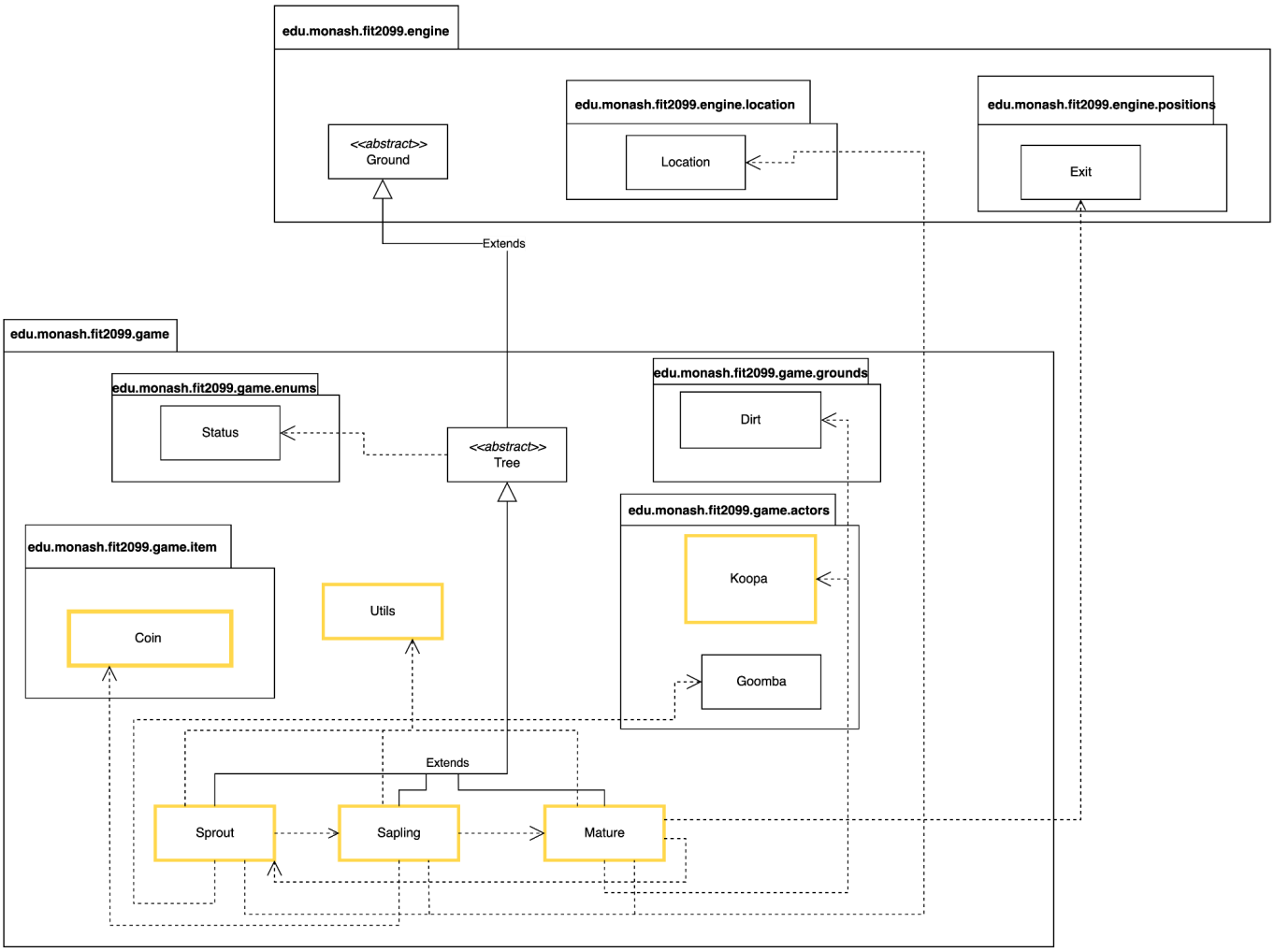
DESIGN

Note:

In the submission instructions in EdLesson, it also stated that we should have a separate text that outlines the overall responsibilities of the new classes (as shown in screenshot below). Therefore, we have added the overall responsibilities of the new classes or new/modified methods after the class and sequence diagrams.
Also, our generated index.html for javadoc is within the *docs* folder.

REQ1:

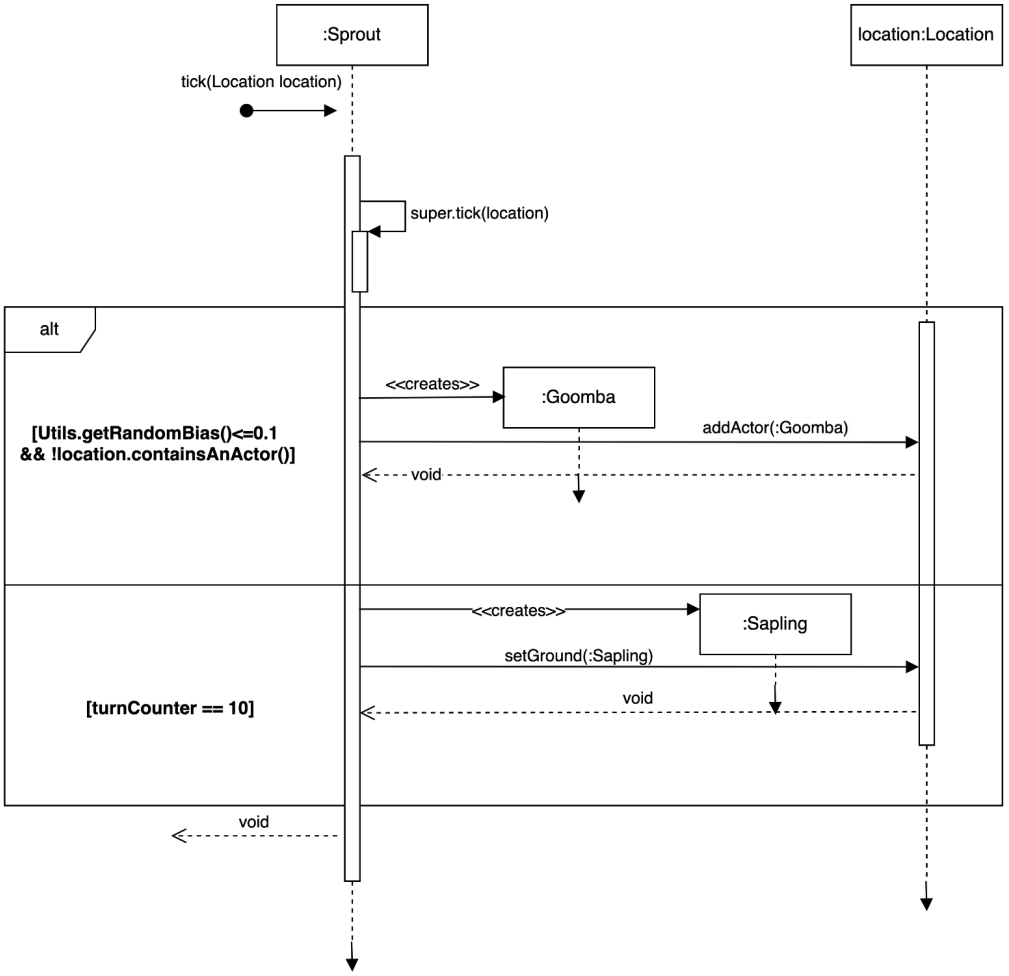
Class Diagram



Note: Only classes relevant to REQ1 have been shown here

Sequence Diagram

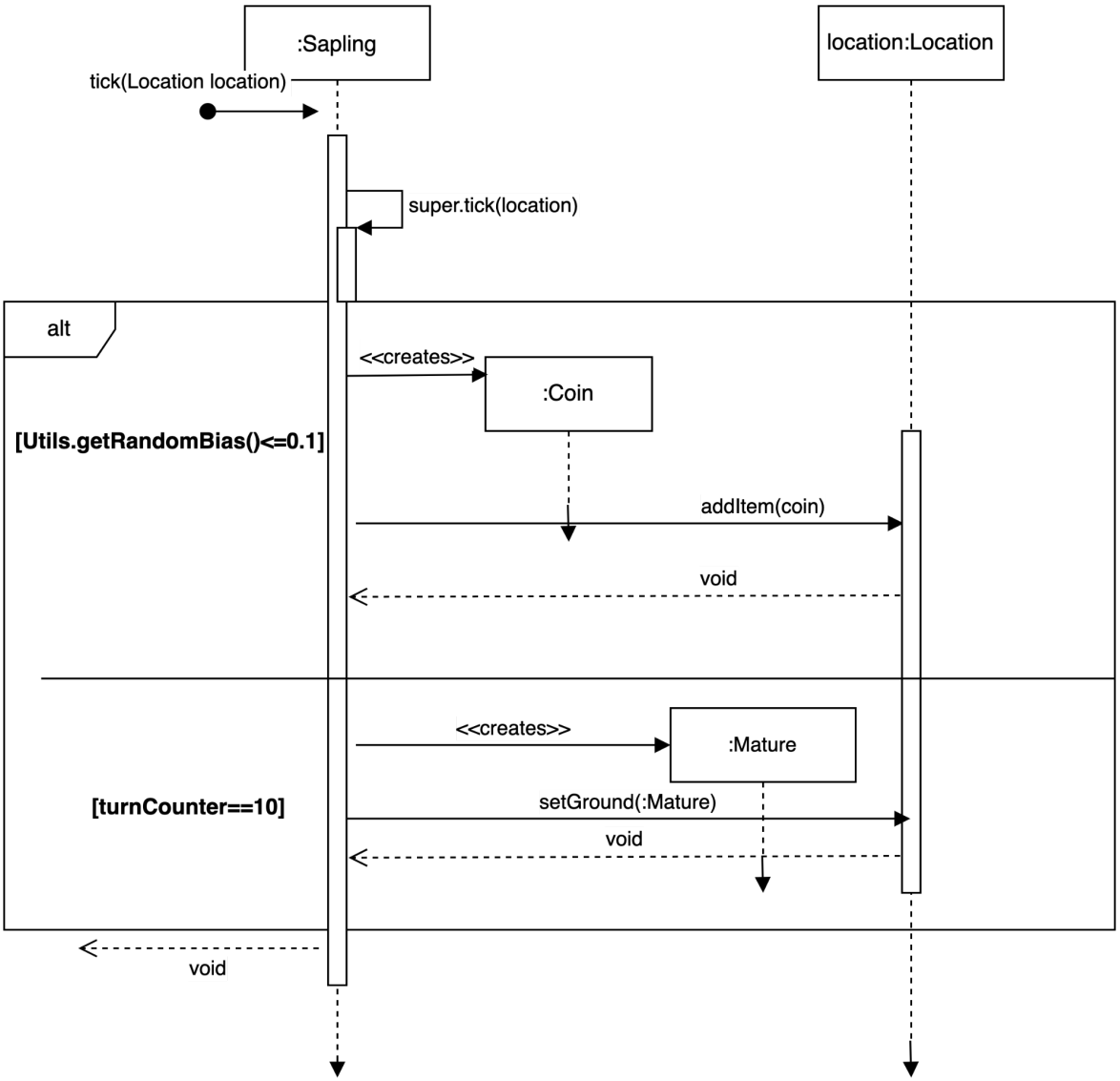
Sprout's tick() method's interaction was chosen for the following diagram :



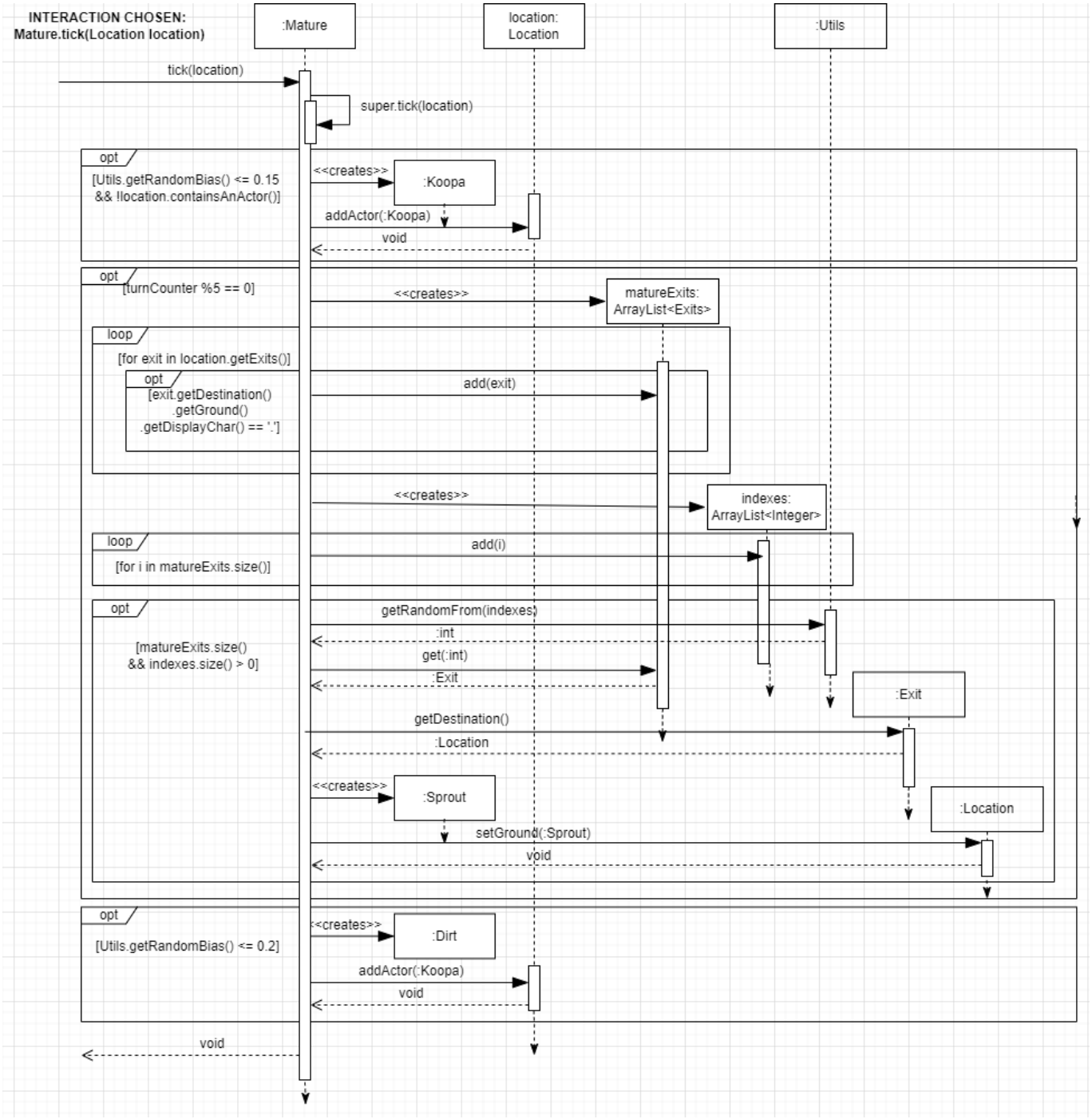
Note: tick() method 's sequence diagram for Sprout here, doesn't show the if statement for the resetting functionality, since for REQ1, we only focus on the stages of Tree and what each stage of Tree does in each turn.

