Work Breakdown Agreement - ASSIGNMENT 1

Team Members: Jin Yao, Vanessa, Zubin (MA_LAB02_Team1)

Tasks & Deliverables

No.	Task	Description	Output Deliverable	Responsible Person(s)	Deadline	Reviewer & Tester	Review/Test Dateline
0.0	Create WBA & Sign WBA	Get tutor's confirmation that WBA is good enough, before getting members to sign it.	Work Breakdown Agreement	Jin Yao, Vanessa, Zubin	28 Mar		
0.1	Create Word Doc to store all our draft UML Diagrams + explanation for each of our REQs	1) Input all our UML Diagram drafts/explanation here, 2) Note: Pull from gitlab Before adding changes to doc & Push to gitlab After adding your part	UML Diagrams	Jin Yao, Vanessa, Zubin	28 Mar		
0.2	Create Word Doc to input Design Rationale	1) Explain our design choices, follow ASGN Design Rationale Requirements 2) Note: Pull from gitlab Before adding changes to doc & Push to gitlab After adding your part	Design Rationale	Jin Yao, Vanessa, Zubin	28 Mar		
1	**Design Draft Implementation & UMLs**	For each requirement, decide on: 1) How to best implement it 2) Why implement it that way 3) What new classes or methods to add 4) How these classes/methods relate & interact with the existing system 5) Draw draft UML class diagrams AND Sequence Diagrams NOTE: Input all this into respective SHARED WORD DOCS (1 for Design Implementation including Diagrams, 1 for Design Rationale as noted in Task 0.1, 0.2)	UML Draft Diagrams (Interaction & Class), Design Rationale Explanation				
1.1	REQ 1-2 Implementation	""	UML Draft Diagrams (Interaction & Class), Design Rationale Explanation	Zubin	31 Mar	Jin Yao, Vanessa	31 Mar
1.2	REQ 3-4 Implementation	""	UML Draft Diagrams (Interaction & Class), Design Rationale Explanation	Jin Yao	31 Mar	Vanessa, Zubin	31 Mar
1.3	REQ 5-6 Implementation	""	UML Draft Diagrams (Interaction & Class), Design Rationale Explanation	Vanessa	31 Mar	Jin Yao, Zubin	31 Mar

No.	Task	Description	Output Deliverable	Responsible Person(s)	Deadline	Reviewer & Tester	Review/Test Dateline
1.4	REQ 7 Implementation	""	UML Draft Diagrams (Interaction & Class), Design Rationale Explanation	Jin Yao, Vanessa, Zubin	31 Mar	Jin Yao, Vanessa, Zubin	31 Mar
1.5	Meeting to discuss final implementations &	Where we discuss: 1) Each other's work/implementations 2) Improvements to make on UML Diagrams/Design Rationale, and 3) To merge implementations	UML Draft Diagrams (Interaction & Class), Design Rationale Explanation	Jin Yao, Vanessa, Zubin	31 Mar	Jin Yao, Vanessa, Zubin	31 Mar
2	**Final UML Diagrams, Sequence Diagrams and Design Rationale**						
2.1	Meeting to finalize UML Diagrams, Sequence Diagrams and Design Rationale	1) Discuss & go through each other's diagrams & implementation 2) Any improvements made for each other's implementation 3) Merge our implementations	Final UML Diagrams (Interaction & Class), Design Rationale	Jin Yao, Vanessa, Zubin	3 Apr	Jin Yao, Vanessa, Zubin	3 Apr
2.2	**Testing & Checking** Final UML Diagrams, Sequence Diagrams and Design Rationale	1) Make sure each diagram & explanation is all good	Final UML Diagrams (Interaction & Class), Design Rationale	Jin Yao, Vanessa, Zubin	4 Apr	Each person checks everything twice & report back	5 Apr
2.3	**Merge** ALL Final UML Diagrams, Sequence Diagrams and Design Rationale INTO ONE PDF		Final UML Diagrams (Interaction & Class), Design Rationale	Jin Yao, Vanessa, Zubin	5 Apr		
3	**Final Assignment Submission**	Push the 1) PDF file, 2) make sure WBA is there, 3) Submit Files to Moodle					
3.1	**Final GitLab Push for PDF file**	Push the 1) PDF file and 2) make sure there's a copy of our WBA.	Submission Made	Jin Yao, Vanessa, Zubin	8 Apr	Jin Yao, Vanessa, Zubin	9 Apr
3.2	**Final Moodle Submission**	ONE of us to submit the 1) PDF file and 2) a copy of our WBA.	Submission Made	Jin Yao, Vanessa, Zubin	8 Apr	Jin Yao, Vanessa, Zubin	9 Apr

Contract Signing

I accept the WBA (Vanessa)

I accept the WBA (Chong Jin Yao) 28/3/2022

I accept the WBA (Zubin)

DESIGN

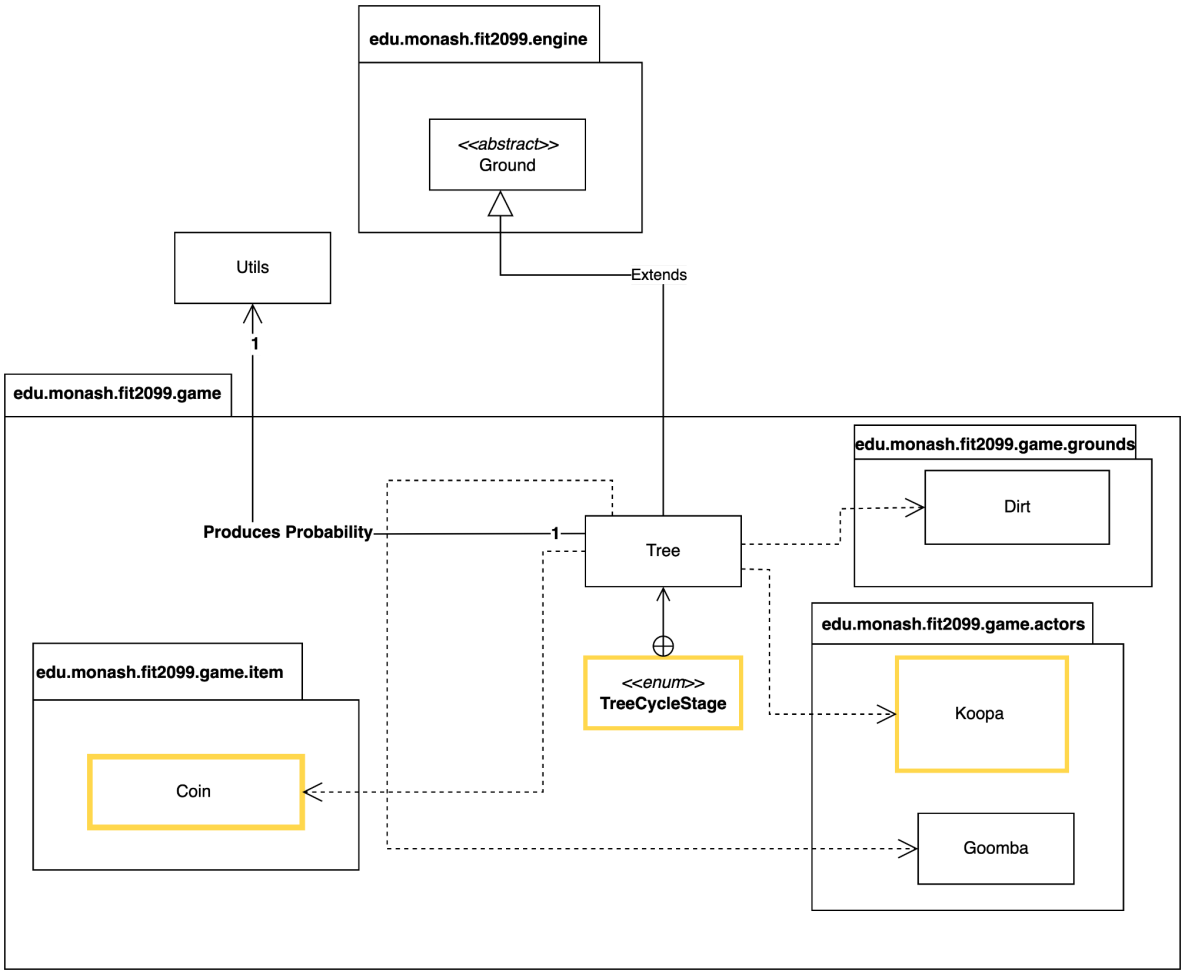
Note:

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.

