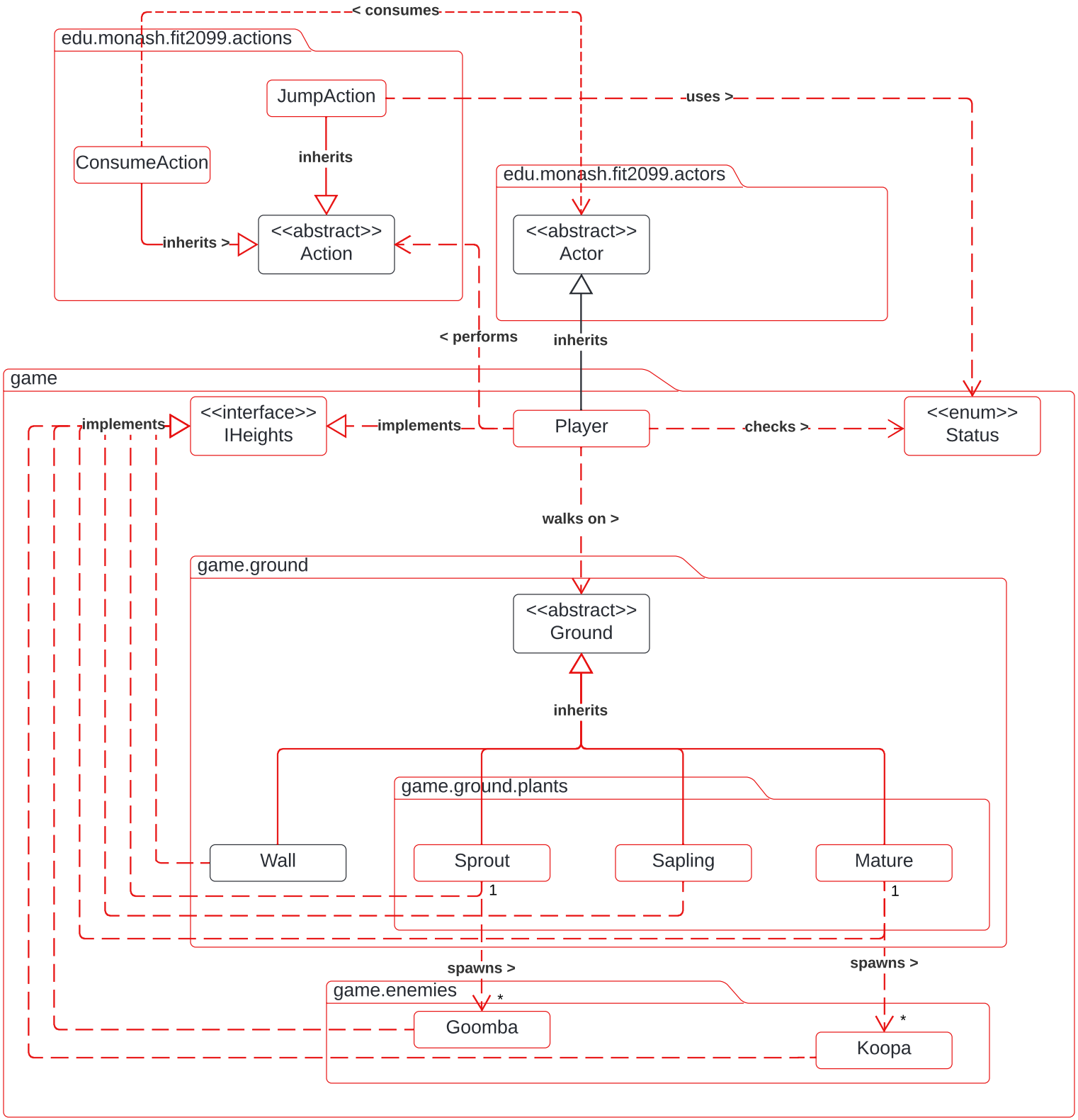
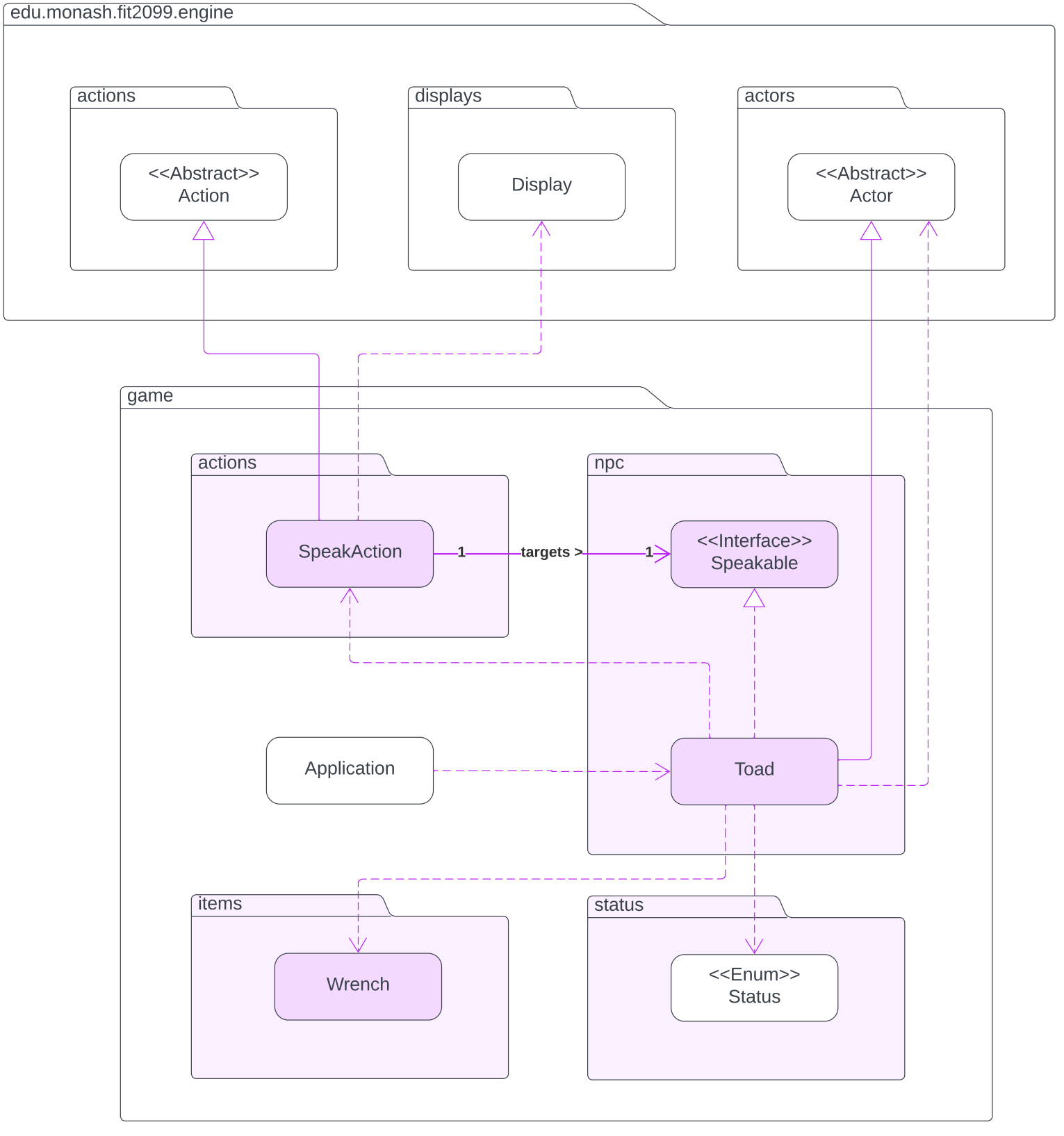