Additionally, since this is a sequence diagram for Sprout's tick() method, it overrides from the super abstract class 'Ground' that it is a child of. Mature and Sapling also have an overridden tick() method and implement them differently according to the requirements, as shown in the next two sequence diagrams.

Sapling's tick() method interaction:



Mature's tick() method interaction



Tree Class

Note that the Coin class will be explained in more detail in REQ5 (Trading).

1. Class Overall Responsibilities:
Is an abstract class which implements the Resettable and HigherGround interfaces and thus, is linked to the functionality of resetting the game (as in REQ7) and having a 'jump' action (as in REQ2)

2. Relationship With Other Classes:
Is a child class of the abstract class Ground

3. Attributes:
protected final double success_rate;
protected final int damage;
protected final String name; This is the Tree's name.
protected int turnCounter; This is the counter of turns for the instance
protected ActionList actions;

The success_rate, damage & name attributes are final because they should not be modified after their initialization in the constructor. This is because an instance of a child class of Tree will not change their specific success rate, damage, nor name. Note that success_rate, damage, and name are more for Requirement 2 (JumpAction implemented)
Also, the above are protected attributes so that the child classes of Tree can have access to these attributes.

4. Constructor:
super(displayChar);
this.success_rate=success_rate;
this.damage=damage;
this.turnCounter=0;
this.name = name;
Resettable.super.registerInstance();

5. Methods:

i) The Ground's tick() method here is overridden to apply the functionalities of resetting the game. There is a 50% chance of all tree objects getting turned to dirt and so we spawn dirt on their current location.

ii) canActorEnter() method is overridden to pass 'false' if actor has a capability of MUST_JUMP and does not have the INVINCIBLE effect, which basically means that if actor comes across an impassable terrain they cannot pass it without jumping over it, it would return false.

iii) resetInstance() provides a capability for the Tree abstract class & its children to be able to be 'RESET' by the reset manager. (fore REQ7).

Note: The explanation for allowableActionsList's overriding is under the REQ2 section since that handles the jumpAction's methods and other functions.

Sprout Class

1. Class Overall Responsibilities:
Is responsible to handle the functionalities regarding the ‘sprout’ stage of Tree.

2. Relationship With Other Classes:
Extends the abstract super class Tree

3. Constructor:
super('+', 0.9, 10, “Sprout”);

4. Methods:

i) The Ground’s tick() method is overridden. First, we check if the RESET option was selected (which is done in super.tick() method from Tree class that it extends). If so, the game resets and applies the appropriate effects on the surroundings. Then, we check if the bias attained is less than or equal to 10% and make sure there is no actor in the player’s current location. If the preconditions are met, a Goomba is spawned.The turnCounter keeps track of the turns played and once it reaches ‘10’, a Sapling is put at the location of this Sprout, which replaces the Sprout and represents Sprout turning into a Sapling..

Sapling Class

1. Class Overall Responsibilities:
Is responsible to handle the functionalities regarding the ‘sapling’ stage of Tree.

2. Relationship With Other Classes:
Extends the abstract super class Tree

3. Construtor:
super('t', 0.8, 20, “Sapling”);

4. Methods:

i) Quite similar to the overridden method in Sprout class, we override the tick() method here and do a check if the player chose to ‘reset’.If so, the game resets and applies the appropriate effects on the surroundings.Then, we check if the bias attained is less than or equal to 10%.If this precondition is met, a Coin is spawned/added at the location of this Sapling object. The turnCounter keeps track of the turns played and when it reaches ‘10’ a Mature is put at the location of this Sapling, which replaces the Sapling and represents Sapling turning into a Mature.

Mature Class

1. Class Overall Responsibilities:

Is responsible to handle the functionalities regarding the ‘mature’ stage of Tree.

2. Relationship With Other Classes:
Extends the abstract super class,Tree

3. Constructor:
super('T', 0.7, 30, “Mature”);

4. Methods:

i) The tick() method here has a more involved functionality in this class. It again calls the super.tick() method which helps with the resetting functionality in Tree abstract class.
Then, we add a new actor ‘Koopa’ if the bias attained is less than or equal to 15% and if there are no actors in the current location of the now Mature tree.There is also 20% chance for the Mature tree to turn into dirt, i,e when it withers and dies, so we use: location.setGround(new Dirt()), to replace the Mature object with a Dirt object at that location.
Now, we implement the function of spawning ‘sprouts’ on fertile ground. So for every 5 turns, we initialise an arraylist of type Exit. Firstly, we need to find the exits around the tree object in this case in all four directions, i.e North,South,East and West, that have fertile ground (Dirt). Only these exits are stored in the matureExits arraylist.
We now initialise a second arraylist of integers for the sole purpose of adding indexes or markers of the dirt exits. This is useful so that we can now spawn a sprout in a *random* direction given it does have dirt since that’s the only fertile ground being considered at the moment.

Design Rationale

1. Tree, Sprout, Sapling, Mature
In our previous design, we used the existing Tree class to code all the functionalities. This was done by having an enumeration class called TreeCycleStage class which split the cycle stages and was used within every single method to set a new stage using the setters in the class.
Thus, we recognise that our previous design designed a God class where the Tree class handled a lot of checking and had too many responsibilities. This alternative we did therefore violated Single

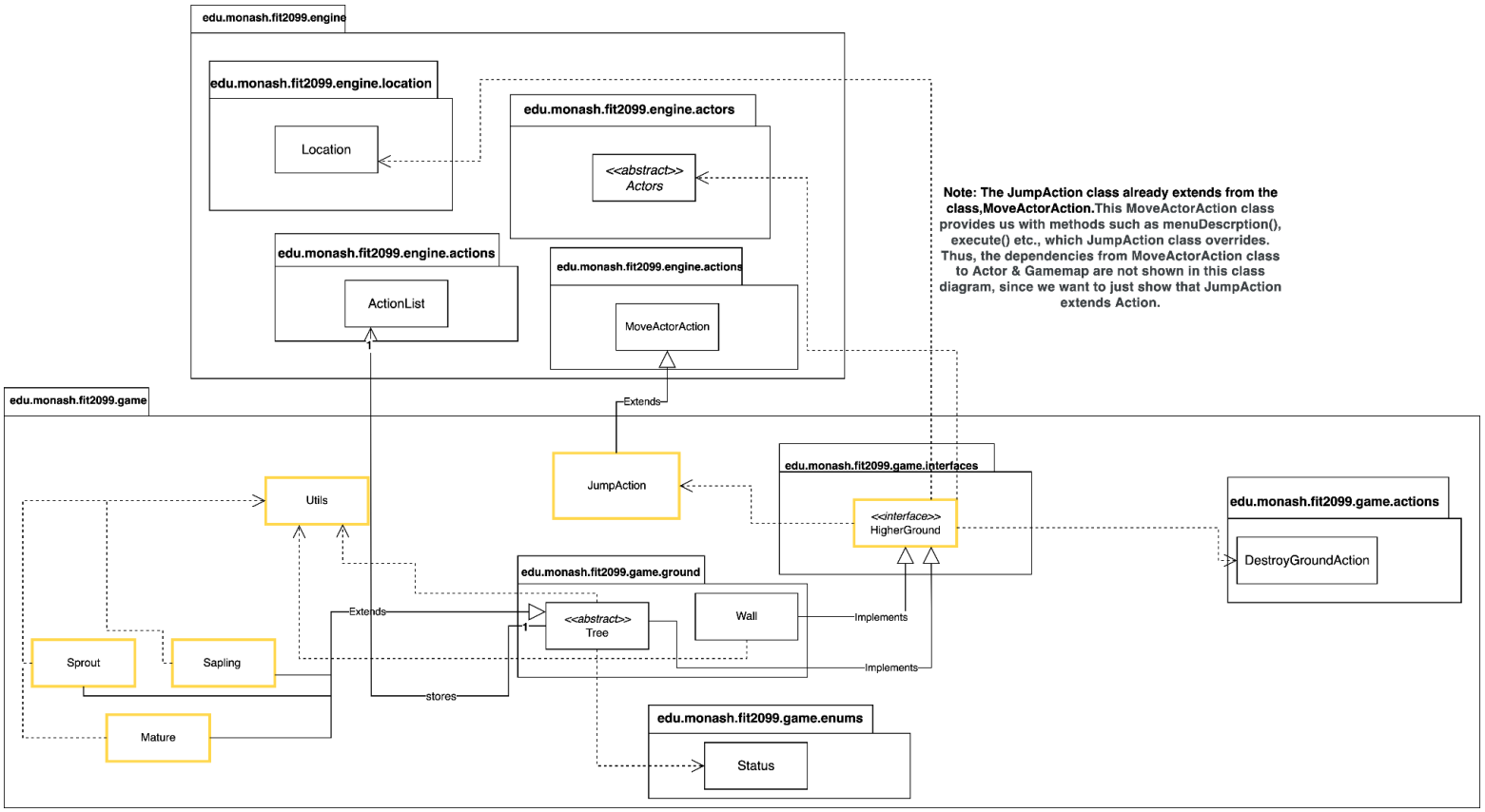
Responsibility Principle as Tree class was a God class, and violated Open-Closed Principle because to be able to extend the Tree class (for example having more stages for a Tree), the Tree class would need to be heavily modified to accommodate the added stages..

Therefore, in our new design, the Tree class is now made into an abstract class, where Sprout, Sapling and Mature class will extend this abstract Tree class. The reason why its made into an abstract class is because:
1) Tree class will never be instantiated on its own.
2) Sprout, Sapling and Mature share some of the same functionality (ie: need to be reset to Dirt when resetting (REQ7))
3) By having a Tree abstract class, we are able to adhere to the Single Responsibility Principle, as the different ways that Sprout, Sapling and Mature classes handle each turn (ie: tick() method) are implemented differently in *separate* classes, thus the responsibility is divided well.
4) We are able to adhere to the Open-Closed Principle, because if we want to add more stages to a Tree and extend the Tree class, we would just need to add a new class for that stage that extends this Tree abstract class. So, the Tree class would not need to be modified.

Therefore, the above rationale is why we implement Tree, Sprout, Sapling and Mature in this way.

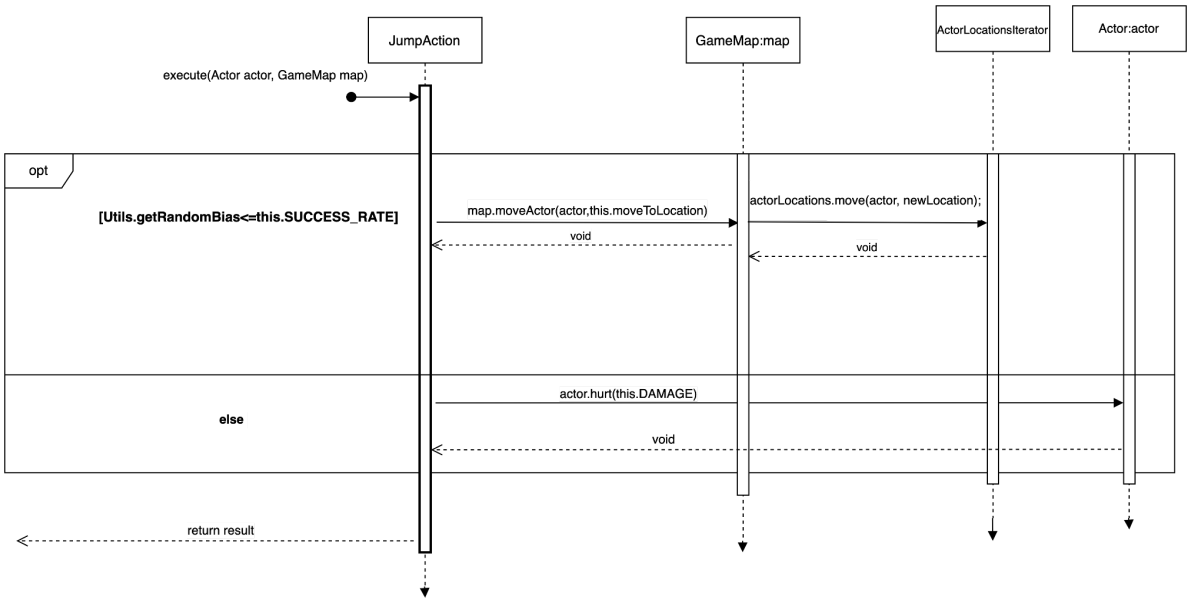
REQ2:

Class Diagram:



Sequence Diagram

JumpAction class's execute() method's interaction was selected:



Overall Class Responsibilities

Jump Action Class

1. Class Overall Responsibilities:
We added a new class called JumpAction which extends the Action class. The responsibility of this class is to allow the actor to jump whenever it is next to higher ground. We also account for the success rate of jumps across various objects on which the character can jump on.

2. Relationship With Other Classes:
Extends the abstract Action class since it is an additional action the player can perform.

3.Attributes:
private final Location LOCATION;
private final double SUCCESS_RATE;
private final int DAMAGE;
private final String DIRECTION;
These attributes are final because they should not be modified after their initialization in the constructor. This is because an instance of JumpAction will not change their specific success rate, damage, nor direction of the jump..

4.Constructor:
this.LOCATION = location;
this.SUCCESS_RATE = sucess_rate;
this.DAMAGE = damage;
this.DIRECTION = direction;

5. Methods:

i) (execute(Actor actor, Gamemap map) {};

Based on JumpAction's success rate, we will determine using Util's getRandomBias() function to see if it allows the actor to jump over to the high ground. If it does, then it will use the map GameMap variable to move the actor to the target destination with the moveActor() function. Otherwise, we implement the actor.hurt(damage) method and deal corresponding damage to the actor once they fail the jump.
In both cases, a suitable meaningful message is shown accordingly.

ii) Overridden the menuDescription() method from Action abstract class to show meaningful menu description for a JumpAction..