> **Your class diagrams should not show the entire system.** The sample diagram in the base code shows the whole system to help you understand how the `game` works with the `engine`. But, in this assignment, you only need to show the new parts, the parts you expect to change, and enough information to let readers know where your new classes fit into the existing system. As it is likely that the precise names and signatures of methods will be refactored during development, you do not have to put them in this class diagram. However, the overall responsibilities of the class need to be documented *somewhere*, as you will need this information to begin implementation. This can be done in a separate text document (`.md` markdown format).

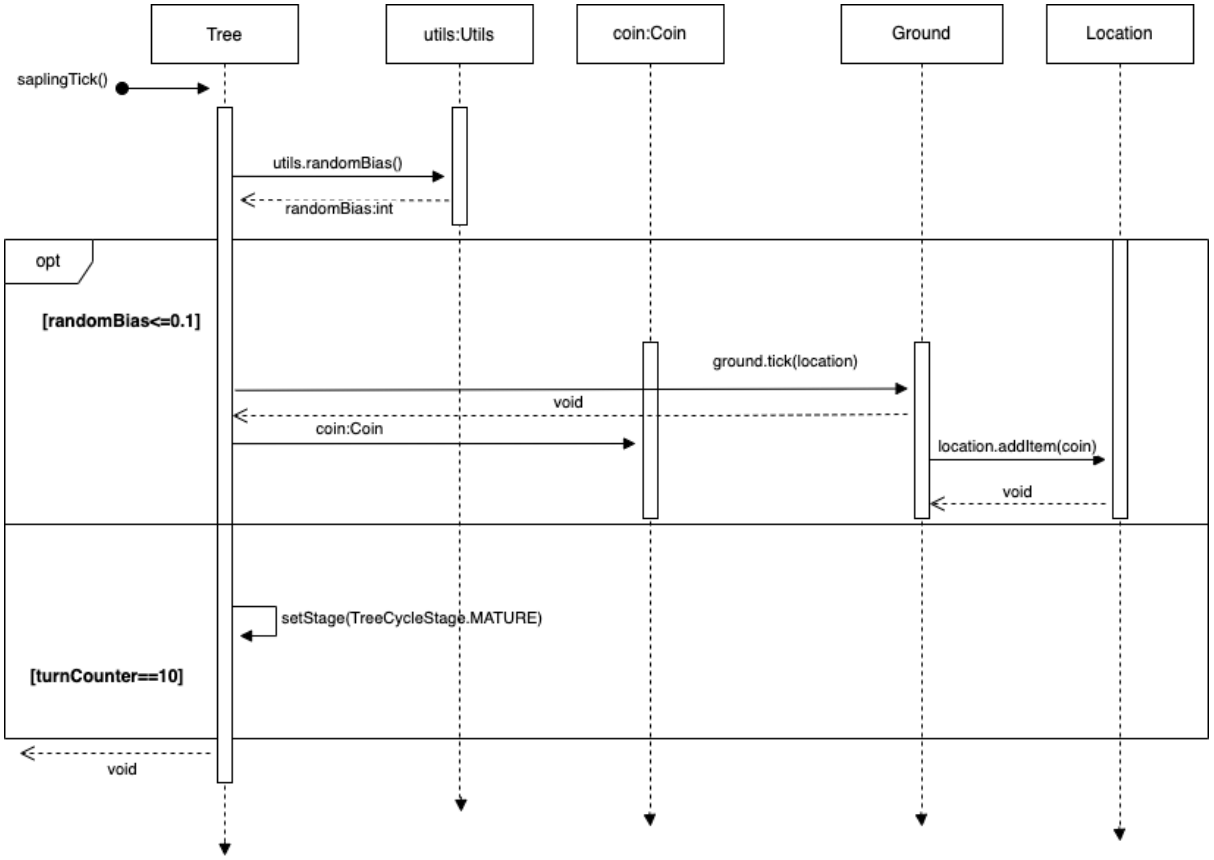
REQ1:

Class Diagram



Sequence Diagram

saplingTick() method's interaction was selected



Overall Class Responsibilities

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

Tree Class

1. Class Overall Responsibilities:

This is an existing class where we implement the different growth cycle stages from a sprout to a mature tree. Additionally, certain functions take place during the growth which is also accounted for, within the methods.

2. Relationship With Other Classes:

Has dependencies with Ground, Coin, Goomba, and Koopa

3. Attributes:

private int turnCounter;
private TreeCycleStage stage

Setter/s:

1. setStage();

TreeCycleStage enum class :

i) Has SPROUT, SAPLING & MATURE enum values

ii) Extends the Tree class

4. Constructor:
Modify the constructor so, as to display a character “+” when a Tree object is instantiated as a sprout
and to set the stage as the TreeCycleStage.SPROUT.

“t” is displayed when the saplingTick() method is called and finally “T” is called when matureTick() method is called.

5. Overridden method/s:

Override the ground.tick() method so that it is as follows:

```
ground.tick(Location location) {

        if(stage = TreeEnumCycle.SPROUT) { sproutTick() }

        Elif (stage = TreeEnumCycle.SAPLING) { saplingTick() }

        Else { matureTick() }

}
```

Where for each stage the Tree is in, it applies different checkings at each turn of time.

6. Methods:

1.sproutTick() method:

i) Use a random bias generator to calculate the probability. If randomBias<=0.1,

Spawn a Goomba. We first get the location of the ground with the argument given in the tick() function). Then, Location.addActor(new Goomba());

ii) We also need to check if an actor stands on it. So we call the ground.tick() method to check the current location for an actor object,
the Goomba spawns else, nothing happens.

iii) A turnCounter will keep iterating every time this method is called.Once turnCounter reaches a value of 10,setDisplayChar(“t”);
& setStage() to set stage attribute of Tree to be a SAPLING.Reset the turnCounter to 0 before exiting the method.

2. saplingTick() method:

i) Use the random bias generator again, and if randomBias<=0.1, a Coin object is instantiated.
The coin is dropped on the current location of the sapling which can be found using the location argument within the groundTick() method.

ii) Once turnCounter reaches a value of 10, setDisplayChar(“T”); & setStage() to set stage attribute of Tree to be MATURE.
Reset the turnCounter to 0 before exiting the method.

3. matureTick() method:

i) Use the random bias generator, if randomBias<=0.15, we spawn a Koopa object on the current location of the actor using the ground.Tick()
method.Then, add the actor to the location by using Location.addActor(new Koopa());
ii) Now, check for ‘Dirt’ objects in the surrounding area. To do this, have a list of dirtDestinations in the matureTick() method for each of the exits from the location of the Mature tree.