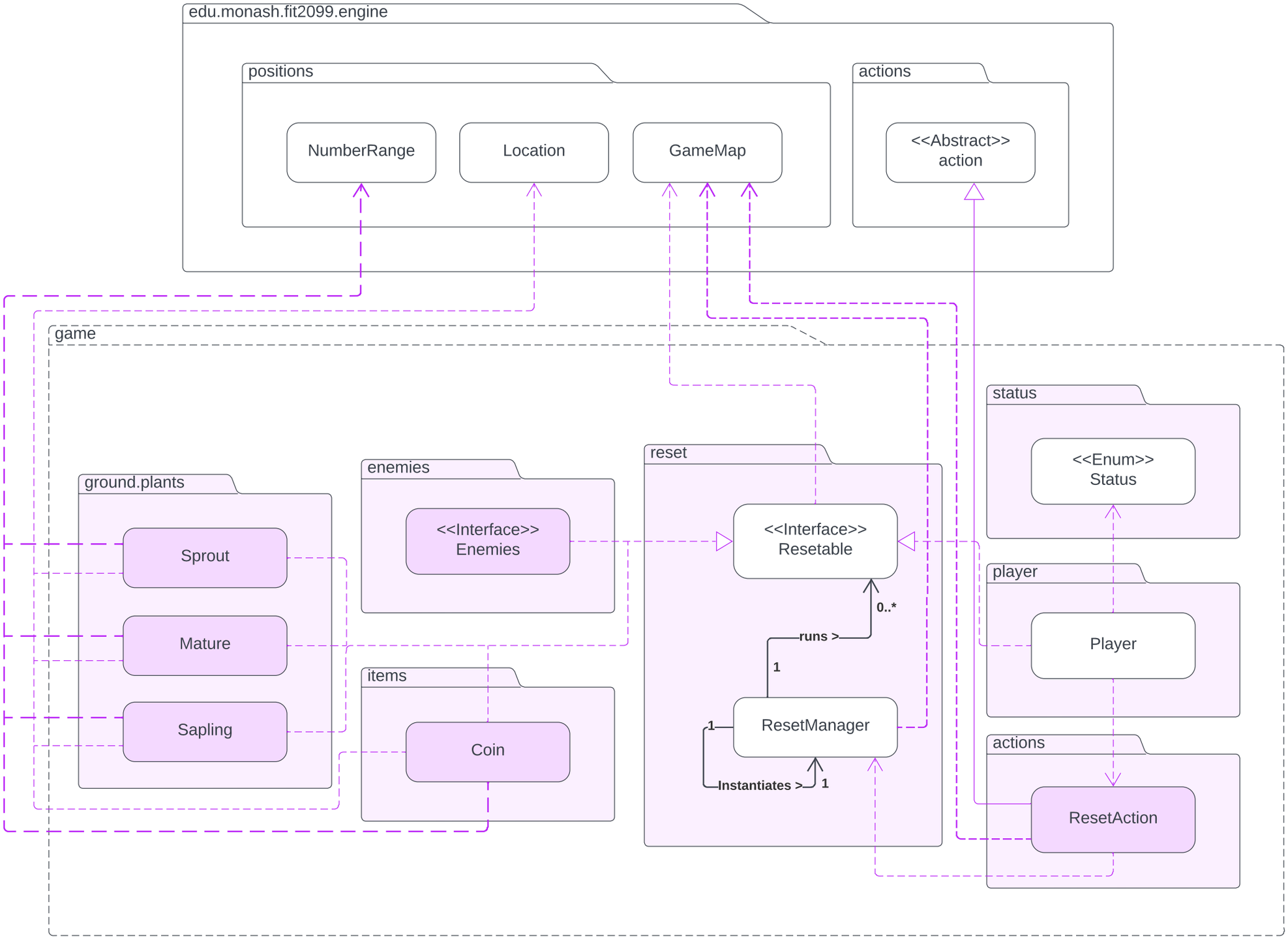
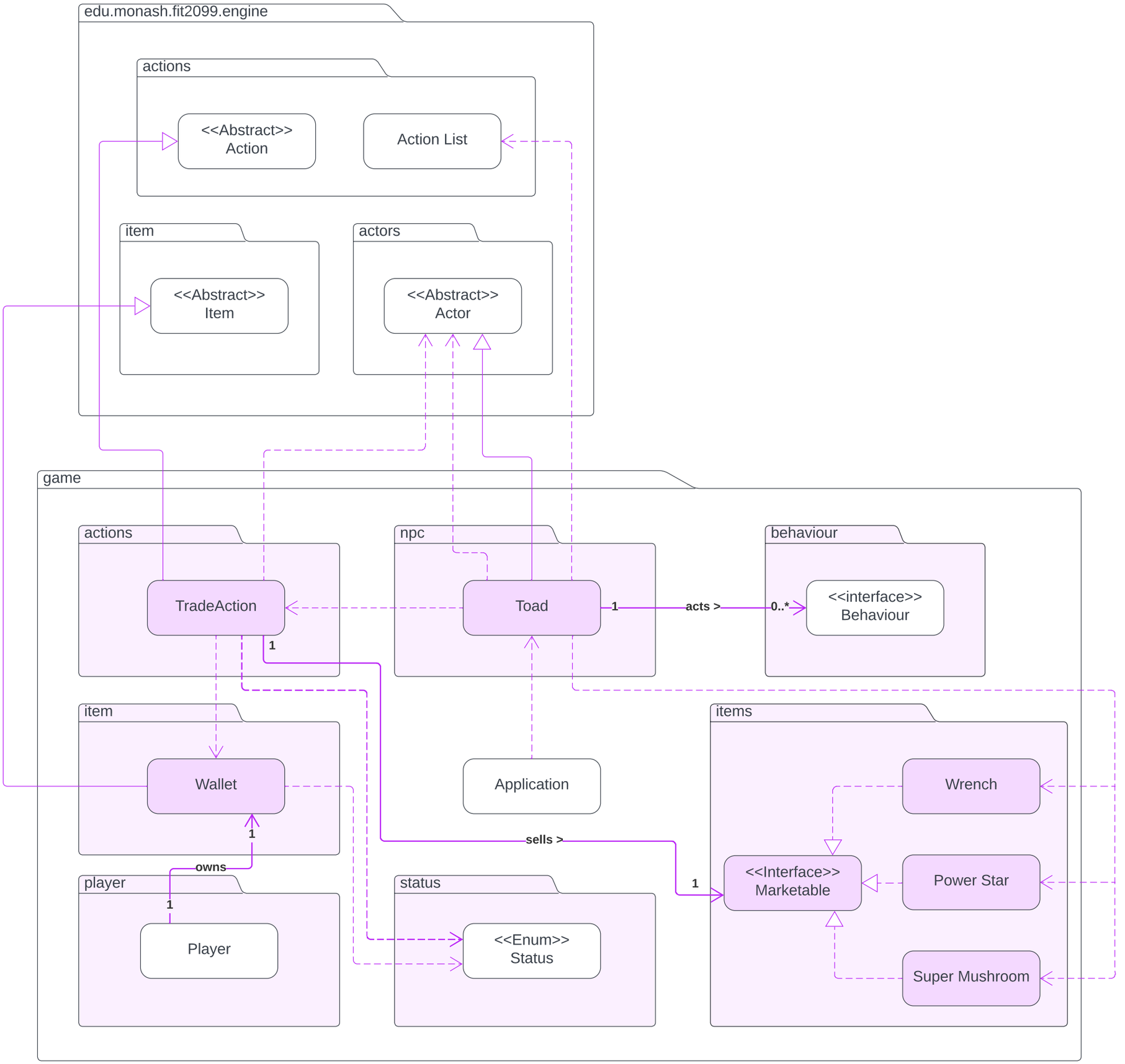