HigherGround Interface Class

1. Methods:
   ii) default implemented method for getMovementAction();

This class helps avoid the violation of Don't Repeat Yourself principle. This is to be explained more in the rationale below.

Additional Actions:
We have created a new default method of type MoveActorAction called getMovementAction(). In this method we initialize the MoveActorAction action object as null. Now we check the preconditions; first we make sure the actor is not jumping on its own location (this is checked for all the other preconditions and hence isn't repeated in the following ones),then check if he has the capability of 'MUST_JUMP' and 'INVINCIBLE', i.e when Mario has a Power Star and encounters higher ground. He has to destroy the higher ground he lands on, therefore,a DestroyGroundAction is instantiated and assigned to the action we first initialized.

Again, if Mario has capability of 'MUST_JUMP' but not 'TALL', i.e Mario does not possess a Super Mushroom and encounters higher ground, a new jumpAction object is instantiated where the success rate, failure damage,direction and getName() attributes are passed.

Lastly, if Mario has both the capabilities: 'MUST_JUMP' and 'TALL',i.e Mario has a Super Mushroom and encounters high ground, he gets to jump to higher ground with a 100% success rate, hence it is passed as '1' and no fall damage, which is why it's just passed as '0'.

In the Wall & Tree Class we override the ActionList's allowableAction and add the getMovementAction to the list to provide the player with the jump option when they indeed encounter a wall or a tree object by passing the success rate,damage,getName() and actor)

Player Class & Status Enum Class:
We modify the Status enum class to include the enum value MUST_JUMP, which signifies that the actor cannot walk over certain tall objects.
Thus, in the player class's constructor, we add MUST_JUMP to its capabilitySet. (This MUST_JUMP will be checked in HigherGround class)

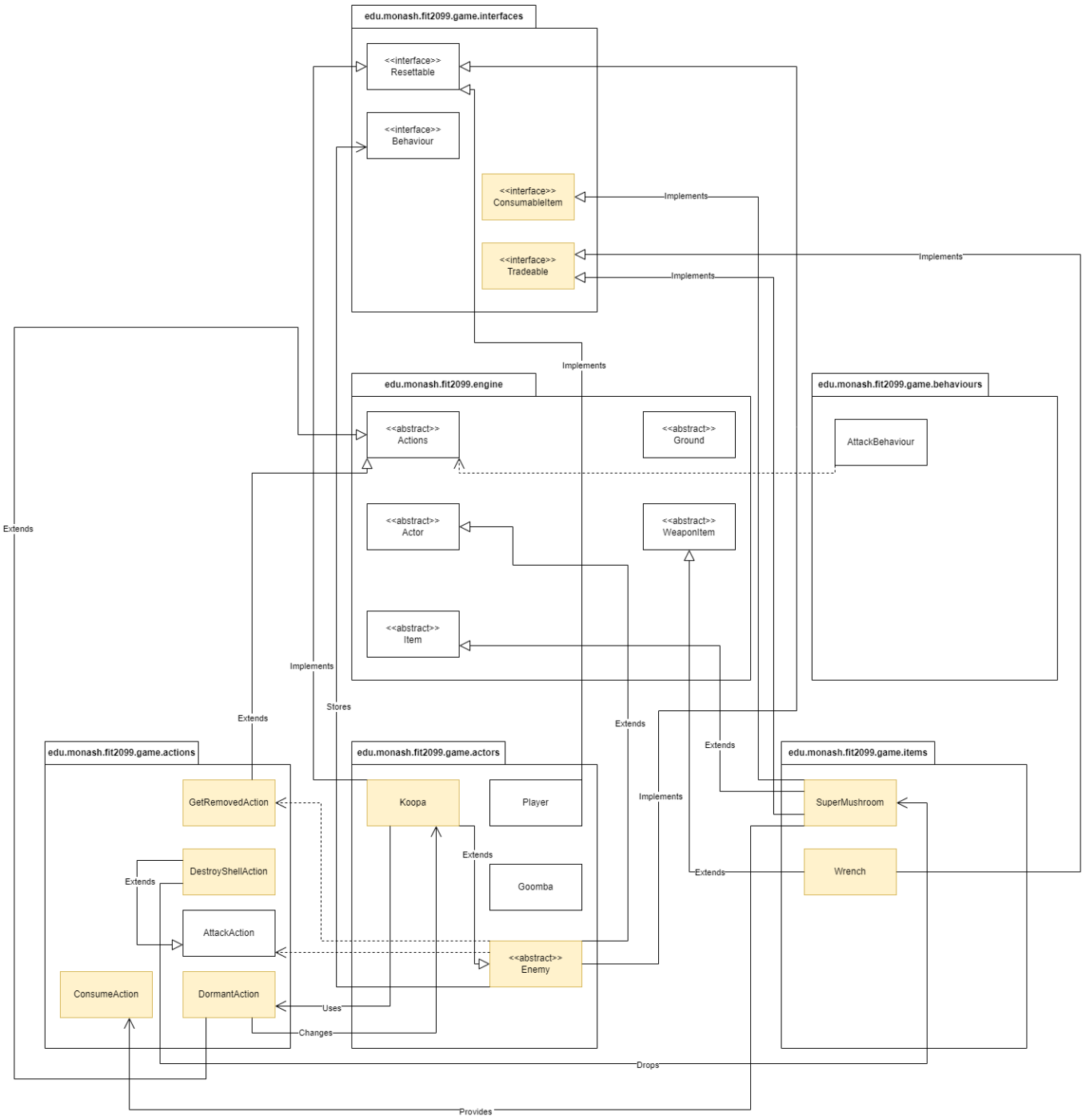
Design Rationale
For this requirement, we have decided to create a new class called Jump Action, and inherit the MoveActorAction class. This makes sense because the jump action is an addition to all the possible actions the game provides that a player can perform. To make the appropriate things work for this requirement we needed to make changes to a couple of other classes, namely: Wall & Tree.Additionally, we came up with a new interface called 'HigherGround' which provides a default method called getMovementAction.The Wall and Tree classes implement this interface since for the current implementation, these two objects are considered as higher ground & players would need a different way of moving instead of the normal MoveActorAction.

In our previous design (Assignment 1), the jumpAction class handled the checking of whether the actor had TALL/INVINCIBLE effects or not, which added dependencies. Now, the default method in the HigherGround interface handles this and will do so for any other interaction the player may have with higher grounds in the future which makes it extensible as new higher grounds can use this default method. This also follows the Single Responsibility Principle as when the jumpAction no longer has the responsibility of checking if the actor is INVINCIBLE/TALL to be able to execute the jumpAction.

We decided to set the final success rate and damage values for every growth stage objects within their respective classes so we can compare them to the random bias in the JumpAction class' execute method, and if only they match, the jump functionality will take place.

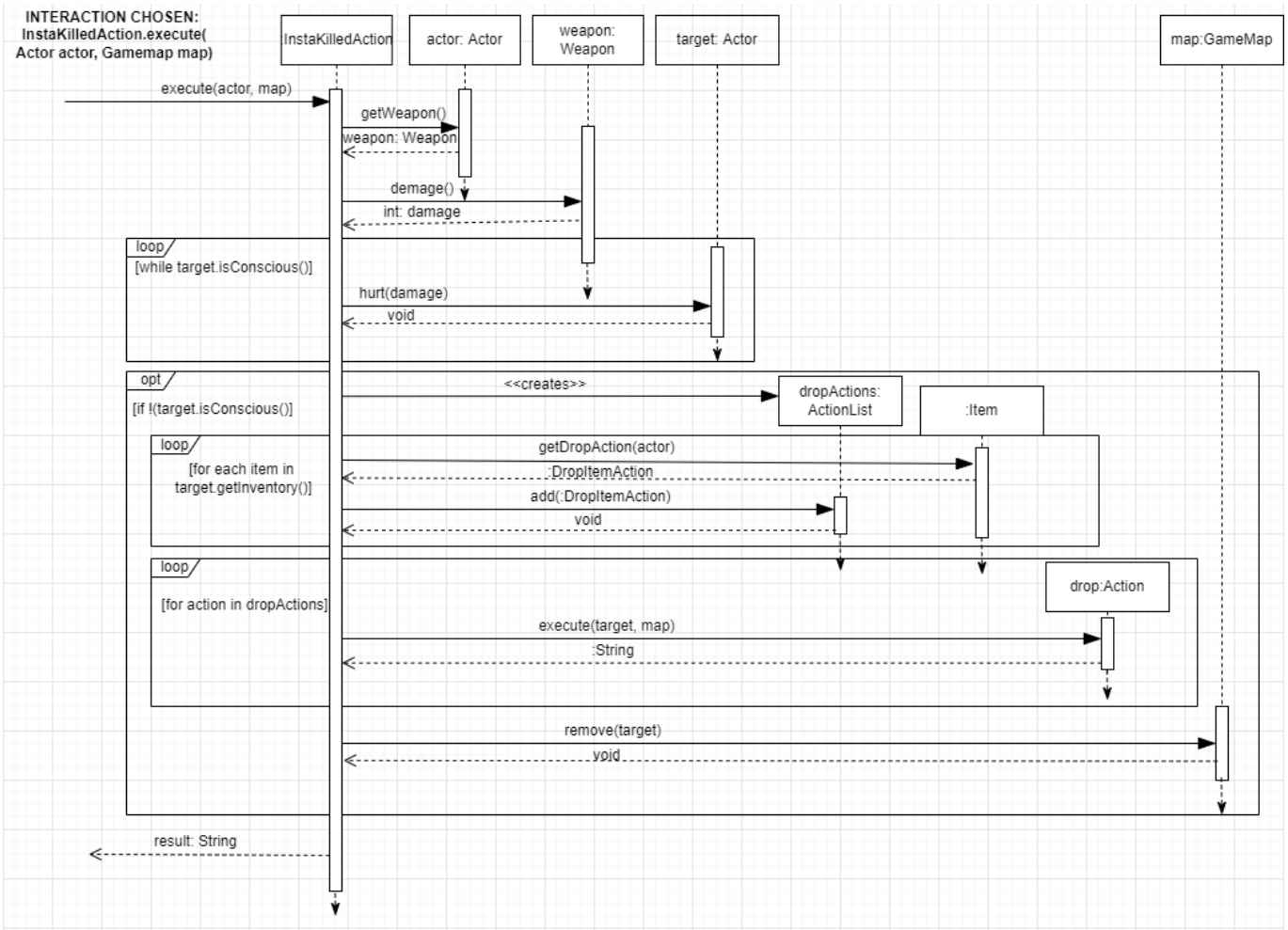
REQ3:

Class Diagram

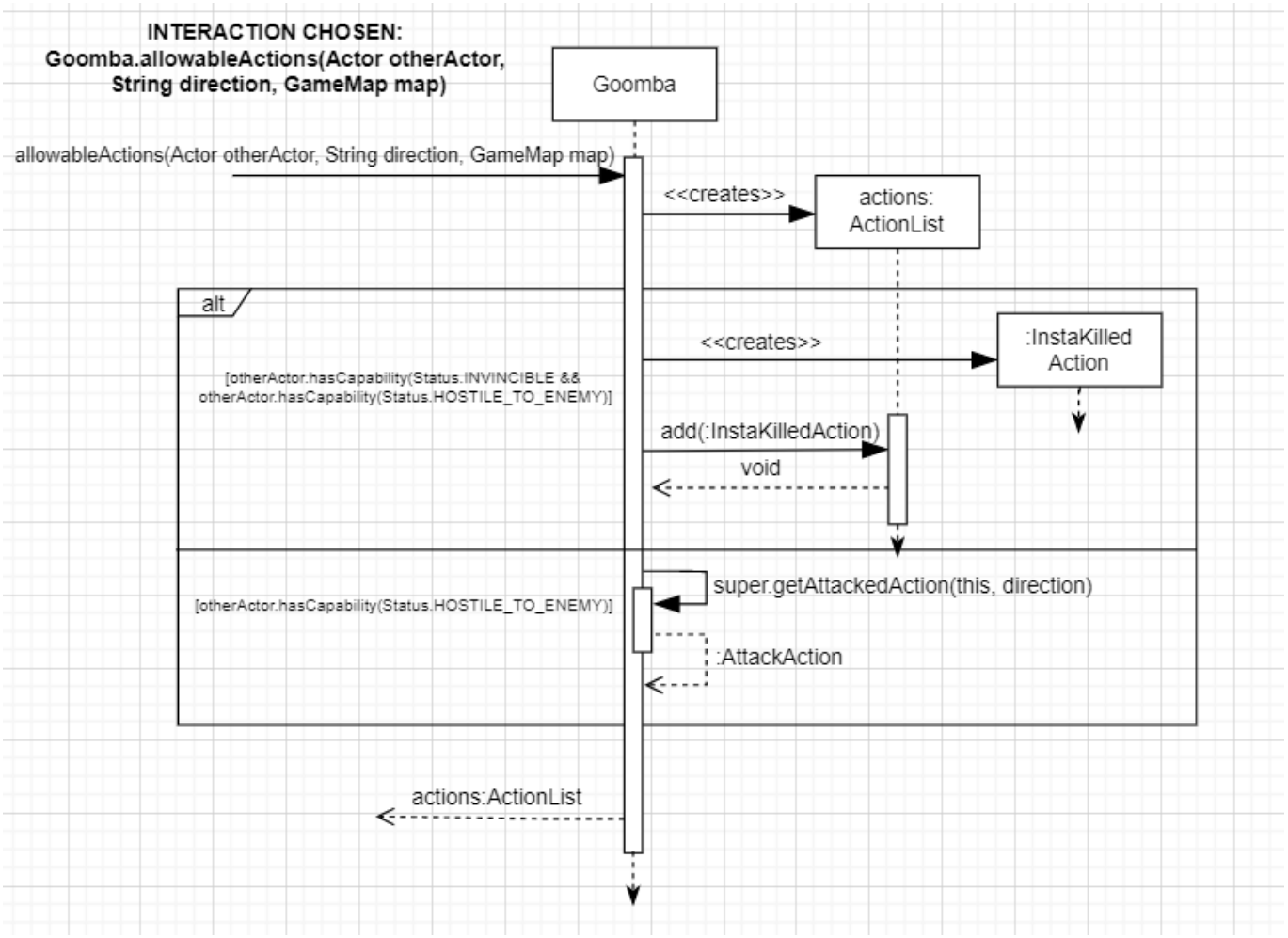


Sequence Diagram

The interaction chosen for REQ 3 is to show the interaction of InstaKilledAction.We choose this so that it can show how the player (actor) can use this to instantly kill the target enemies (target).