Find dirtDestinations:

a) Location of mature.getExits() -> returns exits, for each exit->
exit.getDestination(),getGround() to check if the ground is a dirt.
If yes, add the location to dirtDestinations.

b) If turnCounter%5==0 && dirtDestination is true, we spawn a sprout using the setDisplayChar() method.

c) Again, if randomBias<=0.20, get location of ground using ground.Tick() method and then, Location.setGround(new Dirt()).
Meaning the sprout has withered and died.

Design Rationale

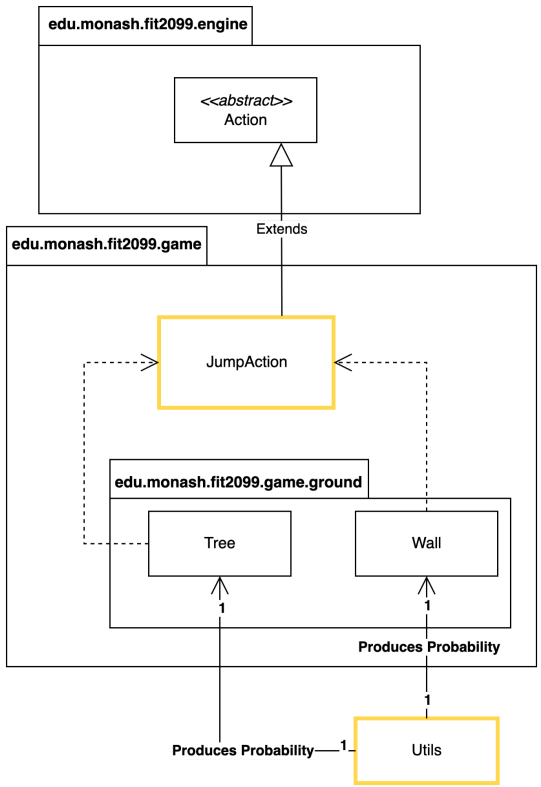
For this requirement, we use the existing Tree class to code all the functionalities. We could have used an interface for the methods to spawn all the different growth cycles of trees and have the Tree class implement that interface, and on paper, this would successfully fulfill the requirements of the Dependency Inversion Principle. In reality, however, this is rather unnecessary since the sprout, sapling, and mature phases are used by the Tree class and the Tree class only.

Instead, we used an ENUM class called TreeCycleStage class which already helps split the cycle stages and can be used within every single method to set a new stage using the setters in the class. The use of enumeration here fulfills the criteria of requiring a predefined list of values that refer to some textual/numerical data and thus justifies our choice of the ENUM class. Furthermore, the dedicated method for all the different stages fulfills the criteria for the Single Responsibility Principle.

We also increment and reset turn counters within each method so to say, every single method is responsible to fulfill its own functionalities while also keeping track of when to switch to the next stage.

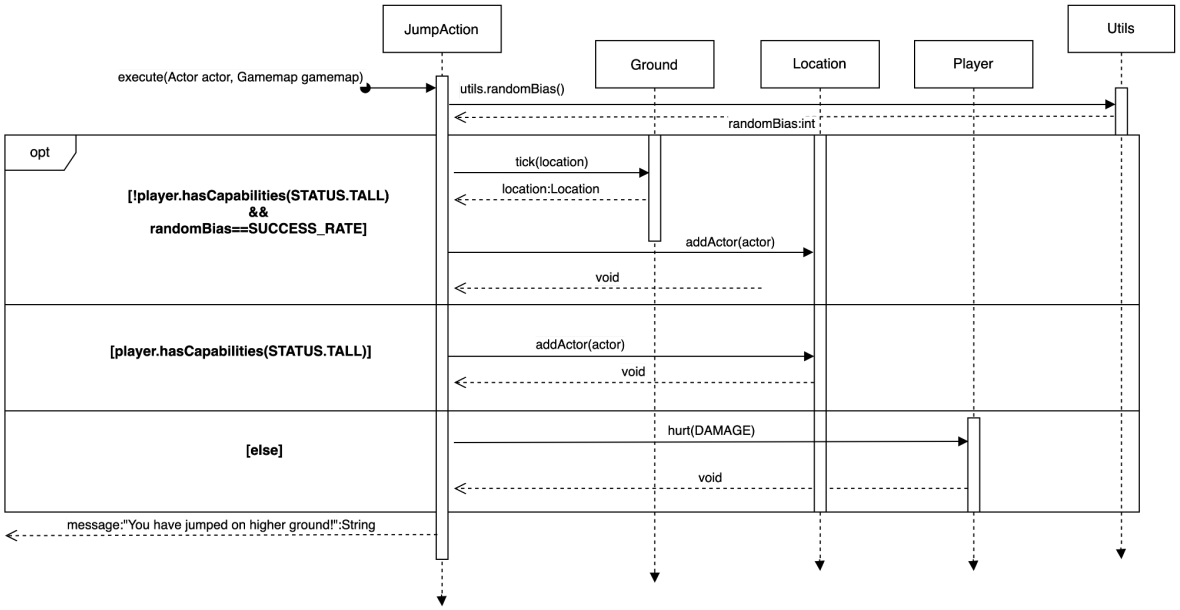
REQ2:

Class Diagram:



Sequence Diagram

execute() method's interaction was selected:



Overall Class Responsibilities

Jump Action Class

1. Class Overall Responsibilities:
We add a new class called JumpAction which extends the Player class. The responsibility of this class is to allow the actor to jump whenever it is in front of 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:
Inherits from Action since it is an additional action the player can perform.

3. Constructor:
In this constructor, the damage, location, and success rate for all the objects,i.e when Wall, Sprout, Sapling, and Mature are passed as the actor, in their corresponding classes/methods.

4. Attributes:

Location location

int damage

Utils utils=new Utils();

5. Methods:

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

If no Super Mushroom is consumed (checks actor's status on whether they have consumed it), then based on JumpAction's success rate, which we will determine using the randomBias within the Utils class,

we let the player go to jump over to the high ground. Otherwise, we implement the player.hurt(damage) method and deal corresponding damage to the player once they fail the jump.

If a Super Mushroom is consumed (checks actor's status on whether they have consumed it):

Let the player go to jumped over location straight away and print a line on the console based on the outcome.

Additional Actions:

In the Wall class: Override ground's allowableActions(Location location) method to also add a new JumpAction(Location location, damage, SUCCESS_RATE) to each of these IF the player has capability MUST_JUMP & doesn't have INVINCIBLE capability & if (actor at location != current actor)

In the Tree class: We again override the ground's allowableActions() method within every single 'tick' method but change the pre-condition before executing the jumpActions's execute method that corresponds to sprout,sapling and mature's success rate.

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

Design Rationale

For this requirement, we have decided to create a new class called Jump Action, and inherit the Action abstract 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 need to make changes to a couple of other classes, namely: Wall & Tree.