edu.monash.fit2099.Engine
Location
Shows
1
1
<<Abstract>>
Ground
game
ground
Extends
Extends
Extends
Wall
Floor
Dirt
becomes >
Extends
game.ground.plants
implements
<<interface>>
dirtable
Sprout
becomes
Sapling
Mature
< spawns
Drops
1
implements
enemy
item
Goomba
Koopa
Coin
< spawns

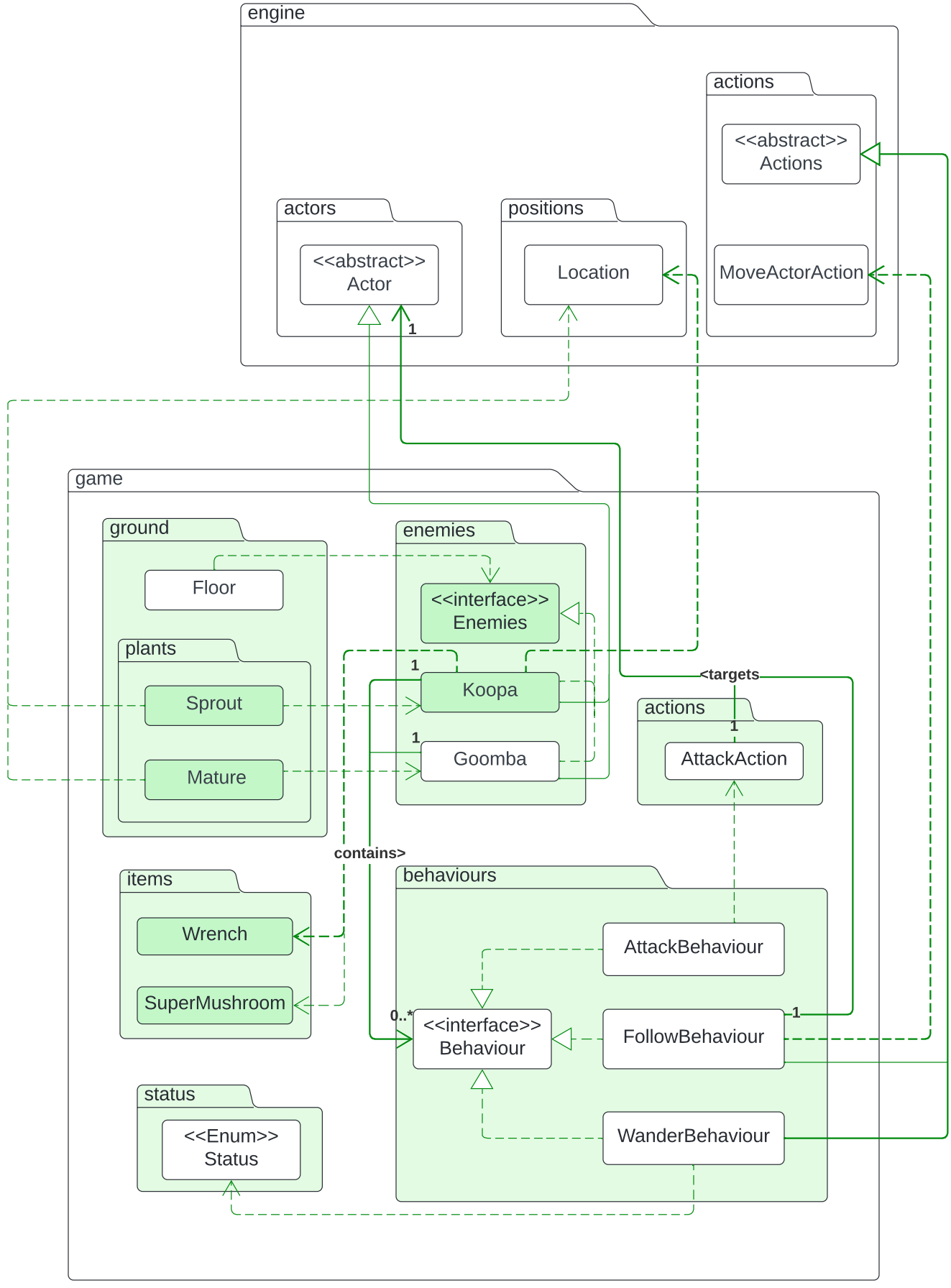

edu.monash.fit2099
items
<<Abstract>>
item
Picks
Drops
PickUpItemAction
DropItemAction
actors
<<Abstract>>
Actor
Inherits
Stores
weapons
<<Abstract>>
WeaponItem
implements >
<<Interface>>
Weapon
stores >
game
actions
ConsumableAction
Consumes
actor.status
Status
item
<<Abstract>>
ConsumableItem
Inherits
< implements
Wallet
Updates
Super Mushroom
Power Star
Coin
Wrench
0..*