The second sequence diagram for REQ3 is the interaction for Goomba's allowableActions method, as shown below. We choose to show this so that it can show the interaction of how the Goomba that is near to the player, figures out which actions the player can have pertaining to the nearby Goomba.



Overall Class Responsibilities

REQ3
New classes :
SuperMushroom
Enemy abstract class
Koopa
DormantAction
DestroyShellAction
Wrench
GetRemovedAction

Updated/changed classes :
Updated Player
Updated Goomba
Updated AttackBehaviour
Updated AttackAction

<<abstract>> Enemy Class

1. Class overall responsibilities

This class serves as the baseline for all enemy types added to the game. Created to provide all necessary functions needed by enemies through its methods.

2. Relationships with other classes

Extends <<abstract>>Actor Class
Implements Resettable Interface
Association with Behaviour interface through attribute
Dependency on GetRemovedAction, AttackAction

3. Attributes

Protected HashMap to store behaviours

4. Constructor

Takes String name, char displayChar and integer representing HP of enemy as input.
Uses super to initialise enemy object through parent Actor class.
Gives enemy required capabilities and behaviours for basic function.
Wander behaviour given last priority in HashMap
Attack behaviour given highest priority in HashMap
ENEMY Status added to Enemy Actor's capabilities
Registers the instance of Enemy to the singleton ResetManager resettable List

5. Methods

OVERRIDE playTurn() Returns Action
Takes ActionList, Action, GameMap and Display instances as input.
Returns new GetRemoveAction instance if Enemy object has RESET capability, this removes the current instance from the game
Returns null otherwise

addFollowBehaviour() void
Takes target actor as input and adds a follow behaviour with second highest priority to the enemy object's HashMap

getAttackedAction() Returns Action
Takes target actor and String direction as input.
Returns a new AttackAction object using the given input.

OVERRIDE resetInstance() void
Adds the RESET capability from Status Enum Class to the Enemy object's capabilities.

Koopa Class

1. Class overall responsibilities

This class implements a new enemy type that will attack the player actor and follow it around after being attacked or when a hostile actor is in its vicinity. It will implement a new dormant interface that lets you know whether it is dormant or not. When knocked unconscious by the player and remain on the map until destroyed by a player with a wrench using DestroyShellAction. It drops a supermushroom object when its shell is destroyed. Behaves similarly to Goomba besides dormancy. Punches with 30 damage and 50% accuracy.

2. Relationships with other classes

Implements Resettable Interface
Extends <<abstract>> Enemy Class
Association with DormantAction through attribute

3. Attributes

DormantAction instance called dormantState

4. Constructor

Takes no input. Uses super with ‘Koopa’, ‘K’ and 100 to initialise the object and assigns dormantState attribute to a new instance of DormantAction

5. Methods

callSetDisplayChar() void
Takes char displayChar as input, calls the final protected method from the Actor <<abstract>> class to change the current display character of Koopa object

allowableActions() Returns ActionList
Takes Actor otherActor, String direction and GameMap map as input. Checks capabilities of player as well as koopa to return an ActionList of possible actions the player can carry out on the Koopa object. Current possible actions include Insta-killing, destroying the shell of dormant Koopa or just a normal Attack.

playTurn() Returns Action
Uses super.playTurn() first to see if Koopa has RESET capability, to know whether it should be removed. Then carries out checks on Koopa to see if it should be made dormant, or if it's already dormant. Will return dormantAction if unconscious and doesn't have DORMANT capability or returns DoNothingAction if DORMANT capability is preesent. Otherwise it goes on with normal behaviour, by going through HashMap and returning the first non-null action.

resetInstance() void
Adds RESET to Koopa’s capabilities, this allows it to be removed/killed in the next playTurn

hasDormancy() Returns boolean
True or False based on whether Koopa object has DORMANT capability

getWeapon() Returns Weapon
Returns intrinsic weapon of Koopa since enemies are currently not allowed to pick up items in current version of the game

getIntrinsicWeapon() Returns IntrinsicWeapon
Returns new intrinsic weapon object with damage value of 30 and String ‘punches’ to describe the attack.

SuperMushroom Class

1. Class overall responsibilities

To give player ability to jump freely and increase max health points. Buff lasts until damage of any form is taken. Needs to change display character of player. Execute methods of any actions that damage player actor have to be changed to check for TALL status. (For this requirement it has to drop when Koopa Shell is destroyed). Is shown on the map with ‘^’ character.

2. Relationships with other classes

Extends Item
Implements Tradeable, ConsumableItem
Association with ConsumeAction through attribute

3. Attributes

final int VALUE = 400, represents the cost for purchase of SuperMushroom.
final int healthIncrease = 50, represents amount for MaxHealth of player to increase by.
final Status buffStatus = Status.TALL, represents Status enum to be given to the player.
boolean isConsumed, to inform whether SuperMushroom has been used by player.
ConsumeAction consumeAction, consume action bounded to the Item so player can only use it once.

4. Constructor

Takes no input and uses super() with parameters "Super Mushroom", '^', true to initialise SuperMushroom from Item class. Sets isConsumed to false and consumeAction to null.

5. Methods

tick() void :
Ticker to update the superMushroom item when in player inventory.

Checks every turn to see if consumeAction should be appended to the Items ActionList based on whether the item has been consumed yet, through checking attribute. Removes consumeAction once the item has been consumed through the same checks.

tick() void :
Ticker to update SuperMushroom item when on ground. Does nothing besides carry out basic ticks for now, can be updated in future for removal after a long period of time so game does not get filled with SuperMushroom items.

getTradeAction Returns TradeAction :
Returns a TradeAction object with VALUE and SuperMushroom object as parameters

newInstance Returns Tradeable :
Returns new PowerStar instance for trading purposes.

getValue Returns int:
Returns cost VALUE for SuperMushroom

getDropAction Returns DropItemAction :
Takes Actor player as input and returns null for now.

getPickUpAction Returns PickUpItemAction :
Returns the super.getPickUpAction from Item class which allows player to pick up supermushroom from the ground.

consumedBy void :
Takes Actor (player) as input and increases health, gives capability and changes display characters. (Main functionality of the Item) Also sets consumed to true

setIsConsumed :
Takes boolean as input and sets the isConsumed attribute to said input value.

DormantAction Class

1. Class overall responsibilities
2. Relationships with other classes

Extends Actions<<abstract>> class
Association with Koopa through attribute

3. Attributes

protected Actor target, to store Koopa object
protected char newDisplayChar, stored a ‘D’, for when Koopa needs to change to dormant

4. Constructor

Takes as input Actor object (Koopa). Assigns input to attribute target. Also sets newDisplayChar to ‘D’.

5. Methods

execute() Returns String :
Adds DORMANT capability to the Koopa object and uses Koopa’s callSetDisplayChar() to change it’s display character. Returns a String to notify user that Koopa has gone dormant.

menuDescription() Returns String :
Returns null as user is already notified that Koopa has gone dormant.

DestroyShellAction Class

1. Class overall responsibilities

This class provides the player with a new action to destroy any dormant Koopa objects on the map. The action requires a wrench and has 100% accuracy. Drops a SuperMushroom object when action successfully carried out. Also stores damage, hit rate of the wrench and verb of the wrench.

2. Relationships with other classes

Extends AttackAction
Association with SuperMushroom through Attribute

3. Attributes

mushroomDrop, instance of SuperMushroom to be dropped for Player to pick up

4. Constructor

Takes inputs Actor target and String direction, uses super to initialise action and assigns mushroomDrop to a new SuperMushroom object

5. Methods

execute() Returns String :
Takes actor and GameMap as input. Uses location of actor to drop the SuperMushroom item then removes actor from the map. Will then return a String to inform user that he has successfully destroyed the Koopa shell.

menuDescription() Returns String :
Actor as input actor carrying out the action.
Returns a description of the attack, with actor carrying it out, target and direction of target from actor.

Wrench Class

1. Class overall responsibilities

A weapon for players to use against Enemies when attacking. To be stored in players inventory and can be purchased from Toad or picked up from map. Can be dropped and picked up by player as well. Has 80% hit rate and 50 damage.

2. Relationships with other classes

Extends WeaponItem
Implements Tradeable

3. Attributes

final static int VALUE, representing cost of purchase for Wrench item from Toad

4. Constructor

Has two constructors, one taking input and one without input.
Generally will use the no input constructor and use super to initialise with "Wrench", 'w', 50, "hits" and 80 representing name, displayChar, damage, verb and hit rate.
Other constructor will take input for String name, char displayChar, int damage, String verb and int hitRate and use super with input values.

5. Methods

getTradeAction() Returns TradeAction :
Returns new trade action with Wrench object and value as parameters

newInstance() Returns Tradable :
Returns new Wrench instance for trading purposes

getPickUpAction() Returns PickItemUpAction :
Adds HAS_WRENCH capability to the Player Actor.
Returns super.getPickUpAction() with actor as parameter.

getValue() Returns int :
Returns VALUE attribute

GetRemovedAction Class

1. Class overall responsibilities

For removal of actors, a description can be provided to the console through the game engine without having to make any changes to it.

2. Relationships with other classes

Extends Action

3. Constructor

No input constructor

4. Methods

execute() Returns String :
Takes actor to be removed as input and GameMap
Removes actor from map and returns menuDescription method.

menuDescription() Returns String :
Takes actor to be removed as input and returns a string notifying player that said actor has been killed/removed.

UPDATED CLASSES

Player

Now implements resettable
Has default hp final attribute

Constructor adds new capability MUST_JUMP

Has new callSetDisplayCharacter which calls the protected final setDisplayCharacter from parent actor class to change display Character.

Changes in playTurn to check for INVINCIBLE status

Goomba

Added a new static attribute called goombaCount which keeps track of how many Goombas are in the game. Checks with this number in the goomba's playTurn method() to see if it's more than a given number. If more then the goomba will be removed

For suicide, we used Utils in playTurn to roll for 10%. Removes if conditions are met, not if otherwise. This is done in playTurn()

Dependency on Utils for suicide of Goomba

New resetInstance() method to add RESET status to Goomba

Has an intrinsic weapon 'kick' that deals 10 damage using getIntrinsicWeapon() method.

AttackBehaviour

Dependency on actions

Implemented for loop to go through exits around Enemy actor, checks if player is present by checking for HOSTILE_TO_ENEMY status and creates new AttackAction against player

AttackAction

In execute,
Attacks now have to check if target does not have INVINCIBILITY capability before proceeding. Additional checks past this include checking for whether player has consumed SuperMushroom, will change display character if so. INVINCIBILITY check will result in message informing player that he's immune to enemy attacks.

When damaged, checks for whether target actor does not have HAS_DORMANCY status and unconscious, then removed from map

Further check to see if target is has ENEMY capability, follow behaviour will be added to Target object if true

String is returned at the end based on which condition is met during the attack.

Design Rationale

implementing dormancy
Initially for implementing dormancy we elected to use a TreeMap instead to store behaviours, and use a Dormant Behaviour class to make sure the Koopas did nothing when dormant. We found this to be quite redundant as there was already a DoNothingAction class for us to use in order to keep Koopas dormant and by doing that, we would make our code less consistent throughout and more difficult for newer programmers to understand without deeper knowledge and reasoning as to why since Goomba and uses HashMaps.

Instead we have now chosen to implement a dormant action bound to the Koopa so instead of having the Koopa behave like a god class by having all changes occur in the Koopa's playTurn() we had all functionality shifted into the DormantAction. DormantAction now adds capabilities to Koopa and changes its display character with methods given from Koopa. This more strictly follows the SRP principle than before as we split the functionalities of Koopa into different classes. We decided to make the action associated to Koopa and vice versa as every Koopa should be able to carry out the dormant action as an extension of itself when it exists.

We have also given all Koopas the HAS_DORMANCY status which prevents them from being removed from the map, so a simple check when attacking enemies is required to ensure that the dormant functionality is implemented. This also means AttackAction will not be responsible for handling when the Koopa is going dormant, instead the Koopa will be able to perform that action itself as an extension of its being. Koopa's playTurn() method is responsible for it going into dormancy, the AttackAction will not be required to make any changes to Koopa, this shows that the SRP principle is not violated and strictly adhered to.

suicide for goomba and handling spawns
For the suicide of Goomba, we decided the functionality would not require any sort of Handler for Goomba objects as it would create unnecessary associations or dependencies. We felt there was a simple solution by using Utils to generate a random number between 1-100 and providing a condition that it needs to be less than 10 to perform suicide by removing itself from the map.

Destroying Koopa Shells
By using a newly created DestroyShellAction class we extend AttackAction, this is because destroying a shell represents a type of attack action. This makes it fitting as both actions behave similarly and this aligns with the meaning of object-oriented programming.

This form of delegation allows us to code with both SRP and OCP as we delegate different types of similar attacks to seperate classes(SRP). In terms of Open Closed Principle, we are extending the AttackAction and not changing anything in the original class. We will also be able implement more types of attacks in future with ease as shown with the implementation of the InstaKilledAction class, which also extends AttackAction.

*** Replacing Enemy Interface with Enemy Abstract Class***
Initially we used an enemy interface to reduce dependencies between Koopa/Goomba and other classes but we found that having an Enemy Abstract class allowed us to have more functionality. For example we were able to allow the enemies to Attack and Follow the player by using the constructor of the abstract class to give each enemy the ability to carry out those actions from the get go. This was done by adding the behaviour to the behaviours attribute upon initiation.

With the interface we would be required to add such functionality into the constructor of every single new enemy we decided to add, this would make any programmers new to working with the game more prone to making mistakes when dealing with such large amounts of code. Even with methods to add such behaviours to each enemy with the interface, we would be required to call those methods upon initiation of each new enemy.