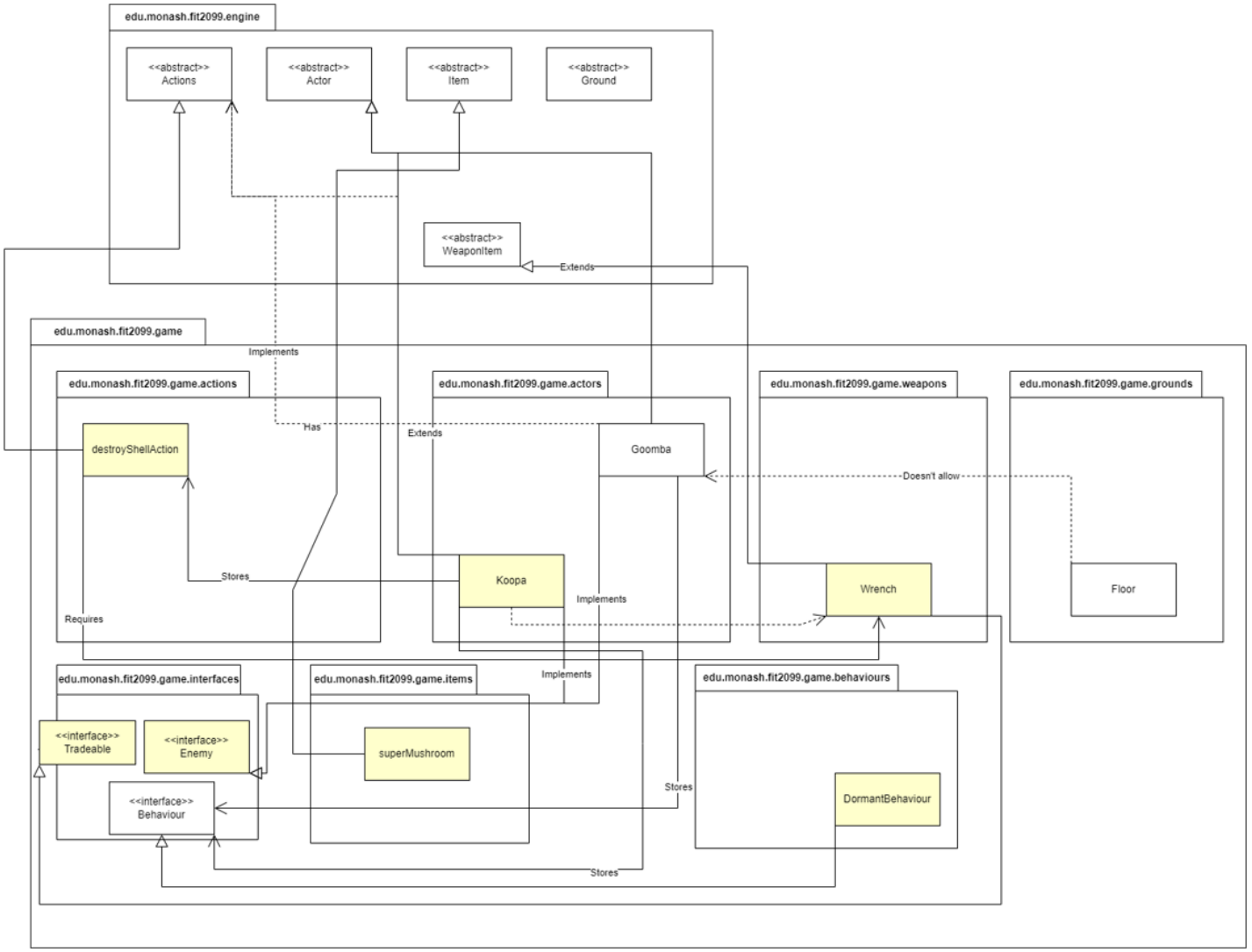
In the Wall class, we set the final attributes of success rate and damage referring to the that these values do not change and then override the allowable actions list.

We set final attributes within the class and set a precondition while overriding the method so that it doesn't affect the Single Responsibility Principle which clearly states that every module, class, or function in a computer program should have responsibility for a single part of that program's functionality, and it should encapsulate that part. Hence, the Utils class which generates our probability bias will not be responsible to handle all the success rates ranging from growing sprouts to completing jumps. Similarly, we also decided to set the final success rate and damage values for every growth stage object so we can compare them to the random bias, and if only they match, the jump functionality will take place.

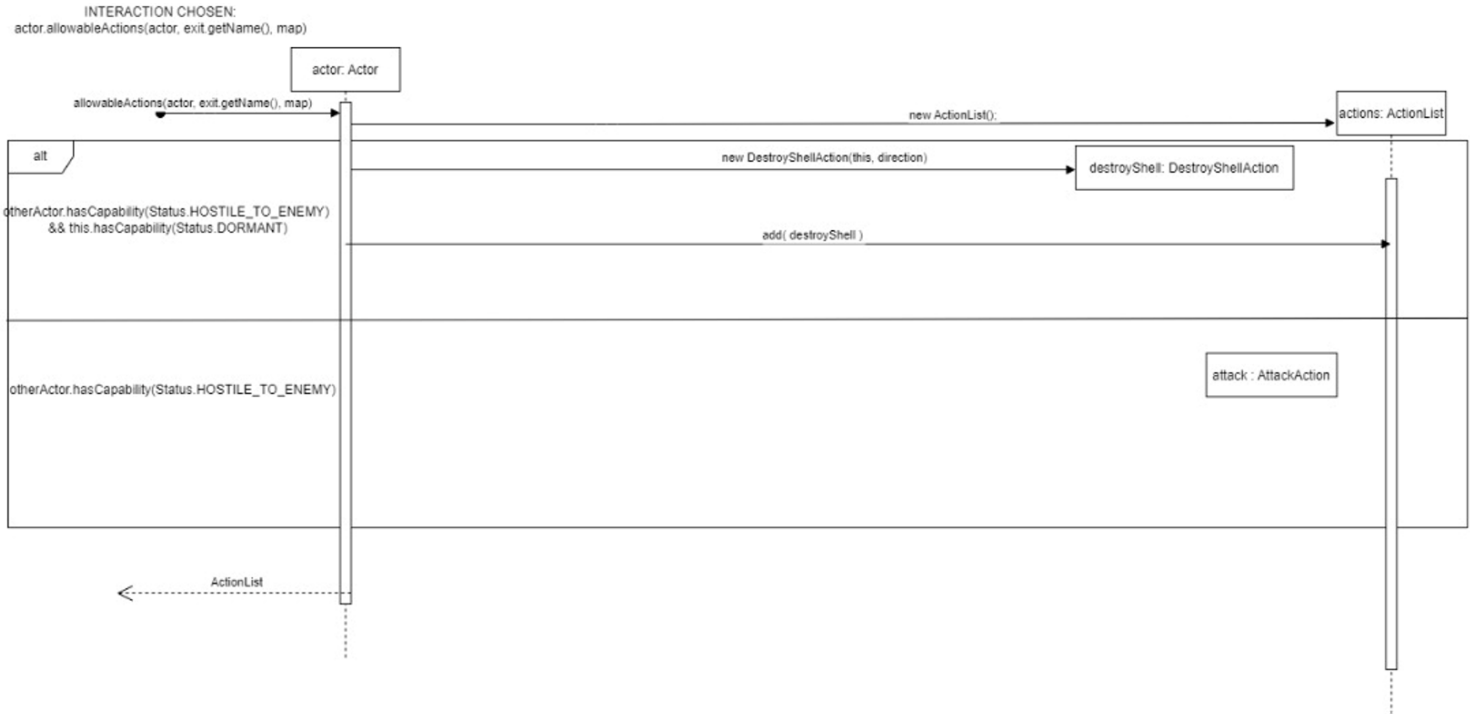
Additionally, if we were to add new high ground objects, we could implement them by calling the JumpAction constructor in that particular object's class and override the allowable actions list, the same way we did previously (given it also extends from Ground) without actually having to make any modifications to the Jump Action class specifically. This goes to show, that the class adheres to the Open-Closed Principle.

REQ3:

Class Diagram



Sequence Diagram



Overall Class Responsibilities

New classes added for REQ3: Koopa, DestroyShellAction, Wrench, Dormant behaviour, SuperMushroom, Enemy interface

Goomba

1. Changes to class:

Add logic to playTurn() method to use RandomBias from Utils class to perform 10% suicide possibility (through removal from map).