< consumes
edu.monash.fit2099.actions
JumpAction
uses >
ConsumeAction
inherits
edu.monash.fit2099.actors
inherits >
<<abstract>>
Action
<<abstract>>
Actor
< performs
inherits
game
implements
<<interface>>
IHeights
implements
Player
checks >
<<enum>>
Status
walks on >
game.ground
<<abstract>>
Ground
inherits
game.ground.plants
Wall
Sprout
Sapling
Mature
1
1
spawns >
spawns >
game.enemies
*
*
Goomba
Koopa

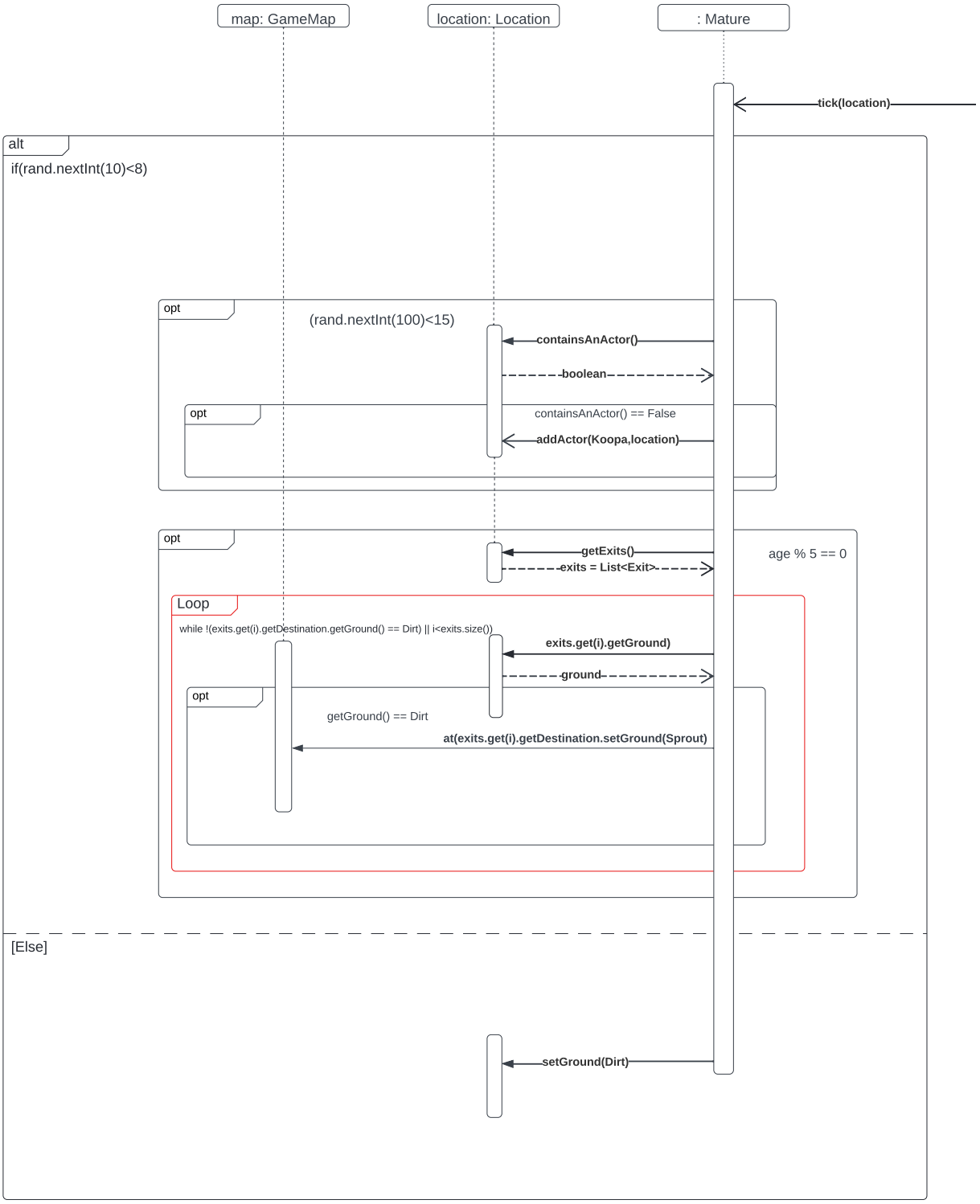

edu.monash.fit2099.engine
actions
<<Abstract>>
Action
displays
Display
actors
<<Abstract>>
Actor
game
actions
SpeakAction
1
targets >
1
npc
<<Interface>>
Speakable
Application
Toad
items
Wrench
status
<<Enum>>
Status


edu.monash.fit2099.engine
positions
NumberRange
Location
GameMap
actions
<<Abstract>>
action
game
status
<<Enum>>
Status
reset
enemies
<<Interface>>
Enemies
<<Interface>>
Resetable
ground.plants
Sprout
Mature
Sapling
0..*
runs >
1
player
Player
items
Coin
1
ResetManager
1
Instantiates >
actions
ResetAction