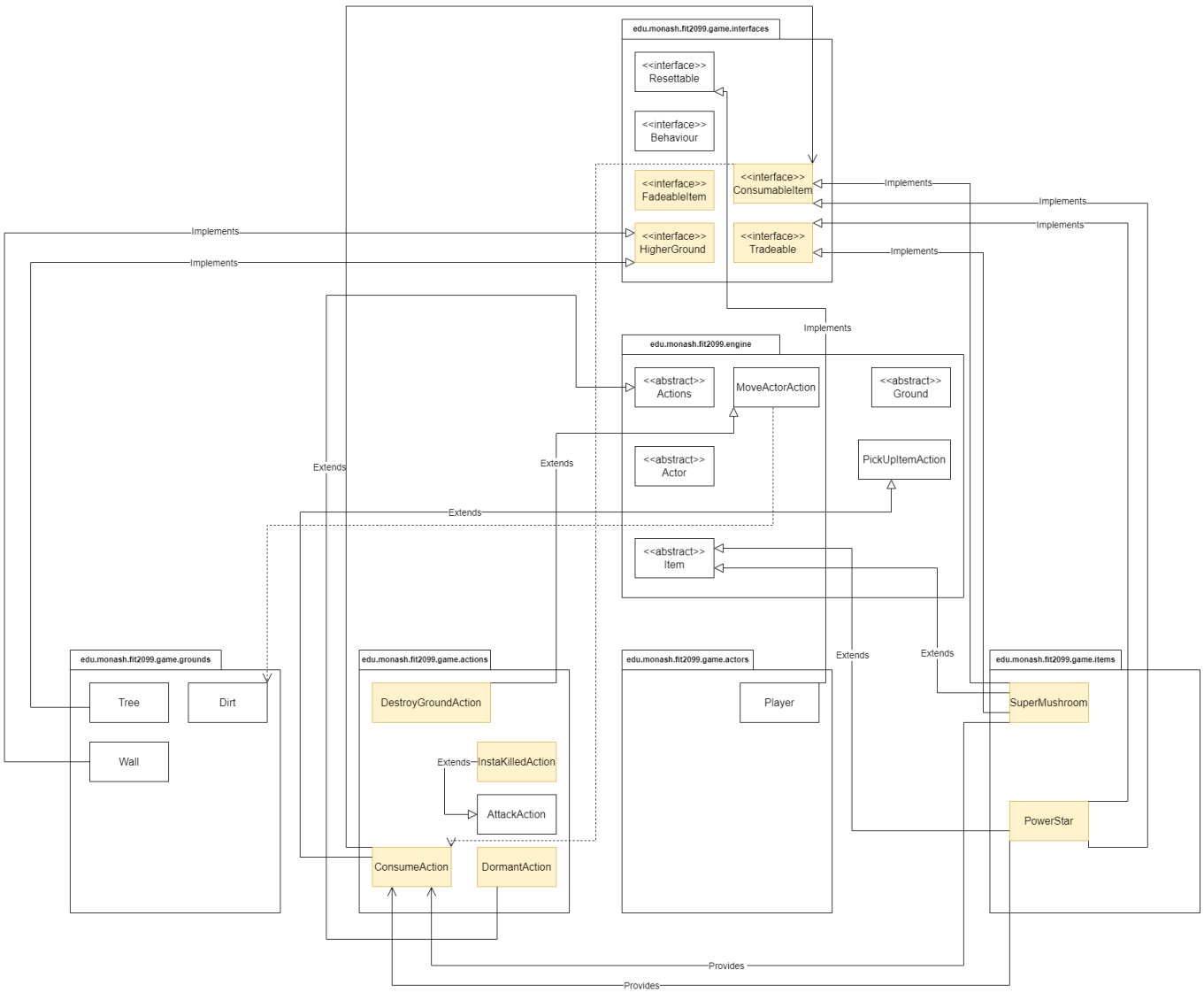
The addition of this class also allows us to add more functionality and behaviours to all enemies, thus greatly improving the future extensibility of our code, for example, adding behaviours/actions for enemies being able to jump. Besides providing new behaviours, we were also able to give a more solid foundation to each enemy we are able to implement and plan for changes much more easily. An example of the convenience this abstract class provides is the ability to reset all enemies by instantly registering them to the ResetManager in the constructor of one class instead of all Enemy objects. By making this change, we allow our code to strictly follow the Open Closed Principle, since all methods of the Enemy<<abstract>> class can be overridden if different functionality is required but nothing in the abstract class itself changes.

REQ4:

Note that for REQ4, we are not inputting the sequence diagram for it, as

1) We have explained in detail how the requirement should work in the below class responsibilities and rationale,
2) We have prepared all other complex sequence diagrams for other requirements, even including *multiple* sequence diagrams for a single requirement.

Class Diagram



Overall Class Responsibilities

New classes :

SuperMushroom
PowerStar
ConsumeAction
DestroyGroundAction
InstaKilledAction
ConsumableItem interface
FadeableItem interface
HigherGround interface

SuperMushroom Class

1. Class overall responsibilities

To give player ability to jump freely and increase max health points. Buff lasts until damage of any form is taken. Needs to change display character of player. Execute methods of any actions that damage player actor have to be changed to check for TALL status. (For this requirement it has to drop when Koopa Shell is destroyed). Is shown on the map with '^' character.

2. Relationships with other classes

Extends Item
Implements Tradeable, ConsumableItem
Association with ConsumeAction through attribute

3. Attributes

final int VALUE = 400, represents the cost for purchase of SuperMushroom.
final int healthIncrease = 50, represents amount for MaxHealth of player to increase by.
final Status buffStatus = Status.TALL, represents Status enum to be given to the player.
boolean isConsumed, to inform whether SuperMushroom has been used by player.
ConsumeAction consumeAction, consume action bounded to the Item so player can only use it once.

4. Constructor

Takes no input and uses super() with parameters "Super Mushroom", '^', true to initialise SuperMushroom from Item class. Sets isConsumed to false and consumeAction to null.

5. Methods

tick() void :
Ticker to update the superMushroom item when in player inventory.
Checks every turn to see if consumeAction should be appended to the Items ActionList based on whether the item has been consumed yet, through checking attribute. Removes consumeAction once the item has been consumed through the same checks.

tick() void :

Ticker to update SuperMushroom item when on ground. Does nothing besides carry out basic ticks for now, can be updated in future for removal after a long period of time so game does not get filled with SuperMushroom items.

getTradeAction Returns TradeAction :
Returns a TradeAction object with VALUE and SuperMushroom object as parameters

newInstance Returns Tradeable :
Returns new PowerStar instance for trading purposes.

getValue Returns int:
Returns cost VALUE for SuperMushroom

getDropAction Returns DropItemAction :
Takes Actor player as input and returns null for now.

getPickUpAction Returns PickUpItemAction :
Returns the super.getPickUpAction from Item class which allows player to pick up supermushroom from the ground.

consumedBy void :
Takes Actor (player) as input and increases health, gives capability and changes display characters. (Main functionality of the Item) Also sets consumed to true

setIsConsumed :
Takes boolean as input and sets the isConsumed attribute to said input value.

PowerStar

1. Class overall responsibilities

To give player ability to walk freely and heal player by 200 health points. Buff lasts 10 turns in game. Makes player invincible to damage. Allows player to use InstaKilledAction and DestroyGroundAction when having status and walk over walls and trees. Has to convert walls and trees into dirt and drop a coin ($5). Ticks on ground and in inventory up till 10 turns before it disappears.

2. Relationship with other classes

Extends Item class
Implements Tradeable, ConsumableItem, FadeableItem

3. Attributes

final int VALUE = 600, represents the cost for purchase of power star.
final int HEALTH_HEAL_AMT = 200, represents amount of health for player to heal by.
final Status buffStatus = Status.INVINCIBLE, represents Status enum to be given to the player.
boolean isConsumed, to inform whether power star has been used by player.

ConsumeAction consumeAction, consume action bounded to the Item so player can only use it once.
private int fadingTimeOnFloorInventory, fading ticks when on ground
private int fadingTimeOnPlayer, fading ticks when in player inventory

4. Constructor

Uses no input and takes super with "Power Star", '*' and true to initialise powerstar item. Sets fading attributes to 10, isConsumed to false and consumeAction to null.

5. Methods

canFade() boolean :
Implementation of Interface method that returns true

getHealthHealAmt() int :
Getter for heal amt attribute

getBuffStatus() Status :
Getter for buff status attribute

getFadingTimeOnFloorInventory() int :
Getter for fading time on floor/inventory attribute

getFadingTimeOnPlayer() int :
Getter for fading time when consumed attribute

getIsConsumed() boolean :
Getter for isConsumed attribute

setIsConsumed() void :
Setter for isConsumed attribute

setFadingTimeOnPlayer() void :
Setter for fading time in inventory attribute

tick() on floor : void
Decrements floor/inventory fading attribute by one and checks if less than 0, removes if condition met

tick() in inventory : void
Decrements inventory fading by one and checks if player has invincibility status and consumption of PowerStar to know whether consumeAction should be removed or added to the Items allowable actions.

Removes item from inventory if not consumed and fading attribute less than 0
When consumed and attribute for on player is less than 0 item is removed from inventory

consumedBy() : void
Takes consumer actor as input and makes changes to health according to attribute. Adds INVINCIBILITY capability and sets consumed to true.

ConsumeAction

1. Class overall responsibilities

To provide an action for players to consume ConsumableItems and print an informative message on the console to notify when said action is carried out.

2. Relationship with other classes

Extends PickUpItemAction
Association with ConsumableItem

3. Attributes

consumableItem to store the item to be consumed

4. Constructor

Takes an instance of ConsumableItem and uses super to initialise the action. Binds the item to the consumableItem attribute

5. Methods

execute() String :
Requires actor and GameMap as input
Uses consumedBy() method of the item to make the changes necessary on the actor.
Removes item from actors inventory after consumption and returns a String to notify player that item has been consumed

menuDescription() String :
Requires actor as input
Returns String to let player know he has consumed super mushroom or powerstar, shows how many turns left on powerstar if powerstar is consumed.

DestroyGroundAction

1. Class overall responsibilities

To move the player to a specified location and ignore walls and trees on the map when the player has consumed a PowerStar. Has to also drop $5 coin after a wall or tree is destroyed

2. Relationship with other classes

Extends MoveActorAction
Dependency on Dirt

3. Attributes

hotKeyMap

4. Constructor

Takes location and String as input and uses super to initialise action

5. Methods

execute() String :
Takes Actor actor, GameMap map as input and moves actor to the given location.
Sets the given location to new Dirt instance and adds $5 coin on the location
Returns menuDescription

menuDescription() String :
Returns string to inform console that player has moved to specified location and destroyed the obstruction and turned it into dirt.

hotkey() String :
Returns hotkey from attribute based on given direction.

InstaKilledAction

1. Class overall responsibilities

Class provides player with the ability to instantly kill any enemy it comes across on the map when actively having the INVINCIBLE status.

2. Relationship with other classes

Extends AttackAction

3. Attributes

None

4. Constructor

Takes actor and direction for attack and uses super to initialise the action.

5. Methods

execute() String :
Takes actor and direction as input
Gets weapon from actor input and damage of weapon, uses while loop to damage actor until it is unconscious
If target cannot go unconscious it is removed since Koopas can only go unconscious
Returns String describing death of target of attack.

ConsumableItem Interface
Default method to return consume action to player to remove dependency on consume action
consumedBy method for all items to implement
boolean canFade() method to determine whether item will have to disappear or not

FadeableItem Interface
Method to return fading time for items that can fade to implement

HigherGround Interface
Has default method return appropriate actions (JUMP freely or Destroy ground or normal) based on Status of player

Design Rationale
REQ4
consuming items
Initially we planned on using the Player class' playTurn() method to loop through their inventory and consume items that were instances of ConsumableItem. Through further study of the code, we found that creating a ConsumeAction to add to the item's allowableActions was the better choice. So instead of making the player responsible for looking through items and checking for PowerStar or SuperMushroom, we now leave it to PowerStar to provide that action to the player as we feel it adheres more to the Single Responsibility Principle since the player is not responsible for looking for the item. It would also make the code perform better since we need not loop through the inventory at each turn (troublesome and performance-heavy with large amount of items in inventory).

We have opted to use a consumedBy() method in the ConsumableItem interface to ensure all consumable items that are added in the future have to implement the method. The consumedBy() method also makes the item itself responsible for providing the player with any necessary changes. We believe this follows the SRP as it makes sense for the Item itself to be responsible for making changes to the player, as without that it would merely be an functionless class storing the type of changes and amounts to change without actually doing so.

We believe having the item remove itself from the player's inventory to be the violation of SRP in this case. Therefore, we delegated said task to ConsumeAction, this makes it responsible for calling consumedBy() and removing the item after consumption which we believe better aligns to the definition of consumption.

Having consumeAction be provided to the player by the item also allows us to constantly provide the ability to consume an item whenever present in their inventory. Without the ConsumeAction, we would have had to check for the individual items in the inventory to see if they were instances of SuperMushroom or PowerStar and manually consume them, shifting the responsibility to the Player.

With the ConsumableAction, we do not need any checks whatsoever.

I believe the way in which we coded ConsumeAction and added a ConsumableItem Interface also allows us to follow the Open Closed Principle of design. This is because from now onward, whenever adding a new ConsumableItem, we would simply have to implement the consumedBy() method in the newly added item without having to change any code in ConsumeAction and ConsumableItem.

Since this implementation aligns with two principles of design, we have deemed it to be good design for our game code.

The act of consuming Magical Items provides players with new actions.
InstaKillAction. The player will also be able to jump freely over walls and trees as well as destroy them if they consumed a PowerStar.