Has new attribute to count whenever Goomba is added or removed from map Has new attribute called Max_Goomba_Limit, final number limit for Goomba spawns

2. Relationship with other classes:

Implements Enemy interface Association with Suicide/GoombaHandler and BehaviourClasses (Wander/Dormant) Dependency on ActionList class, DoNothingAction

3. Methods:

Boolean method canSpawn returns true or false based on GoombaCount and Max_Goomba_Limit

Getters for new attributes

Koopa

1. Class Overall Responsibility:
This class implements a new enemy type that will attack the player actor and follow it around. It will implement a new dormant behaviour when knocked unconscious by the player and remain on the map until destroyed by a player with a wrench. Drops a mushroom when its shell is destroyed. Behaves similarly to Goomba besides dormant behaviour. Punches with 30 damage and 50% accuracy.

2. Relationship with other classes:
Extends actor class Implements Enemy interface Association with Behaviours and BehaviourClasses (Wander/Dormant) Dependency on ActionList class, DoNothingAction, DestroyShellAction

3. Attributes:
TreeMap of Behaviours

4. Constructor:
Takes no input but uses super("Koopa", "K", 100) and puts WanderBehaviour into the TreeMap with priority (Integer) as 10.

5. Methods:
Has its own implementation of playTurn method with added checks to see if TreeMap for behaviours contains Dormant behaviour, returns new DoNothingAction() whenever Dormant behaviour is found.

Overrides allowableActions() and implements its own version, which checks if player has a Wrench in their inventory, adds the newly created DestroyShellAction to ActionList instance

DestroyShellAction

1. Class Overall Responsibility:
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. Relationship with other classes:
Extends AttackAction

3. Attributes:
Similar to AttackAction... an Actor that represents the target of the attack String to represent attack direction No need for random number attribute Attribute representing wrench object

4. Constructor:

Takes input for target Actor and String for direction Assigns attributes to input.

5. Methods:
Overrides execute() and removes Koopa actor from the map, no need to ensure actor is Koopa as Dormant behaviour currently only used by Koopas. No need to ensure player has wrench as it is done in playTurn(). Returns message telling player that Koopa shell has been hit with Wrench. Needs to create a SuperMushroom object on the map at the location of Koopa object.

Overrides menuDescription() to return different message notifying player shell has been destroyed successfully, e.g "KAPLOW KOOPA SHELL HAS BEEN OBLITERATED!!".

Wrench

1. Class Overall Responsibility:
New weapon for player class to use. Can be picked up and dropped and has 50 damage and 80% acccuracy. Used for destroying Koopa shells and can be used for basic attacks as well (AttackAction).

2. Relationship with other classes:
Extends WeaponItem class ImplementsTradeable interface AttackAction has dependency on this class

3. Attributes:
Integer attribute representing damage, should not be changeable value Integer attribute representing hitRate of weapon, should not be changeable value String attribute representing UI display verb for attacking with weapon, e.g "KRONK" or "BONK"

4. Constructor:
Requires 5 input values. All representing name of weapon, display character of weapon, damage of weapon , attacking verb and hit rate of weapon. Will be required to use super(name, displayChar, True/False(weapon portability))

5. Methods:
Overrides all methods from parent class damage(), returns damage dealt by weapon verb(), returns attacking verb chanceToHit(), returns hit rate of weapon

DormantBehaviour

1. Class Overall Responsibility:
To be applied to Koopa objects that have been defeated by the player. Responsible for making sure defeated Koopa objects do not move carry out any actions or behaviour. They cannot move, attack, wander, follow or interact with anything. To be added to Koopa's behaviours.

2. Relationship with other classes:
Implements Behaviour

3. Attributes: none

4. Constructor: none

5. Methods:
Overridden getAction() method that requires Koopa object and game map as input. Will check if Koopa is on map. Returns null always so playTurn does not receive any actions for Koopa object.

SuperMushroom

1. Class Overall Responsibility:
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)

2. Relationship with other classes:
Implements ConsumableItem interface Extends item class

3. Attributes:
Similar to items... Int representing hp increase value Boolean representing consumption of item String representing name of item char representing displayChar of item '^' boolean representing portability of item ActionList object CapabilitySet object

4. Constructor:
super(name, displayChar, portable) assigns hp increase value attribute 50

5. Methods:
allowableActions() returns to player ConsumeAction instance Getters for name, hpIncrease attribute, displayChar two tick() methods for when on ground or actor inventory

Enemy Interface getAttackAction()

Allows enemy Actors to implement and use all methods within it. Provides enemy actors with a getAction method to so the player can attack it. For future extensibility, can provide actions for attacking players in future.

Implementation in PowerStar and SuperMushroom would require the implementation of the getAction method by...

Taking an Actor, the player and an Item, the magical item as input and returns to the player the ConsumeAction.

Design Rationale

Implementing Dormancy

For REQ3 we mainly had to implement a dormant effect for all Koopa classes that were knocked unconscious by the player. In order to do this , we had to first create a new Koopa class to represent the new enemy type we were going to introduce into the game.

Following what was done for the Goomba class we decided to extend the Actor class and keep the same behaviour attribute. Instead of using a HashMap this time , we elected to use TreeMaps instead. The process of selecting actions from behaviours in playTurn() is slightly random as the entries are not sorted, the way it functions currently gives the first action found to be not null. TreeMap allows us to have sorted entries in the Map, allowing us to give the Dormant Behaviour the highest priority and actually have priority selection function. This allows the unconscious Koopa's to always return dormant actions once labelled as Dormant.

Alternatively, using a for loop ranging from 0 to the size of the HashMap for indexing would have been a plausible method to loop through entries with priority. However, this would require us to have the keys entered from 1-10 (highest to lowest priority). This method as a whole is just more troublesome to code and would have likely made the performance of the game worse. Aside from that future programmers would also have had to been made aware of this to avoid any mistakes.

We plan to keep Koopa's from performing any unwanted actions by returning null constantly when getActions() of the new DormantBehaviour class is called. This is how the playTurn() method of the Koopa class is used to get actions every tick. The if statement in the AttackAction class will be changed to check if the target is unconscious and not a Koopa in order to avoid removing them from the map. An else statement to change the display character of the Koopa to 'D' and to add the Dormant Status to the Koopa will be added as well.

The Dormant status will be used in allowableActions(). The code checks the target of attack for Dormant Status with hasCapability() and gives the player the option to use the new DestroyShellAction class on it if the Dormant Status is present. The old code will have to be updated to check for HOSTILE_TO_ENEMY status and not Dormant in the target actor to perform an attack. This follows the requirement which states that players cannot be given the option to attack dormant Koopas.