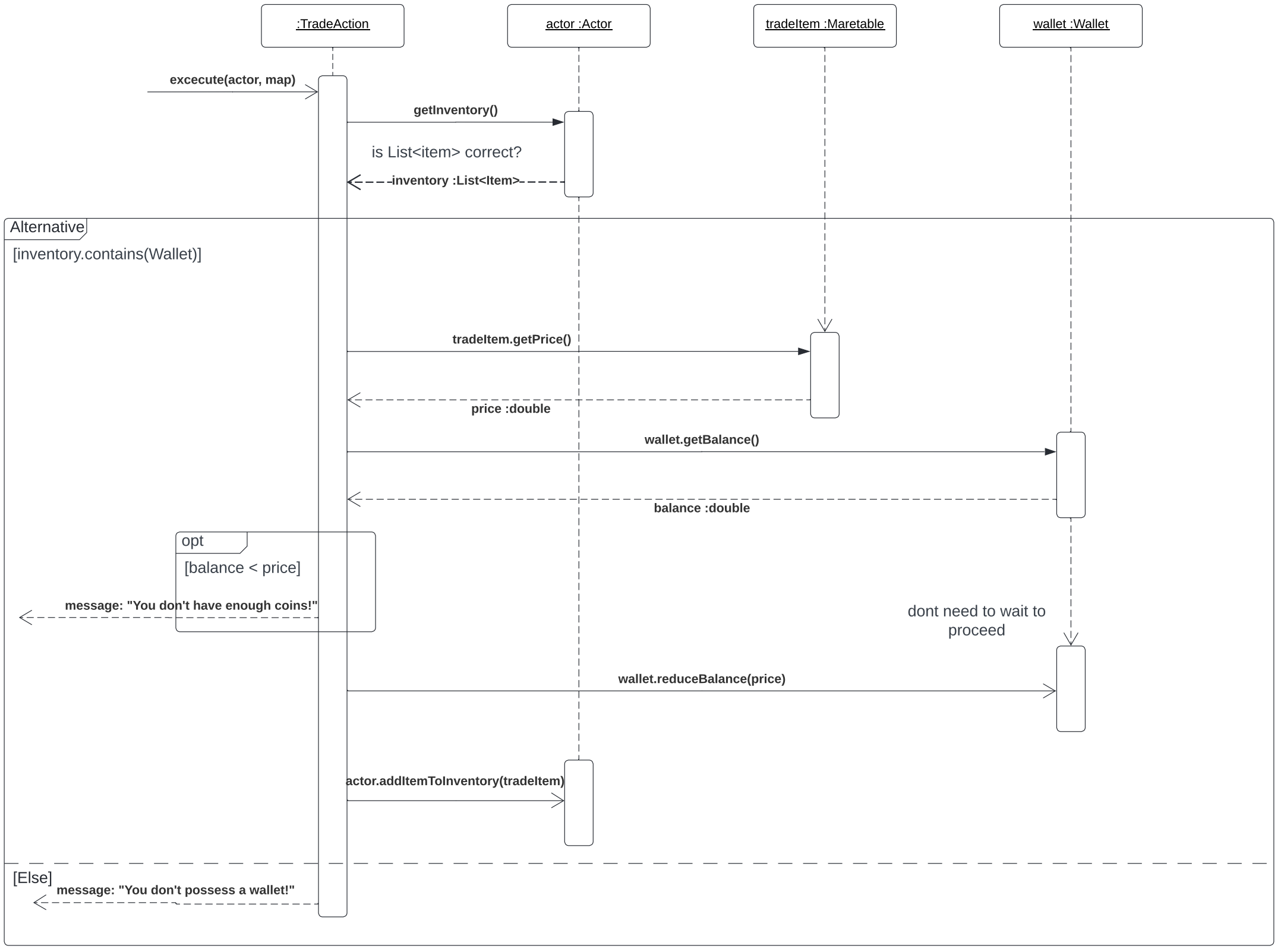

edu.monash.fit2099.engine
actions
<<Abstract>>
Action
Action List
item
<<Abstract>>
Item
actors
<<Abstract>>
Actor
game
actions
TradeAction
npc
Toad
behaviour
<<interface>>
Behaviour
1
acts >
0..*
1
item
Wallet
Application
items
Wrench
1
sells >
owns
1
<<Interface>>
Marketable
Power Star
player
Player
status
<<Enum>>
Status
Super Mushroom


engine
actions
<<abstract>>
Actions
actors
<<abstract>>
Actor
positions
Location
MoveActorAction
1
game
ground
Floor
plants
Sprout
Mature
enemies
<<interface>>
Enemies
1
Koopa
1
Goomba
<targets
actions
1
AttackAction
contains>
items
Wrench
SuperMushroom
behaviours
AttackBehaviour
0..*
<<interface>>
Behaviour
FollowBehaviour
1
WanderBehaviour
status
<<Enum>>
Status

Sequence Diagram - Mature.Tick()



Sequence Diagram - TradeAction.excecute()



Classes and Design Rationale

Classes

World

- this controls and manages how the game is run (IMPORTANT).
- runs the main loop that controls the flow of the game.
- At each iteration of the world, draws `GameMap()`.
- controls state of game map
- tells player state.
- Should manage multiple `GameMaps()`'s at each iteration.
- time should flow for all maps.

IMPLEMENTATION DETAILS (GIVEN):

- `run()`: contains the main game loop, defined in a while loop. the condition for this while loop is determined by the return value of the `stillRunning()`.
- Game will run as long as player is alive.
- Rendering: Renders the map where the player is currently located.
- Finds the GameMap the player is currently located.
- Calls map method of `Location` in order to obtain `GameMap` that the location is currently in.
- `draw()`: next the draw method is called from the GameMap class, and draws each (x, y) position in the game map.
- Processing: Then it processes each actor's turn and checks if `stillRunning()` is true.
- `tick()`: then edits the tick.

GameMap

IMPLEMENTATION DETAILS (GIVEN)

- In the `World()` there can be more than one GameMap.
- Purpose is to help the `World()` in controlling and management of game.
- manages players, objects and states.
- Receives orders from `World()`.
- Time should be same and flow for all different maps.

Location

IMPLEMENTATION DETAILS (GIVEN)

- Purpose is to provide the ability to get the exact locations of Actors/entities.
- Simple use of x and y coordinates
- Each position in the `GameMap()` will be defined by the `Locations()`
- GameMap contains Locations.

- Responsible for telling `World()` and other game entities what is available at position (x,y).
- keeps track of:
- exits
- character representations
- terrain type
- entity representations
- tick timers for each item.
- Contains:
- items
- ground
- exits
- should also keep track of Actor's locations?
- Can move actors, checks if actors can enter location
- can only contain ONE actor at a specific Location.
- location method draw already has resolved display hierarchy.

Exit

IMPLEMENTATION DETAILS (GIVEN)

- Purpose is to describe the surroundings of a given location and represent a route from one `Location()` to another.
- Provides you with the name of the exit, its location and the movement hotKey.
- gives you a list of possible locations that you can move to.

Ground

IMPLEMENTATION DETAILS (GIVEN)

- Purpose is to allow choices for terrain types.
- wall
- safe area (floor)
- sprouts, sapling or mature.
- These terrain types allows different entities to tell whether they can pass through the location or not.

Notable Features:

- want to know whether a certain entity can enter a specific ground.
- Some different ground terrain types may need different abilities,
- safe hiding area
- block projectiles etc.