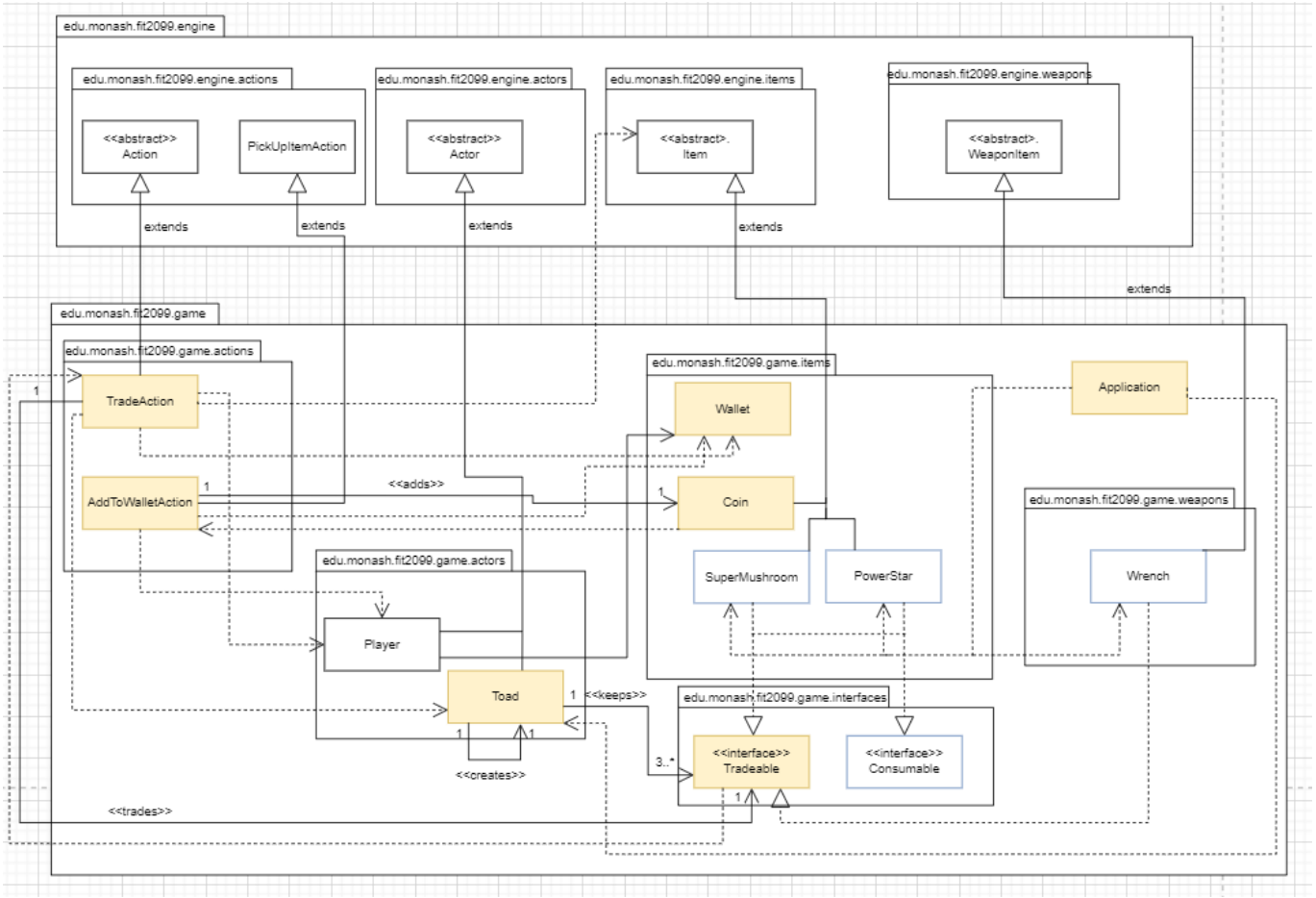
This is implemented by having checks in the Goomba and Koopa classes allowableActions() method. The if statement will check the player capability set to see if it contains INVINCIBLE and returns an ActionList instance with InstaKillAction if present.

We have decided to stick to what we originally planned for implementations of InstaKilled and DestroyGroundAction, which was to use allowableActions of Enemy and a HigherGround Interface to provide Players with the necessary actions to carry out when they have the right statuses.

The main change being that we wanted to implement destroy ground by having Wall and Tree detect when a player has INVINCIBLE status in its allowableActions() method, however we did not feel that Wall and Tree needed to have that responsibility. Having this interface also allows us to not need to rewrite the code in allowableActions() everytime we add a new type of ground, this follows the Open Closed Principle and we felt it was the best way to implement the functionality. Everytime we add a new type of HighGround that requires jumping, for example barbed-wire or a barricade, or a pipe in Mario's case, we would already have the default method handle all required functionality. Because of OCP and the potential for future extensibility, we opted to make use of this design.

REQ5:

Class Diagram

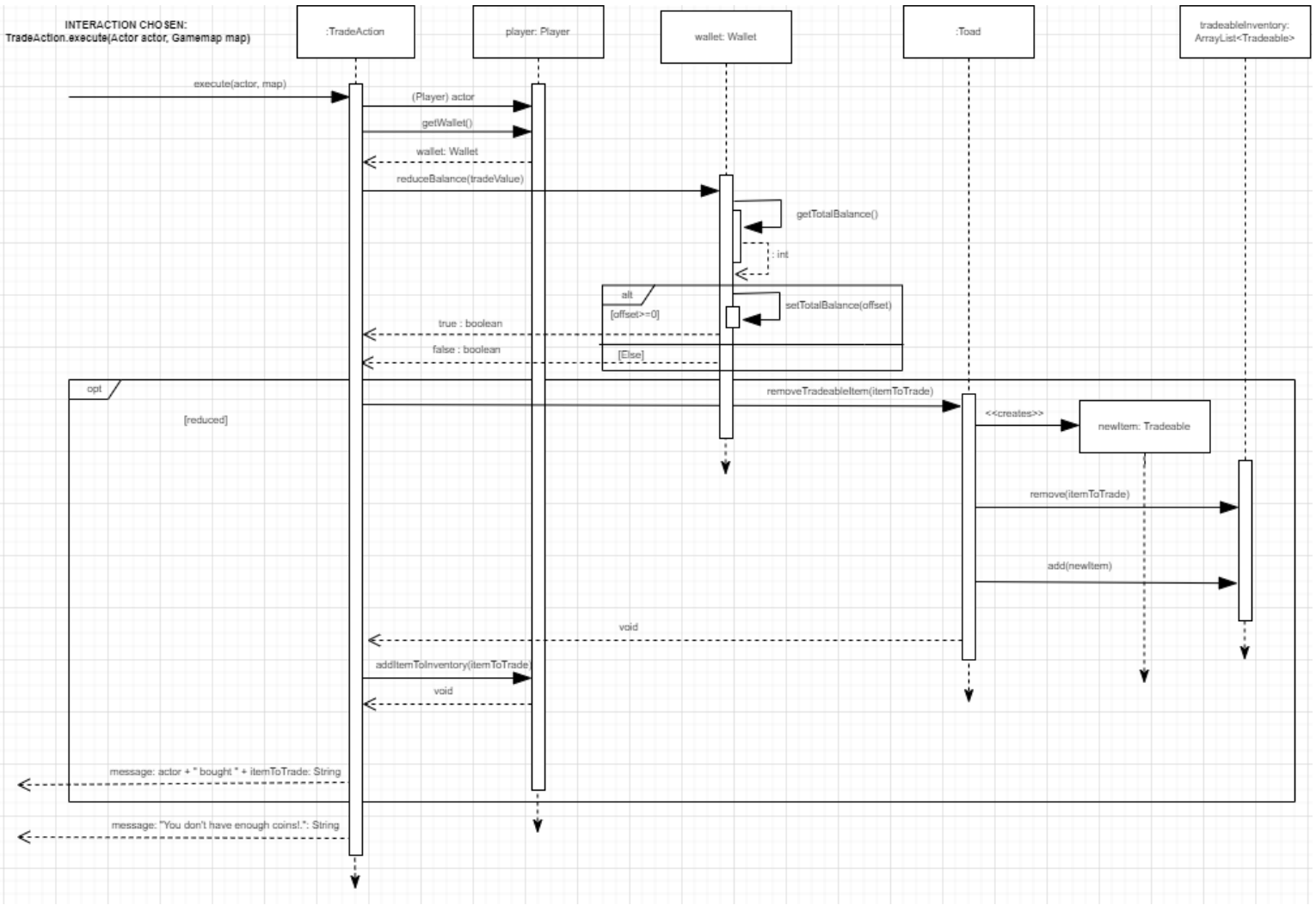


New classes added for REQ5:
Wallet, TradeAction, Coin, AddToWalletAction, Toad, Tradeable.

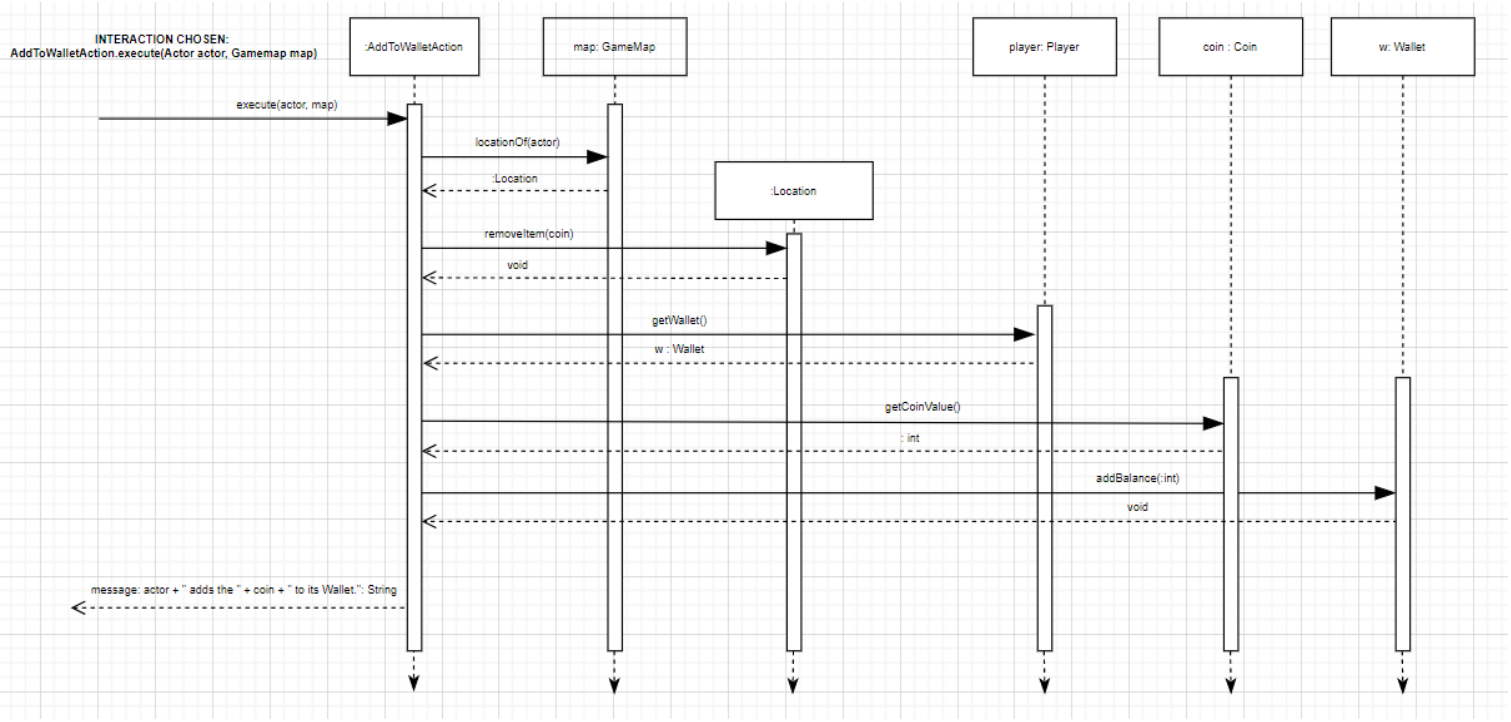
Note in the UML Class diagram for REQ5 that the yellow highlighted classes represent the modified/added classes for REQ5 specifically. Classes that are black represent classes existing in the base code. Classes that have other colours (other than black & yellow) are classes added/ modifed for other REQs.

Sequence Diagram

The below sequence diagram for the TradeAction's execute interaction.



The below sequence diagram is for the AddToWalletAction's execute interaction.



<u>Overall Class Responsibilities</u>

New classes added for REQ5:
Wallet, TradeAction, Coin, AddToWalletAction, Toad, Tradeable.

Player

1. New attributes:
private Wallet wallet; - This is so that we can get an immediate reference to its wallet, even when the wallet has been added to the inventory.
private final int DEFAULT_HP; - This is for the DEFAULT_HP that every player starts off with.

2. methods added:
getWallet() to return its Wallet from its inventory.

3. methods modified:
playTurn() to print the player's position in the map, its hp, and its wallet balance at each turn.

Wallet

1. Class Overall Responsibility:

This class is a class that is used to implement a wallet system where only the player can have an assigned wallet item. It is used to get/reduce/add the total balance that the player has during the game.This assigned Wallet class is a final attribute within the player class, since it is fixed and cannot be removed from every player.

2. Relationship with other classes:

Inherits from Item abstract class (extends this class).

3. Attributes:
private int totalBalance,

4. Constructor:
Creates instance of Wallet item.

5. Methods:
private getBalance() to return the balance
private reduceBalance() to reduce the balance
private addBalance() to add to balance

TradeAction
1. Class Overall Responsibility:
This class is used to implement an action where each player can use their wallet balance to purchase an item from Toad actor.

2. Relationship with other classes:
Inherits from Action abstract class (extends this class).
Has an association with Tradeable interface.
Has a dependency on singleton Toad class & Player class.

3. Attributes:
private Tradeable itemToTrade;
private int tradeValue;

4. Constructor:
Creates an instance of tradeAction with a reference to the item to be traded, and the trade amount/value.

5. Methods:
overrides Action's execute method.
overrides execute() method:
- Retrieves wallet from Player
- Remove balance from Player's wallet
- Removes item from Toad's inventory
- Adds the item that was purchased to player's inventory

Coin

1. Class Overall Responsibility:
This class is used to represent a Coin item (that is spawned from the Sapling class), that can be added to the Wallet item of a player.

2. Relationship with other classes:
Inherits from Item abstract class (extends this class).
Dependency on AddToWalletAction.

3. Attributes:
private int coinValue,
String name = 'Coin', String displayChar = '$', Boolean portable = false
(the ones above are inherited from Item abstract class)

4. Constructor:
creates an instance of Coin with the value of the coin attached to it.

5. Methods:
overrides Item's getPickItemAction() method.
overrides getPickItemAction() method:

- instantiates AddToWalletAction object *instead of* a PickUpItemAction, because PickUpItemAction adds the item to the inventory of the player. Thus, AddToWalletAction will help to add the coin's value to the Player's wallet instead of its adding the coin object to its inventory. AddToWalletAction also removes the coin object from the map.
- returns the AddToWalletAction object

AddToWalletAction

1. Class Overall Responsibility:
This class is used to represent adding coins to Wallet Action. Note that this class will only ever be used for adding coins to the wallet. The coin object is not directly added to the wallet, but rather their coinValue will be used to add to the totalBalance of the wallet. The reason why is that our requirements don't require us to use coins in any other way except for adding value to our wallet.

2. Relationship with other classes:
Inherits from PickUpItemAction class (extends this class).

3. Attributes:
Coin coin; The reason why we store another Coin item references, is because the item attribute put into super is a private final attribute in PickUpItemAction class, and therefore the child class AddToWalletAction class cannot access that attribute.

4. Constructor:
creates an instance of AddToWalletAction with the coin item attached to it.

5. Methods:
overrides PickUpItemActions's execute method

overrides execute() method:

- access the Actor's wallet and add the coin item’s value to the Actor's wallet item.
- remove the coin from the ground it was picked up from.

Toad

1. Class Overall Responsibility:
This class is a singleton class since each game instantiated should only ever have one Toad. It is responsible for representing a Toad who is friendly (can talk to players) and who sells several items (Wrench, SuperMushroom, PowerStar) to players.

2. Relationship with other classes:
Extends Actor abstract class.
Has association with Tradeable interface class.
Has dependency on PowerStar, SuperMushroom and Wrench class.