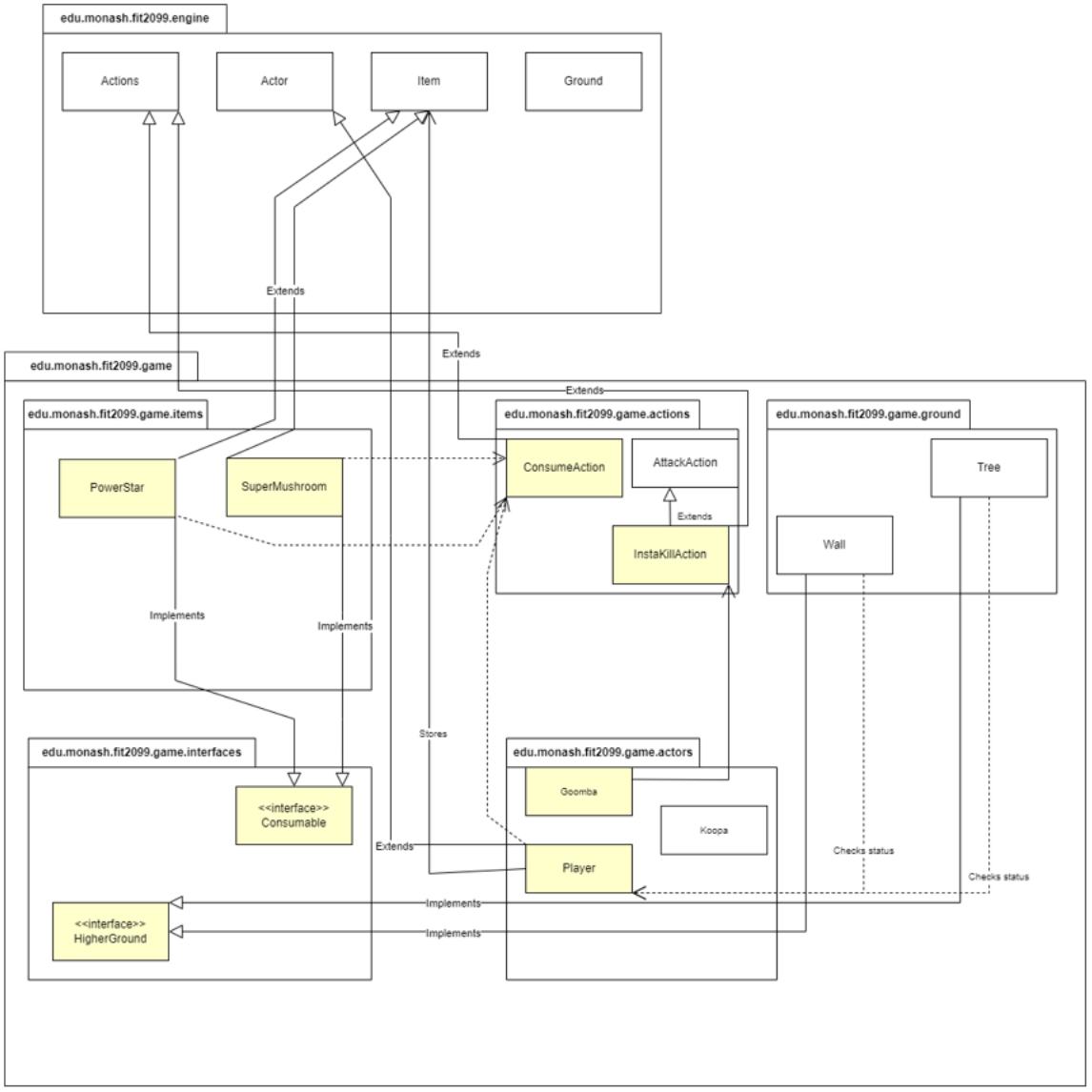
The new DestroyShellAction class will be made to extend the AttackAction. Because destroying a shell is a type of a attack action , it is fitting to extend the AttackAction since both of them behave similarly (aligns with the meaning of object-oriented) .

This implementation of DestroyShellAction and DormantBehaviour allows us to code with the Single Responsibility Principle as we delegate the functionality of Koopa class' actions to other classes, instead of having Koopa be responsible for all of it's actions.

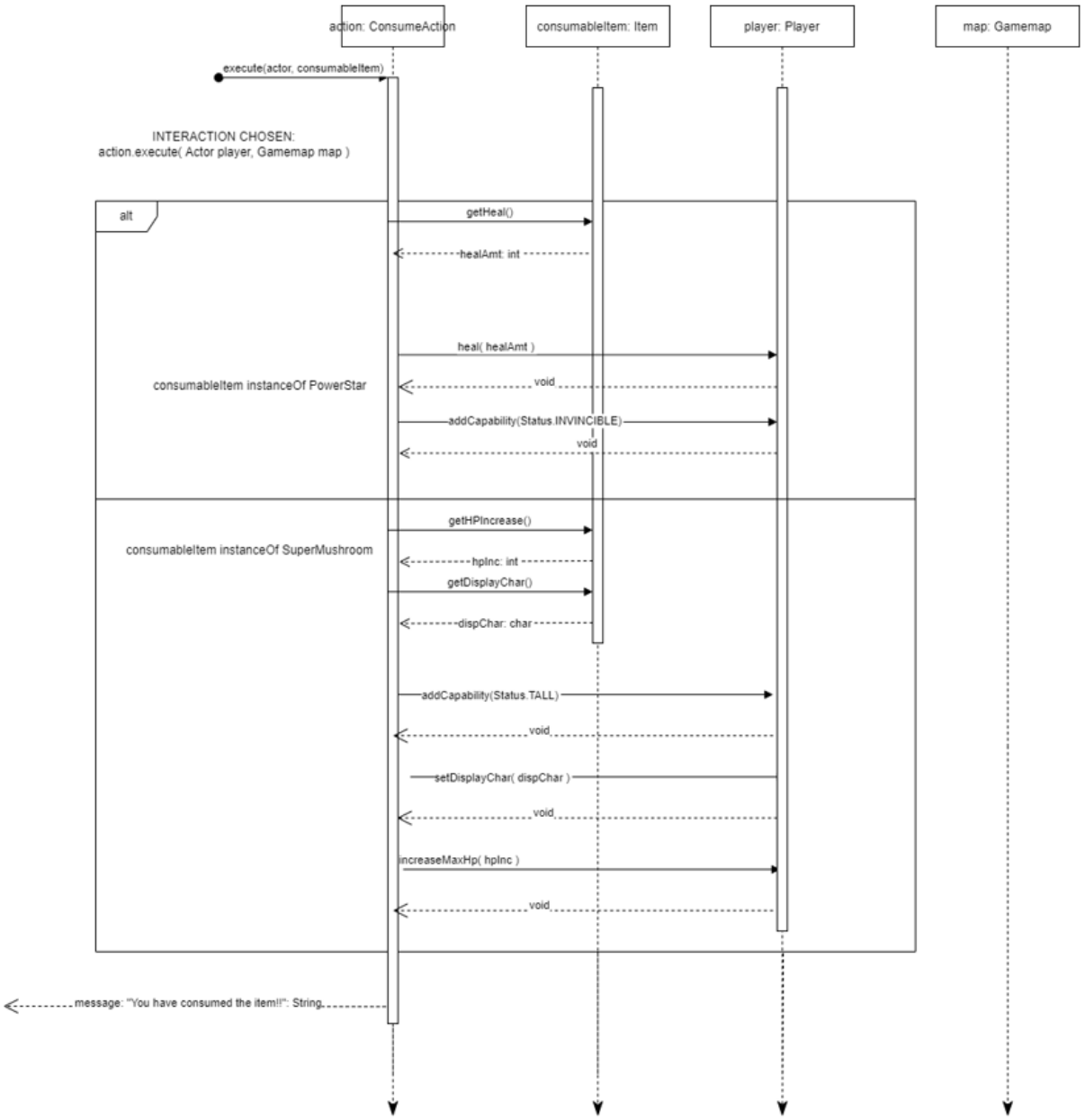
We also plan to implement an Enemies interface to allow for future extensibility. We have so far implemented a getAttackAction() method which simply returns an AttackAction to the Goomba and Koopa classes. We believe adding the interface will allow us to add more functionality to Koopas and Goombas; such as the ability to attack the Player entity.

REQ4:

Class Diagram



Sequence Diagram



Overall Class Responsibilities

New classes for REQ4: SuperMushroom, ConsumeAction, PowerStar, InstaKillAction, Consumable interface, HigherGround interface

SuperMushroom

Class Overall Responsibility:
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.