Sprout

Purpose: To spawn Goomba Objects and then Form into a sapling after it exists for 10 turns

- Single Responsibility: To act as an object that spawns Actors with the added responsibility of turning itself into another object
- Open-Closed: Sprout should only be open to extension to the Ground Class
- Liskov's Substitution: Replacing Ground with Sprout should not result in any unidentified behaviour
- Interface Segregation: The sprout object will implement the Dirtable interface which grants it the ability to implement a method to turn itself into dirt if certain conditions are met

---

Sapling

Purpose: To spawn Coin Items and then form into a Mature after it exists for 10 turns

- Single Responsibility: To act as an object that spawns Items with the added responsibility of turning itself into another object
- Open-Closed: Sapling should only be open to extension to the Ground Class
- Liskov's Substitution: Replacing Ground with sapling should not result in any unidentified behaviour
- Interface Segregation: The sapling object will implement the Dirtable interface which grants it the ability to implement a method to turn itself into dirt if certain conditions are met

---

Mature

Purpose: To spawn Koopas at its location, Sprout objects in its surrounding location and to form into Dirt on a 20% chance

- Open-Closed: Mature should only be open to extension to the Ground Class
- Liskov's Substitution: Replacing Ground with mature should not result in any unidentified behaviour
- Interface Segregation: The mature object will implement the Dirtable interface which grants it the ability to implement a method to turn itself into dirt if certain conditions are met

---

Implementation Details (Sapling, Sprout, Mature):

In order to spawn new Actors, and also spawn a new ground object on the current location of ground. These objects will access the location class through the Location parameter in the tick function. Giving it access to methods implemented within the location class. Allowing it to set new grounds, get Actors and add new actors from the current location it is at. It would also therefore allow the object to access its surroundings through the exit functions, granting it access to perform methods on its surroundings as-well through the map object stored in the Location class, and the location method that retrieves the destinations of the current locations exits

Dirtable

I chose to use an interface to represent the objects that need a method to turn themselves into dirt. This is because throughout multiple stages of the game (ie. During the reset function and during when mario is under the consumption of a Power Star, the Wall, Sprout, Sapling and Mature objects can be turned into

dirt. So an interface is therefore implemented to avoid the repetition of coding, and takes a location parameter which will set the ground at that current location to Dirt.

Item

IMPLEMENTATION DETAILS (GIVEN)

- Purpose, to act as an object that the Player can use.
- needs to be Portable or nonPortable
- can be picked up or dropped.
- stays in players inventory. (not droppable)
- Item may also have a duration – uses `tick()` and overrides it.

Consumables

Purpose: Allows us to consume the item from the inventory in order to unlock buffs for a specified period of time

- Open-Closed: Both the Super Mushroom and Power Star should only be open to extension to the ConsumableItem Class
- Liskov's Substitution: Replacing both PowerStar and SuperMushroom with ConsumableItem should not result in any unidentified behaviour

Implementation Details Both classes should inherit a new abstract class within the Games package called ConsumableItem, this gives the class details in regards to whether or not it is able to be consumed by the player. This helps reduce code repetition as it can be defined in the abstract class and then inherited by both classes.

A new class within the Actions package named ConsumableAction is used that gives the player the option to consume items if it recognises that it has attributes that belong to the ConsumableAction parent class. Otherwise it will not recognise the item as an item that can be consumed (ie. Coin, Wallet, Wrench) In order to grant the player benefits after the consumption of an item. New status can be defined (ie. Possibly referred to as powerStar and superMushroom). That is then added to the capabilities of the person who consumes the item, which would then result in buffs given to the actor

---

Wrench

Purpose: to Act as a item and a weapon used to break the shell of a dormant Koopa

Open-Closed: the Wrench class should only be open to an extension of the weaponItem abstract class

Liskov's Substitution: replacing Wrench with weaponItem should not result in any unidentified behaviour

Implementation details: Wrench is going to be inherit the WeaponItem class, so it can be stored within the Inventory whilst being classified as weapon. This will allow the getWeapon() function to recognise the wrench as a weapon to be used to attack enemies. In order to kill the koopa shells, we would need to ensure that the weapon used is an instance of a Wrench

---

Wallet

Purpose: to Act as an item that is stored in the actors inventory and is used to store the balance of coins when they are picked up by an actor that has a wallet in their inventory Open-Closed: the Wallet class should only be open to an extension of the Item Abstract class Liskov's Substitution: replacing Wallet with Item should not result in any unidentified behaviour

Implementation Details: The main purpose of the wallet class is to serve as an item with a balance value attached to it. It will be added to the players inventory upon initialization of the game, as the player is the only actor that has use of the coins, but to also make sure that other actors can also pick up the coins. This is because when the player picks up the coins, it will update the balance of the Wallet value instead of picking up the coin and storing it in its inventory. This is most likely going to be achieved by viewing the Actors inventory, and checking if any items they have are an instance of Wallet, if so, that object associated values will be updated, else, the actual coin object will be stored. The wallet's value will then be used for trading

---

Coin

Purpose: to Act as an item that is stored in the actors inventory unless they have a Wallet item, which it is then added to the wallets balance and not stored in the inventory

Open-Closed: the Coin class should only be open to an extension of the Item Abstract class Liskov's Substitution: replacing Coin with Item should not result in any unidentified behaviour

Implementation: The coin object will be assigned a particular value that represents the coins monetary value. (ie. 5,10,20,etc) This will be done within the Constructor, after it calls its super back to the Item class. The values of these coins will be dependant on the situation in which the coin is found. For example, coins created through Saplings will be assigned a value of 20 whereas coins dropping from destroyed terrain will only be created with a value of 5. This is manually inputted as we control when the coins are dropped and the situation in which they are created Playing off the wallet class, the Coin acts as an object scattered throughout the map, upon pickup by any actor, the game would first have to check if the actor has a Wallet (which would only be the Player). This is to allow any actor to pick up the coin object without any issues. If the actor does have a wallet, the PickUp action will simply read the value of the coin, update the wallets balance and delete it from the map. However, if the actor does not have a Wallet, it will simply just add the coin item to its inventory without any further actions.

Actor

IMPLEMENTATION DETAILS (GIVEN)

- Purpose: Allows us to represent Player, Enemies and non-hostile entities (such as Toad) as actor classes.
- Implementation Details:
- Hit Points: Already given in the code.
- Knowing the Actor's hit points will allow us to know what the next actions to take are.
- Actors Action List: Knowing what actions the Actor can take is vital to gameplay as well.

- Location will care about actor heights and GameMap will have the logic for whether the Player needs to jump or can simply walk down.