3. Attributes:
private static Toad instance;
public ArrayList<Tradeable> tradeableInventory;

4. Constructor:
A private constructor that creates an instance of Toad, and instantiates PowerStar, SuperMushroom & Wrench items which are added to its inventory arraylist of items.

5. Methods:
getInstance() factory method for instantiating a new instance if it hasn't yet been instantiated, else it returns the first and only instantiated Toad instance.

getTradeableItems() method for returning an arraylist of items Toad can trade.

removeTradeableItem(itemToTrade) method for
- removing the itemToTrade from the Toad's tradeableInventory list, and
- adding a new instance of the type of that item traded into the Toad's tradeableInventory list.

allowableActions() method for:
- Adding a TradeAction for each tradeableItem
- Adding a TalkWithToadAction

to the actions the otherActor interacting with it can have.

Tradeable

1. Class Overall Responsibility:
This class is an interface that represents the functionality of an item being tradeable.

2. Relationship with other classes:
Has a dependency on TradeAction
Wench, SuperMushroom and PowerStar (as of now only these 3) implements Tradeable.

3. Methods:
getTradeAction();
This is to get the TradeAction of the current Tradeable item object.

getValue();
This is to get the value of the trade.

newInstance();
This is to get a newInstance of its class, so that when a Trade happens, the seller can replenish his inventory without having to be dependent on the individual traded item.

Design Rationale

1: Purchasing an item: TradeAction, Toad, Wallet class

TradeAction
tradeAction extends Action. This allows for the action to purchase an item to be done.
We want tradeAction to inherit from Action, so that there is an "is-a" relationship that would allow for reusability of code & extensibility of tradeAction class.

Wallet class
In our previous design, the Wallet extended the Item abstract class and also had the responsibility of adding the tradeActions as allowable actions the player can have. However, this alternative violates the Reduce Dependency Principle as it increased the dependencies of Wallet class to Toad, Tradeable & TradeAction class. Also, it violated the Single Responsibility Principle because Wallet should only be responsible for interactions pertaining to the total balance of the Player and thus the inherited methods from Item (if Wallet did extend it) such as tick() methods are redundant as they are not needed for the functionality of a Wallet object. Therefore, we have removed this part of the design and instead Wallet is now implemented as a class by itself without extending Item, and TradeActions are now handled by the Toad class.

Toad
Toad class is the one that would now handle the tradeActions between the player and itself.
Toad stores a tradeableInventory, and for each tradeableItem, Toad adds an instance of TradeAction for that item, to the actions that the player can have by interacting with this Toad. This is in line with the Single Responsibility Principle where only the related class (Toad) has the responsibility of providing the player with a TradeAction.

Therefore, our final design has now aligned with the Reduce Dependency Principle and Single Responsibility Principle. The application checks for actions allowed by the nearby Toad instance, Wallet and Toad classes now have reduced dependencies, and Toad and Wallet now have separate responsibilities (previously they shared the responsibility of handling trades & so had depended on each other).

2: Coin, Wallet and AddToWalletAction

In our previous design, the wallet item had an arraylist of stored Coins that the player picks up. However, since the wallet item does not actually require the storing of coins to keep track of the balance the player has, having an arraylist of coins would be an added association that the Wallet item class has and therefore is not needed. Instead, in the revised design, the wallet only keeps track of the total balance the player has.
Thus, how we represent adding coins to a wallet is this: a coin's value is added to the Wallet of the player to increase the balance of the wallet, and the just used coin is removed from the map. So, when the player is picking up a coin item from the ground, they will not perform the PickUpItemAction, as this action will add the coin into the players inventory. Instead the AddToWalletAction that extends PickUpItemAction will be the action done when the coin item is picked up.

The reason why we do this is because:
1) We want only the Wallet system to have the responsibility of dealing with coin values/money of the player. The player should not directly be taking care of the functionality of the coins and what we can do with them. This is in line with the "Single Responsibility Principle" so that Player class won't have too many responsibilities.

2) With AddToWalletAction, it directly adds the coin value into the wallet's totalBalance and doesn't use the coin object it was given directly. This is because our requirements don't require us to use coins in any other way except for adding value to our wallet.

Therefore, our final design is in line with the "Single Responsibility Principle", and thus we have decided to go with this design for these 3 classes.

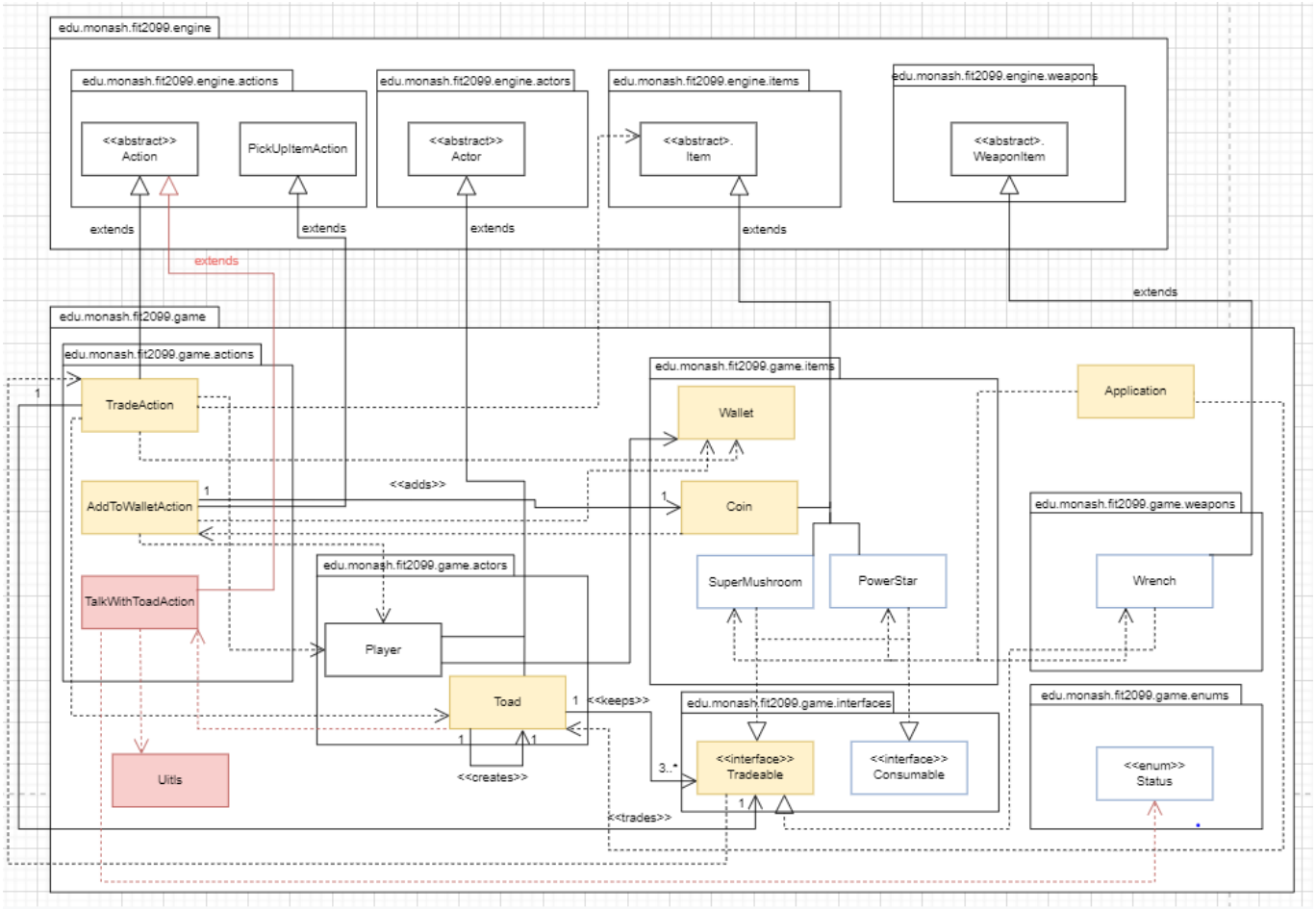
3: Toad, Tradeable, Application
Tradeable is an interface by which the Toad singleton class keeps an arraylist of, and that the items or weapons that are tradeable will implement this interface. Note that in driver class Application, we ensure that we add one of each initial Tradeable item into the Toad inventory. (PowerStar, SuperMushroom, Wrench).

In our previous design, we did an alternative to this: Where Toad has to instantiate one of Wrench, SuperMushroom and PowerStar, to store the current items he can trade with the player. However, this causes the Toad to have 3 more dependencies, one to each of the tradeable items. This goes against the principle of “Reduce Dependency”, as Toad class would have more dependencies. Also, when the Toad wants to replenish his inventory for every Tradeable item he sells, if we followed this alternative, Toad would need to use instanceof to check what specific class the Tradeable item is to be able to instantiate one and replenish his tradeable Inventory - this would then violate the Open Closed Principle since any future tradeable items added to the game would have needed to be type-checked again when replenishing the inventory.

Thus, in our current design, by having weapons, items or future classes implement "Tradeable" interface. Toad would only need to have an association to this Tradeable interface class by having an arraylist of Tradeable items!
This is in line with both the "Reduce Dependency Principle", “Open-Closed Principle”, as well as "Dependency Inversion Principle", where:
1. We are reducing the amount of dependencies/associations for Toad class (RDP),
2. If we were to extend the implementation by adding more Tradeable items, we won’t need to modify the code within the Toad class to ensure it works for each Tradeable item (which would have created more dependencies). (OCP)
3. We are making sure that the Toad concrete class doesn't depend on other concrete classes (Wrench, SuperMushroom, Powerstar) and that Toad depends on an abstraction which is the interface (DIP).
Therefore, the above rationale is why we implemented Toad class and the Tradeable interface in this way.

REQ6:

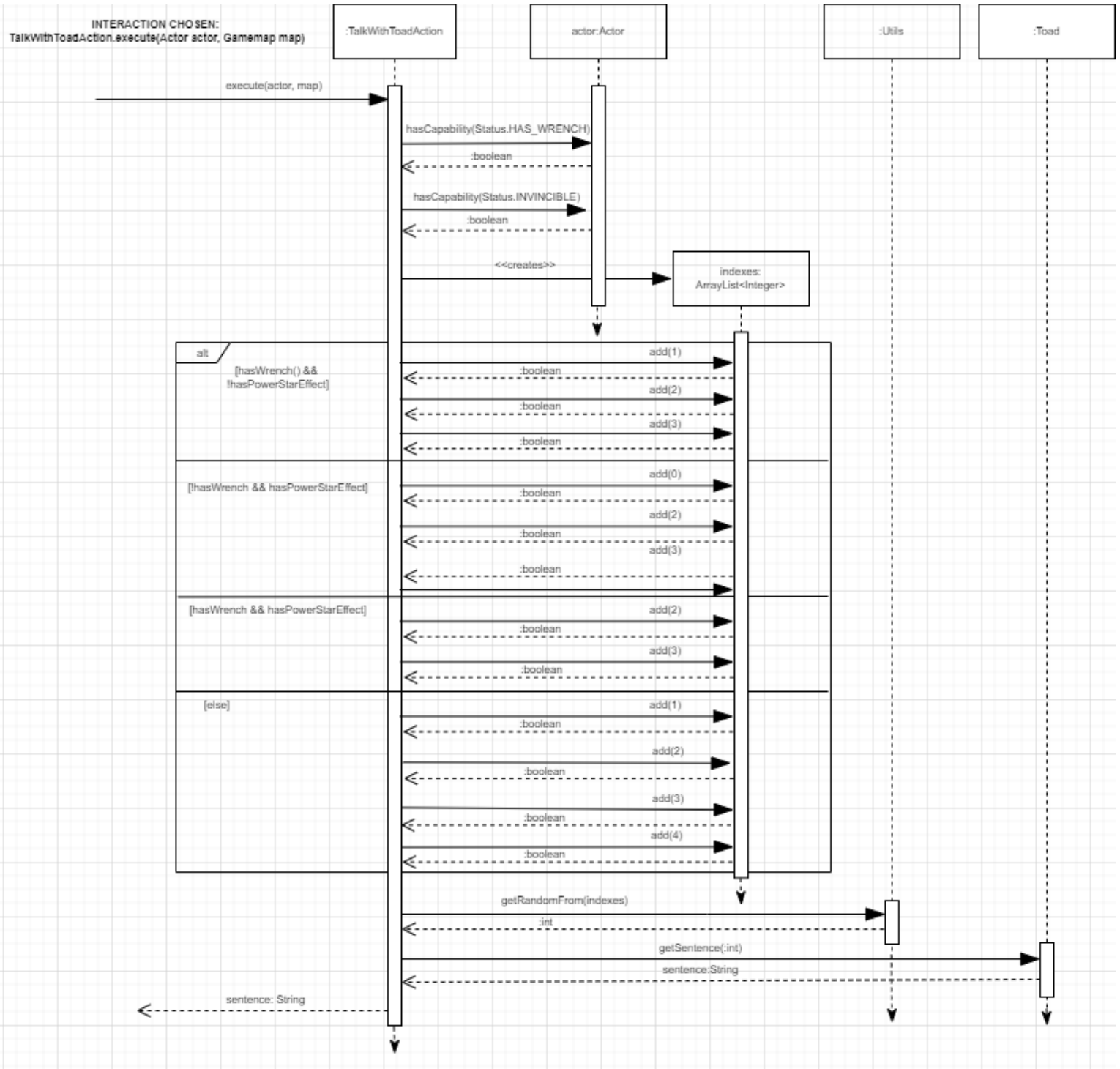
Class Diagram



Note that in the UML Class diagram for REQ6, that the red highlighted lines and classes represent the modified/added classes for REQ6 specifically.
Classes that are black represent classes existing in the base code.
Classes that have other colours (other than black & yellow) are classes added/modified for OTHER REQs.

Sequence Diagram

The below sequence diagram is for the TalkWithToadAction's execute interaction.



Overall Class Responsibilities

Note: Wallet, TradeAction, Coin, AddToWalletAction, Toad, Tradeable classes that are included in REQ6's class diagram are the same as in REQ 5.
Thus, for REQ6 we will include only the other newly modified/added classes for REQ6 in this responsibilities list.

Application

1. The main method is modified to include the Toad singleton instance in the middle of the map. Also it is modified to instantiate & add PowerStar, SuperMushroom, and Wrench into the Toad's tradeableInventory. (This means Toad has these tradeable items from the start of the game).

Toad

1. Modified Methods:
allowableActions method: add a new TalkWithToadAction to the actions list for the player to have it as an option.