Relationship with other classes:

Implements ConsumableItem interface Extends item class
Attributes:

Similar to items... Int representing hp increase value Boolean representing consumption of item String representing name of item char representing displayChar of item '^' boolean representing portability of item ActionList object CapabilitySet object
Constructor:

super(name, displayChar, portable) assigns hp increase value attribute 50
Methods:

allowableActions() returns to player ConsumeAction instance Getters for name, hpIncrease attribute, displayChar two tick() methods for when on ground or actor inventory

ConsumeAction
1. Class Overall Responsibility:
Responsible for providing player with status/buffs in relation to consuming any consumables. Has to change hp values, display characters and update capabilities of player by assigning new Statuses.

2. Relationship with other classes:
Extends actions class

3. Attributes:
Actor object to store target of consumption (player) Item object to store item being consumed

4. Constructor:
Takes input for target Actor and item object Assigns attributes to input.

5. Methods:
Overridden execute() method. Checks what item is being consumed and proceeds with making respective changes based on what item is consumed. Also has to provide player with required Statuses, either TALL or INVINCIBLE. Has to remove item from player inventory after consumption if item is SuperMushroom.
Display method to notify user item has been consumed

PowerStar
1. Class Overall Responsibility
To give player ability to walk freely and heal player by 200 health points. Buff lasts 10 ticks in game. Needs to make player invincible to damage. Execute methods of any actions that damage player actor have to be changed to check for INVINCIBLE status. Allows player to

use InstaKillAction when having status and walk over walls and trees. Has to convert walls and trees into dirt and drop a coin ($5)

2. Relationship with other classes:
Implements ConsumableItem interface Extends item class

3. Attributes:
Similar to items... Boolean representing consumption of item Int representing numberOfTicks Int representing hp increase value String representing name of item char representing displayChar of item '^' boolean representing portability of item ActionList object CapabilitySet object

4. Constructor:
super(name, displayChar, portable) assigns hp heal value attribute 200 isConsumed assigned false

5. Methods:
allowableActions() returns to player ConsumeAction instance Getters for name, hpIncrease attribute, displayChar Override item.tick() to increment numberOfTicks by 1 each time it is called. Before incrementing, should check if boolean attribute representing consumption is true or false If true, set numberOfTicks to 0 Need to remove item when numberOfTicks reaches 10 String Display methods for when item is active on player

InstaKillAction
1. Class Overall Responsibility:
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:
Similar to AttackAction... an Actor that represents the target of the attack String to represent attack direction No need for random number attribute

4. Constructor:
Takes input for target Actor and String for direction Assigns attributes to input.

5. Methods:
Overrides execute() and removes target actor from the map. Returns message telling player that target actor has been killed.

Overrides menuDescription() to return different message notifying player shell has been destroyed successfully, e.g "OBLITERATED TARGET_ACTOR_NAME!!".

Consumable interface
1. Class Overall Responsibility:

An interface that represents items that can be consumed.Allows items to implement and use all methods within it. Provides Magical items with a getAction method so the player can successfully consume an item.

2. Relationship with other classes:
Implemented by SuperMushroom and PowerStar

3. Methods:
getConsumeAction(), will be implemented by SuperMushroom and PowerStar

4. Attribute:
boolean attribute isConsumed.
isConsumed returns True or False depending on whether item is consumed

HigherGround interface
Allows higher ground(Wall and Tree currently), Ground types to use methods within it. Provides all types of higher ground with a convertToDirt() method that when implemented should change the ground to dirt and drops a coin ($5). For future extensibility for other higher grounds that may exist, they could implement the convertToDirt() method in a different way.

1. Method:
convertToDirt()

Design Rationale
Consuming Items
We have added 2 new classes called PowerStar and SuperMushroom, both of which extend Item class and Implement a newly created Consumable class. We plan to implement Magical Items by giving the player the option to consume them whenever present in their inventory.

For the act of consuming a magical item, our team has elected to create a separate ConsumeAction class that is responsible for providing the player with any health and behavioural changes. This form of implementation adheres to the Single Responsibility Principle as we once again delegate the functionality of consuming PowerStar or SuperMushroom to another class.

We also plan to implement a Consumable interface with a getAction method that returns ConsumeAction for the allowableActions method in the PowerStar/SuperMushrooms class to use. Has to check if item is consumed using it's isConsumed() method. Returns consumeAction if isConsumed() returns False. This implementation of the Consumable interface will allow us to code with the Open-Closed Principle. This is because any new consumable items added to the game after the initial implementation of the action in the player class will be able to be consumed without having to modify any code. You would only

have to add the item itself. This ensures that the actions will be made available to the players whenever a magical item is present in their inventory.

Without the interface, we would have had to check for the individual items in the inventory to see if they were instances of SuperMushroom or PowerStar. With the Consumable interface, we can just check if items are an instance of the interface.

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

For the actual consumption of items, we decided on using a separate ConsumeAction class that extends Actions. This class will take as input the Magical Item and the Player object. It overrides the execute method to perform any changes to the health of the player and gives the player Statuses that will allow them to carry out any additional actions, like freely jumping, destroying ground or instantly killing enemies.

After the execute method is completed, the item will be removed from the players inventory.

The act of consuming Magical Items provides players with a new action. 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.

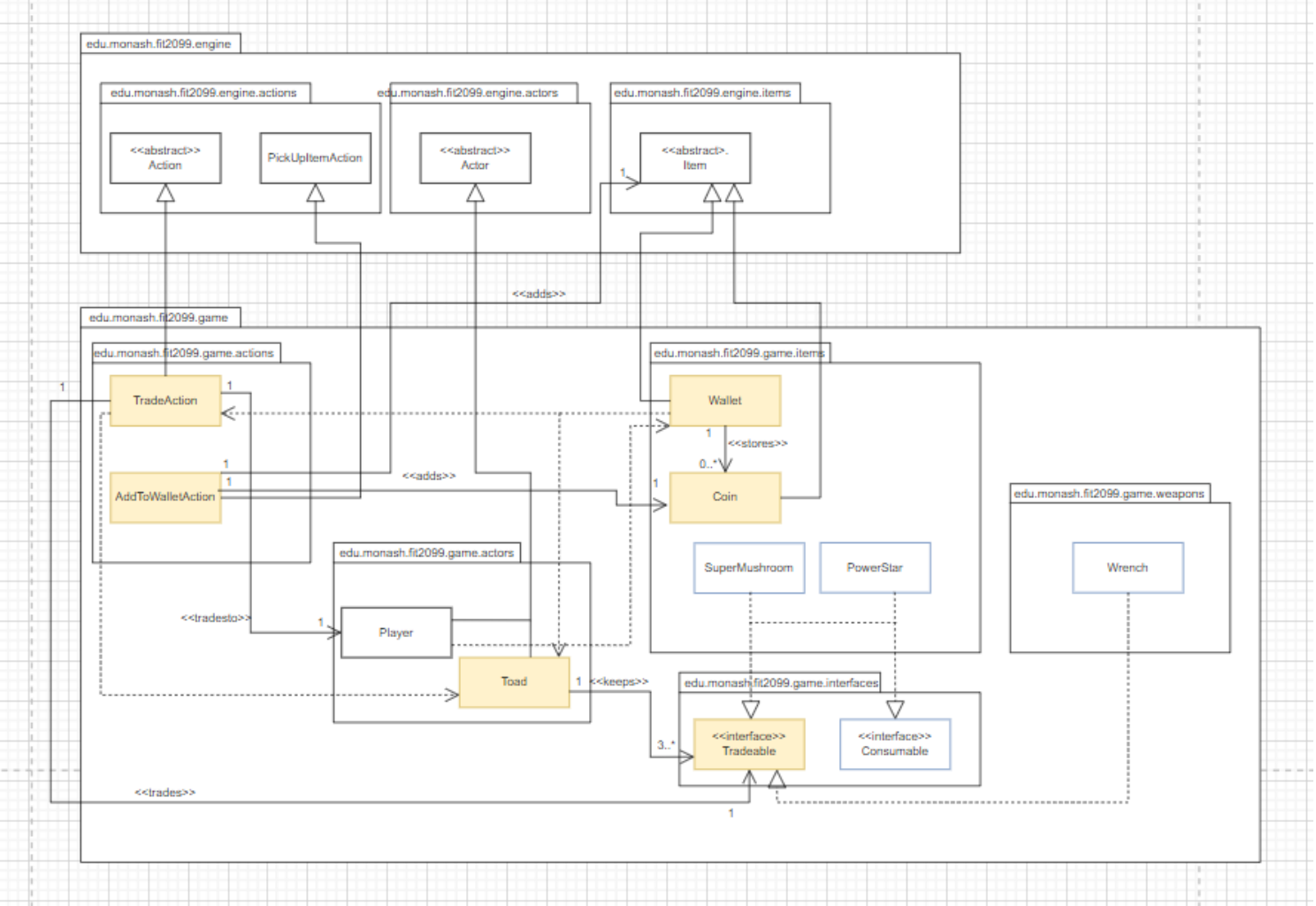
The ability to destroy ground is currently implemented by having Wall and Tree detect when a player has INVINCIBLE status in its allowableActions() method. It then changes itself to Dirt and drops a Coin ($5) with a convertToDirt() method from HigherGround interface.