Capability

- Purpose: Enabling the ability to inflict special statuses on Actors, or entities
- Implementation Details (GIVEN):
- Can create an `Enum` to do this, which stores a variety of different statuses

NOTE:

- Actor: the `Player` can get an ATTACK_UP status once they drink a potion
- Item: a weapon might have a CURSED or a POISONED status attached to it.
- Ground: might have a SLIPPERY or WET status attached to it.
- Capabilities: `Status` could be divided into two types
- Buff: any status that improves performance of actor.
- Debuff: any status that worsens performance of actor.
- Temporary: A status that can be removed after a certain number of turns
- Permanent: A status that cannot be removed the entire game.
- ==Do not use statuses to represent identities==.

Action

- Purpose: Represents the action that an actor can perform.
- Allows Player to perhaps attack the Enemy, consume items, and perform other actions.
- Implementation Details (GIVEN):
- The action that the actor can perform will be given by the object.
- Another `Actor` as the object: If an Enemy stands next to the `Player` the following conversation can occur.
    - Engine to Enemy: What actions is the `Player` allowed to use on you?
    - Enemy to Engine: If the `Player` has the `HOSTILE_TO_ENEMY` status, then they can perform `AttackAction` on me.
    - Engine to Player: `Player`, you can do `AttackAction` on the `Enemy`.
    - `AttackAction` will be displayed to the user.

==NOTE: Do not implement all the actions that the `Player` can implement inside the `playTurn()` - violates SRP.==

Behaviour

- Purpose: To allow NPC such as Enemy to execute actions. Acts as a wrapper to an action that an NPC can perform, as cannot make decisions based on input.

- Implementation Details:
- An NPC would have an attribute that stores a list of Behaviour's. each `Behaviour` from the list will return an action. Therefore, the list of `Behaviour`s will result in a list of actions that the NPC can choose from.
- Behaviour: may return a null action
- Behaviour: may have higher priority than others.
    - Attack behaviour should be at the top of priority
    - Follow behaviour should be next... etc.

Capable

NOTE: This interface is Capable as Capable sets Statuses and This provides a way for an Enemy to attack another actor.

- Single Responsibility: To provide a way for Actors to interact with each other, say you have an enemy, `Goomba` and it is near the `Player` then you want to allow `Goomba` to attack `Player`.
- Open-Closed Principle: Since the `Actor` class is open to inheritance, this allows enemies such as `Gooba` or `Koopa` to inherit its useful attributes and the `Actor` class stays closed to modifications.
- Liskov Substitution: Enemy classes `Goomba` and `Koopa` as well as the `Player` class can be substituted for the `Actor` class and will not have any undefined behaviour.
- Interface Segregation: This Interface has the single responsibility to allow Actors to attack each other.
- Dependency Inversion: since Liskov's and OCP are already adhered to, this one is also adhered to.

Goomba

- Purpose: Act as a hostile entity towards Player instance, attacks Player, follows player if near. Clears the map if unconscious.

- Single Responsibility: A hostile entity to the `Player`.

- Open-Closed: `Goomba` inherits from the open abstract `Actor` class, which is closed to modifications. `Goomba` itself is the child class and will be the enemy that is hostile towards the `Player`.

- Liskov's Substitution: Since `Goomba` follows the OCP, it also satisfies this Liskov Substitution principle. It can be replaced with the `Actor` class and no undefined behaviour should occur. To ensure this, `Goomba` implements the interfaces: WanderBehaviour, AttackBehaviour and FollowBehaviour to allow it to wander around the map, attack the Player if able to and also follow the Player if able to, respectively.

- Dependency Inversion: Follows from satisfying OCP and LSP. This `Goomba` class should not depend on the three `Behaviour` classes, instead it will depend on the Interface `Behaviour`. The `Goomba` class can hold a collection of `Behaviours` that the Object `Goomba` itself will use to decide its next move.

- Implementation Details:

- Since it is bad to have too many levels of inheritance, `Goomba` will only inherit from `Actor` class and then implement its own attack success chance (hit rate) and removes itself from the map if it has

0HP.

- Inheriting from the `Actor` class is beneficial as it allows the `ActorLocationsIterator` to move the Enemy around.
- if unconscious –> remove from map.
- Sprout can spawn –> Goomba so dependency
- Cannot spawn enemy if actor standing on it.
- `Goomba` can attack and follow the `Player` so it may have a dependency on the `Player`.

Koopa

- Purpose: To act as a hostile towards the Player class, it attacks the player with more damage than the Goomba
- Single Responsibility: To act as another hostile enemy towards the `Player` with the added responsibility of dropping `SuperMushroom`s.
- Open-Closed: Goomba should only be open to extension
- Liskov's Substitution: Replacing Actor with Koopa should not result with undefined behaviour.
- Interface Segregation: Like `Goomba`, `Koopa` will implement the `Behaviour` interface to allow the class to be able to Follow, Attack and Wander around map.
- Implementation Details:
- Hit Points: starts with 100HP
- Attacks with punch that deals 30 dmg, 50% hit rate.
- When defeated, Map display icon changes to `D`.
- Mario needs `Wrench()` to destroy the shell.
- When shell is destroyed, Super Mushroom is dropped:
    - map icon should display SuperMushroom icon
- Unconscious: Cannot attack, follow or wander around.

Jump Requirement

- Purpose: The `Actor`, `Player` will need to be able to move to higher ground level such as on top.
- Implementation: Using an interface `IHeights` which has an integer attribute for the height and a method to set the height. By having the different types of `Ground` (Sprout, Sapling, Wall and Mature) implement this interface we can give each object a height as well as having the `Player` implement this interface. Next using the `actorCanEnter(Actor actor)` and overriding it, we can place

conditionals that check if the actor is at a lower level than the ground type of its destinations. If satisfied then it adds the jump capability to the CapabilitySet. This allows the actor to jump using the `JumpAction` class and if the jump option is selected, calculates the chance of success and executes the logic behind the jump.

Before checking if the actor is at a lower level, implementation will check if the actor has a SUPER_MUSHROOM_STATUS effect on it that allows it to have a 100% success rate for jumping. If the status is in effect then the Player can just move to the higher ground without any problems.

The `ConsumeAction` changes the chance of success for a jump to 100%. This action will have a dependency on the `IHeights` interface. As this interface

By using an interface that adds a heights capability it ensures that the children classes of Ground and Actor follow Liskov's Principle, as these two parent classes are open to extension but not modification. Otherwise, adding height attributes to each child class would violate this principle as well as make implementation and maintenance more difficult in the future.

As `Koopa` when unconscious becomes a shell, and when the shell is destroyed it becomes a `PowerMushroom` there will be a dependency between these two classes as well as a dependency between `Toad` and `PowerMushroom` as Toad sells the `Item` to the player.