TalkWithToadAction

1. Class Overall Responsibility:
This class is used to represent talking with a Toad Action. Note that in the current requirements, this class will only be used for talking with the Toad actor.

2. Relationship with other classes:
Inherits from Action class (extends this class).

3. Attributes:
private static String[] sentences = [1st sentence, 2nd sentence, 3rd sentence, 4th sentence];

5. Constructor:
creates instance of TalkWithToadAction.

5. Methods:
getSentence(int Index) which returns the String sentence referenced by the index given;

overrides Action's execute method.

- check if actor has a wrench.
- check if actor has powerstar effect.

if actor's inventory has wrench && powerstar effect not there:
get random sentence from toad's sentences array (2, 3, 4th sentences only);
elif has wrench && powerstar effect is there:
get random sentence from toad's sentences array (3, 4th sentences only);
elif powerstar effect is there:
get random sentence from toad's sentences array (1, 3, 4th sentences only);
else randomly pick any of sentence 1,2,3,4 from sentences array.

Utils

1. Class Overall Responsibility:
This class is used to provide utility methods which other classes can use.

2. Attributes:
private static Random object;

3. Added Methods:
getRandomFrom(ArrayList<integer> list), which returns a random integer from the given integer arraylist.
getRandomBias(), which returns a random double from 0 to 1.

Design Rationale

1: TalkWithToadAction

To be able to interact with Toad, the player needs to have an option to talk with it. Thus, we have the TalkWithToadAction that extends the abstract Action class.

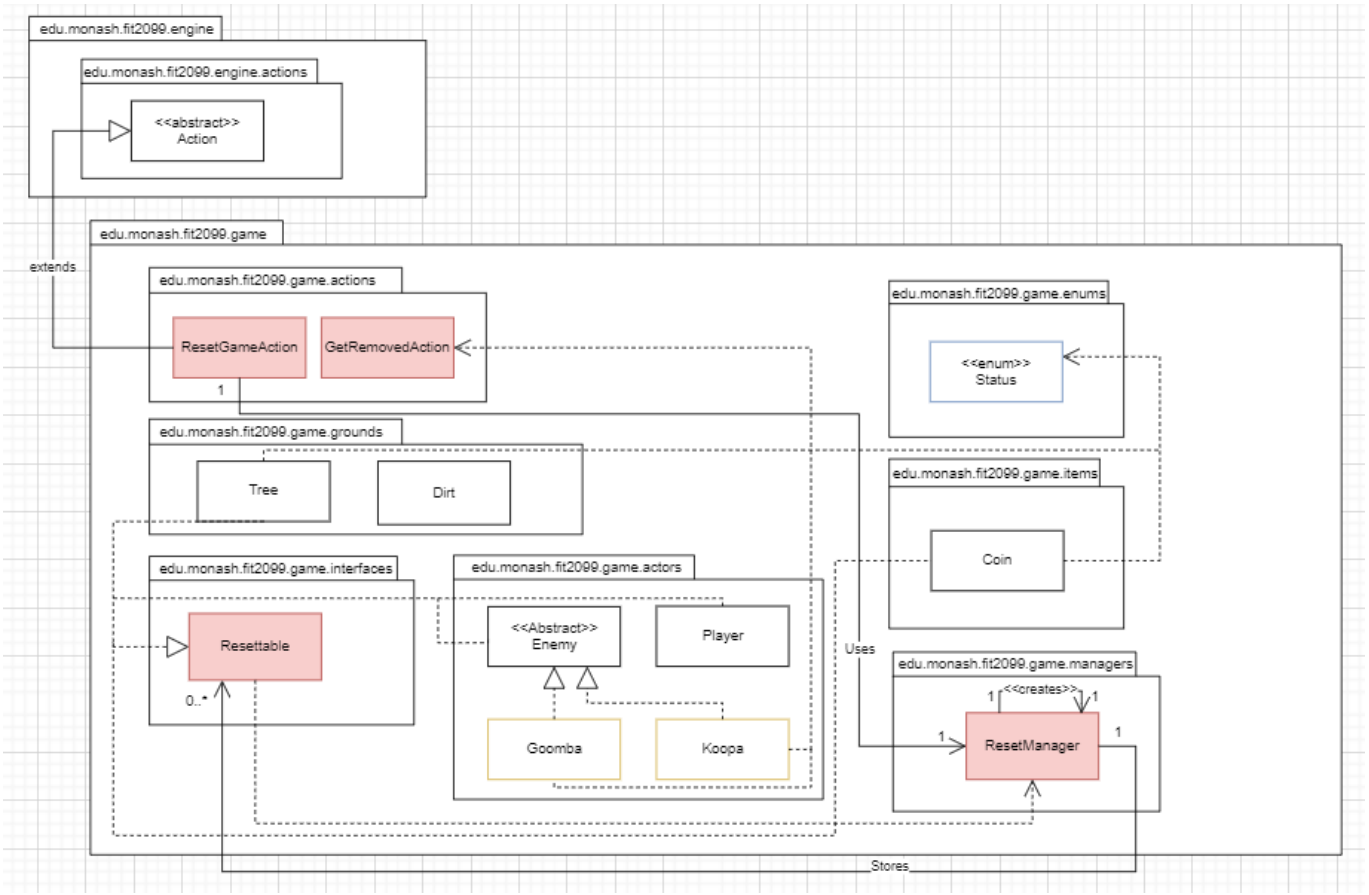
In this game, the ideal way to interact with the object is by attaching an appropriate action to its corresponding object (aligns with the meaning of "object-oriented").
Thus, this is shown with this: **Player ---<<uses>>---> Action** (dependency, not shown in the class diagram for REQ6 as it was already existing in the basecode).
In the engine code, it adds the allowableActions that the current actor can perform based on the surrounding actors/items/ground. For Toad actor, when it is near the current actor, it provides the actor with TalkWithToadAction via the allowableActions method, thus we are adding an instance of TalkWithToadAction to actions the player can execute. This allows the Player's actions shown in the menu to have an action to talk with Toad.

Alternatively, we can have a behaviours hashmap, and have a TalkToadBehaviour class that is added to the behaviours of the player. And in playTurn method can go through the behaviours of the player to get the action from that behaviour for the player. While this alternative accounts for extensibility by using the Behaviour class, however it would cause Player to have an added responsibility to take over (violates Single Responsibility Principle) and would have an added association with behaviour class, which is an added strong dependency for the Player class (which goes against the Reduce Dependency Principle).

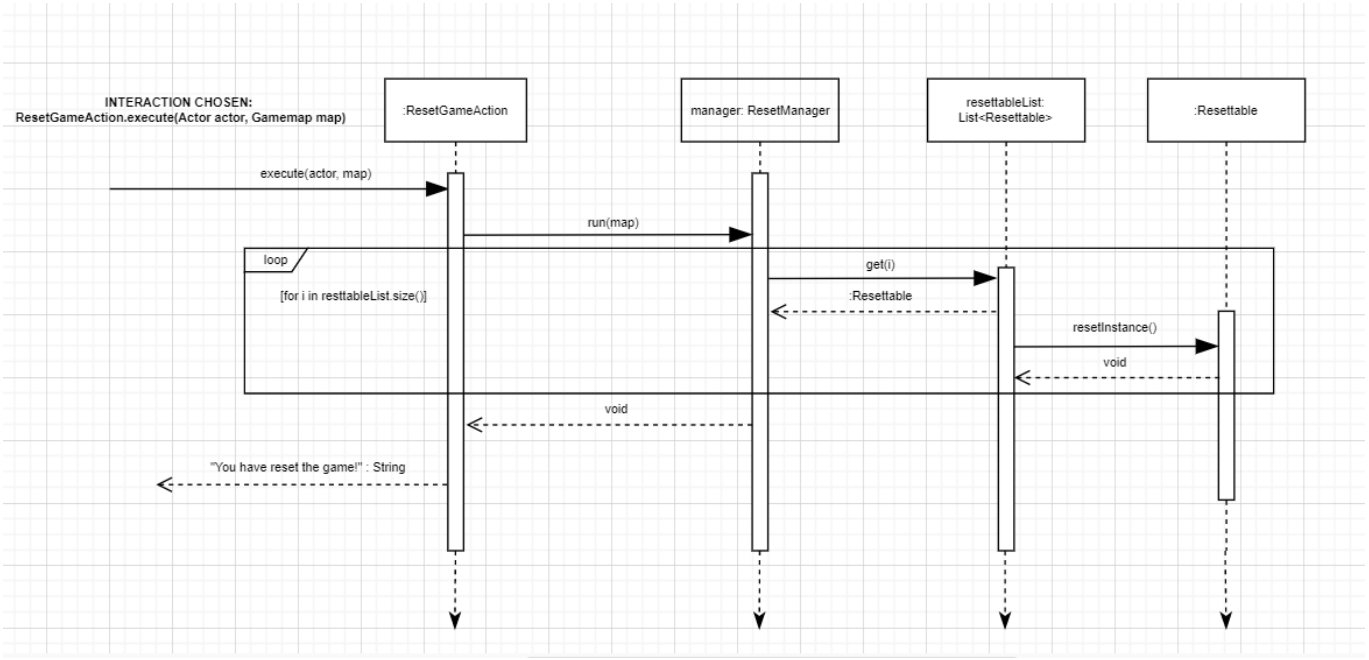
Thus, with our current design, we discard the alternative since it goes against the Reduce Dependency Principle and Single Responsibility Principle, and also because we should not need a list of behaviours for the player to choose from (not in the requirements). So, in our design only Toad Player has an association with TalkWithToadAction and we will go with this implementation.

REQ7:

Class Diagram



Sequence Diagram



<u>Overall Class Responsibilities</u>
Modified classes: Player, ResetManager, ResetManager
Added classes: ResetGameAction, GetRemovedAction.
Note that GetRemovedAction class was introduced in REQ 3 (Enemies - Goomba).

Player
1. Relationships:
It implements the Resettable interface class.

2. Constructor:
The player instance will be registered to the ResetManager singleton class with Resettable.super.registerInstance() method from the Resettable interface.

3. Methods
Implements resetInstance method (Resettable interface).
- For all other resettable instances, we would use resetInstance() to add the capability of that object to have a “RESET” status, and later on only in playTurn or Tick methods check for the RESET status to perform the resetting. However, in this method for Player class only, we are implementing the resetting straight away in this method (not in the playturn method). This is because if we use resetInstance to set the RESET status to the player & later on in playturn only check for the RESET status to perform the resetting, the resetting for player would be done in the NEXT turn, rather than the current turn when the Player chooses to perform the ResetGameAction. Note that either way should be fine, but our team has decided that it is better for the resetting of the player to be done in the turn that it executes the ResetGameAction.

ResetManager:
1. Class Overall Responsibility:

This class is used to represent the singleton manager that does the resetting on resettable instances within the game. All original methods in this class from the base code are implemented and none were removed.

2. Modified methods:
run()
- after resetting each instance in the list, it will remove those instances from the resettableList using the cleanUp method.

Tree
1. Relationships:
It implements the Resettable interface class.

2. Constructor:
The Tree instance will be registered to the ResetManager singleton class with Resettable.super.registerInstance() method from the Resettable interface.

3. Methods
Implements resetInstance() method (Resettable interface).
- For all resettable instances other than Player class, we use resetInstance() to add the capability of that object to have a “RESET” status, and later on only in playTurn or Tick methods check for the RESET status to perform the resetting. For Tree class, the checking of RESET status is done in the tick() method since it doesn’t have a playTurn() method.

Overrides tick() method.
- Checks if the Tree has a RESET capability status. If it does, it will replace the Tree with Dirt.

Coin
1. Relationships:
It implements the Resettable interface class.

2. Constructor:
The Coin instance will be registered to the ResetManager singleton class with Resettable.super.registerInstance() method from the Resettable interface.

3. Methods
Implements resetInstance() method (Resettable interface).
- For all resettable instances other than Player class, we use resetInstance() to add the capability of that object to have a “RESET” status, and later on only in playTurn or Tick methods check for the RESET status to perform the resetting. For Coin class, the checking of RESET status is done in the tick() method since it doesn’t have a playTurn() method.

Overrides tick() method.
- Checks if the Coin has a RESET capability status. If it does, the coin will be removed from the map.

Enemy
1. Relationships:
It implements the Resettable interface class.

2. Constructor:
The Enemy instance will be registered to the ResetManager singleton class with Resettable.super.registerInstance() method from the Resettable interface.

3. Methods
Implements resetInstance() method (Resettable interface).
- For all resettable instances other than Player class, we use resetInstance() to add the capability of that object to have a “RESET” status, and later on only in playTurn or Tick methods check for the RESET status to perform the resetting. For Enemy child classes, the checking of RESET status is done in the playturn() method since it doesn’t have a tick() method.

Overrides playTurn() method.
- Checks if the Enemy has a RESET capability status. If it does, the Enemy’s turn will execute the GetRemovedAction() to remove that Enemy from the map.

Koopa
1. Methods
Overrides playTurn() method.
- Checks if super.playTurn() returns a non-null action (in Enemy is where the checking of RESET is done). If it does return a non-null action, then it returns this action to be executed by Koopa.

Goomba
1. Methods
Overrides playTurn() method.
- Checks if super.playTurn() returns a non-null action (in Enemy is where the checking of RESET is done). If it does return a non-null action, then it returns this action to be executed by Goomba..

ResetGameAction
1. Class Overall Responsibility:
This class is used to represent the act of Player resetting the game. This will be added to the player's actions list if the player has yet to reset the game once.

2. Relationship with other classes:
Association with ResetManager.

3. Attributes:
Stores a private static attribute reference to the singleton ResetManager class object.

4. Constructor:
Creates an instance of ResetGameAction.

5. Methods:

overrides Action's execute method to call ResetManager’s run() method.
overrides Action's hotKey() method to return the specific hotkey, which is: "r".
Overrides Action’s menuDescription() to return a meaningful menu description for resetting the game.

Design Rationale

1: ResetGameAction, Resettable, ResetGameAction

We make use of the reset manager, resettable interface and resetGameAction.
By having a reset manager, the resetManager will be in charge of resetting every resettable instance in the game. And by having the resetGameAction class, this action class will be the one to use the reset manager to reset every instance of the class.

The alternative (what we did previously) was to have resetGameAction iterate through every relevant ground, item and actor on the map and perform a reset on it, without utilising resetManager class. This would have caused more dependencies on these classes and therefore violates the reduce dependency principle. Thus, instead of having resetGameAction handle resetting of every single class and have so many dependencies, we use resetManager and resetGameAction as explained explained earlier which will be in line with the reduce dependency principle, and also the single responsibility principle since only the resetGameAction and resetManager classes will be handling the functionality of reset.