Toad

-Purpose: Toad is a friendly NPC actor that spawns in the centre of the map surrounded by brick walls. The Toad provides the player with the ability to trade the player's collected ‘coins’ in exchange for items. Additionally, the toad has a set of available dialogue options that is randomly presented to the player when they speak with the toad

-Implementation: The Toad will have in its available actions method a speak action and a trade action for all the marketable items. When the toad creates the items for all the trade actions it will set the portability for every item as false.

- Single Responsibility: to provide a trading entity
- Open-Closed: `Toad` inherits from `Actor` which is open to extension and closed to modification. So this satisfies the OPC.
- Loosely coupled `Wallet` to trading to the `Player` is not tightly coupled and allows other entities to be added to the game and allows them to trade as well.
- interface Segregation: Implemented the interface `Marketable` to give certain `Items` the ability to be tradeable. This is good as they do not rely on other interfaces they don't need such as `Behaviour` to be tradable.

Trade Action

-Purpose: TradeAction represents an action an actor can perform to obtain a specific item in exchange for their ‘coins’.

-Implementation: The action will store a specific ‘Marketable’ item for which it is selling. The action works by performing its validation of whether the actor has a wallet/sufficient balance after it has been executed

as to present all the tradeable actions to the actor. The execute action works by checking to see if the player has a wallet, if so it gets its balance and returns "You don't have enough coins!" if there is insufficient. Otherwise the balance is decreased and the item is added to the actors inventory

- Single Responsibility: responsible for presenting an action that an `Actor` can perform to obtain an `Item`. Utilises `Coin`s to commit the trade.
- Open-Closed: `TradeAction` is extendable such that it can be extended to implement being able to sell items in exchange for currency. Also, it is open to allow more NPCs to trade with.
- Liskov's Substitution: `TradeAction` inherits from `Action` which is open to extension and thereby adheres to this principle. The `TradeAction` can be substituted with `Action` and will not throw undefined behaviour.

Wallet

-Purpose: Acts to store the actor's entire balance of collected ‘coins’ which is on the actor. -Implementation: The wallet is created as an extension of an item. This is so that a wallet can be added to an actor's inventory. This is to allow the implementation to work with all existing infrastructure which accepts arguments of actors which is seen throughout actions.

- Single Responsibility: To take in `Item`s and convert to currency. In this case it takes in coins that add to the balance. Subtracts from balance once a trade is completed.
- Open-Closed: Wallet inherits from the `Item` class which is open to extension. Wallet implements a balance attribute which allows the player to use currency. Wallet itself can be extended by a child class that wants to store other types of currency. e.g., TokenWallet which could possibly collect an `Item` - `Token`, and convert these to points or another type of currency.
- Liskov's Substitution: `Wallet` can be replaced by `Item` without any undefined behaviour.

<< Interface >> Marketable

- Purpose: Adds the ability for any item that implements the interface to be able to be sold/traded.
- Implementation: Adds to the items a specified trading price that the toad will sell the item for. Additionally, it will also contain a getter method to retrieve the price from the item.
- Single Responsibility: Makes an `Item` tradable and gives the `Item` a price.
- Interface Segregation: This interface has the sole responsibility of allowing an item to be tradable at a given price. All `Item`s in the game currently need these properties so this does not violate the Interface Segregation Principle.

SpeakAction

-Purpose: Causes any ‘speakable’ actor to ‘speak’ which inturn prints their message into the console.

-Implementation: The action targets a specific speakable class that for when the action is executed it will display in the console the classes desired dialogue. On execution the method calls the speakables method to retrieve a string of dialogue from which the speak action displays the text in the console.

- Single Responsibility: Allows an `Actor` to be able to display dialogue on the console. Instead of implementing speaking actions separately for each `Actor` that could've been capable of speaking, makes use of an interface to give specific `Actor`s this capability.
- Open-Closed: `SpeakAction` inherits from the `Action` class which follows the principle that the `Action` class is open to extension but closed to modification.

<< Interface >> Speakable

-Purpose: Instantiates a class with its corresponding dialogue and provides the ability to return a given line of communication. -Implementation: Instantiates every class that inherits with an independent set of various lines of dialogue. Provides a method to call upon one of the given lines of dialogue which can be dependent on the class's, and further, any of the world's circumstances/requirements. In the case of toad, it will be instantiated with 4 various lines of dialogue. Upon being called it will select a random line of dialogue, however, only out of select lines depending on the caller actor who will be passed as a parameter. The first line "You might need a wrench to smash Koopa's hard shells." will only be possible if the actor does not have a wrench in their inventory. The second line "You better get back to finding the Power Stars." will be another possibility if the actor has not consumed a power star and is not experiencing any of its capabilities.

- Single Responsibility: Gives a class a predefined set of dialogue to be chosen at random and displayed on the console.

<< Interface >> Resettable

- Purpose: Upon the game being reset, all classes that implement this resettable will be affected and from which they can perform an action specific to that actor or its class.
- Implementation: The resettable interface will have two separate execute methods. One for the execution of an action specific to that object called 'excecuteInstance' and another static method used for the execution of an action that is performed upon all the object that are instances of that class 'excecuteClass'. Both methods will be passed the parameters of the game map. After execution 'excecuteInstance' will return a boolean on whether there is further execution required for the entire class. For example, the player will only need the execute instance to heal the player to their full health, however a tree will use the excecuteClass method to iterate through all the locations and change the ground where a tree occurs to dirt. This drastically improves the speed compared to every individual tree object searching through the game map for its one occurrence.
- Single Responsibility: Allows ResetManager to not have to implement an instance of to test what to perform on any object. Implementation of the class would be closed to modification following the Open-Closed Principle.

ResetManager

-Purpose: Keeps track of every resettable class in the game. Upon being called the reset manager will go through every ressetable object and execute its specific action.

-Implementation: Implemented as a singleton class, every time a class implements a 'Resettable' interface they will be added to the resetmanager. This easily keeps track of every object that needs to be reset

without needing to check if it is an instance of. When the Resetmanager is run it will iterate through all the objects and run the excecuteInstance method. If it returns true, meaning the object has additional reset functions for the class, the method adds the object's parents to a set from which it later iterates through and runs their respective executeClass methods.

- Single Responsibility: Responsible for going through resettable each object and resets it if needed.
- Reduce Dependencies (ReD): relaxes the need to find specific objects for each resettable instance which reduces the dependencies needed.

ResetAction

- Purpose: A callable action that will allow for a game map to be reset. Will only reset the game if the actor has not reseted the game before.
- Implementation:. Upon being executed the action tests to see whether the actor calling the action does possess the “RESSETED” capability. Otherwise, if it doesn’t it will be added to the actor and have the ‘ResetManager’ run its reset method.
- Single Responsibility: Simple enforces a refresh of the current GameMap if it has not been reset before.