Alternatively, we tried implementing a new method in both Wall and Tree called convertToDirt() but decided that implementing an interface would be much better for future extensibility.

Another alternative we tried was implementing a DestroyGroundAction to follow the Single Responsibility Principle but doing so would require the actor to perform the action. Using allowableActions() also causes the DestroyGroundAction to be shown in the player's menu as an option.

REQ5:

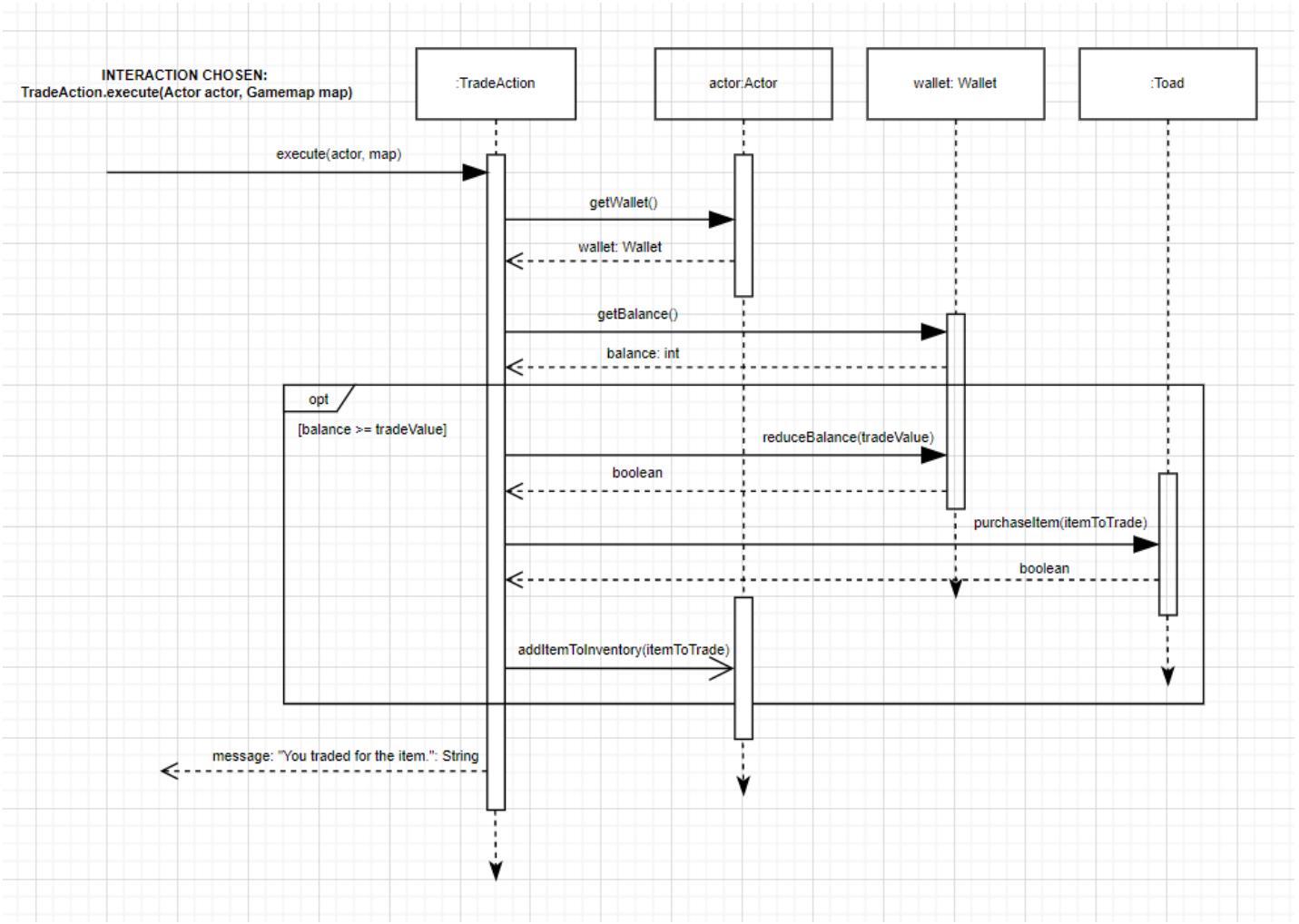
Class Diagram



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 basecode. Classes that have other colours (other than black & yellow) are classes added/modifed for OTHER REQs.

Sequence Diagram



Overall Class Responsibilities

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

Player

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

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 that contains coins and monetary value for trading items within the game. It is also used to get/reduce/add the total balance of value of coins that the player has during the game.This assigned item should automatically be in the inventory list of the player during player construction, since it is fixed and cannot be removed from every player.

2. Relationship with other classes:

Inherits from Item abstract class (extends this class).
Dependency on TradeAction class.
Association with Coin class.

3. Attributes:
private int totalBalance,
private ArrayList<Coin> coins,
String name = 'Wallet', String displayChar = 'w', Boolean portable = false
(the ones above are inherited from Item abstract class)
*When instantiated, it will NOT be portable

4. Constructor:
Creates instances of tradeAction, one for each of Toad's tradeableItems.
These tradeActions are added to the list of allowable actions for the Wallet item.

5. Methods:
private getBalance() to return the balance
private reduceBalance() to reduce the balance
private addBalance(ArrayList<Coin> coins) to add to balance

TradeAction
1. Class Overall Responsibility:
This class is used to implement an action where each player can trade some coins to purchase an item from Toad actor.

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

3. Attributes:
private Player player;
private Tradeable itemToTrade;
private int tradeValue;
private String hotkey; (inherited from abstract class)

4. Constructor:
creates instance of tradeAction with a reference to the item to be traded, and the player making the trade.

5. Methods:
overrides Action's execute method.
overrides Action's getHotKey() method to return the specific hotkey attribute it has.
overrides execute() method:
- Retrieves wallet from Player
- Remove coins from Player's wallet (if balance is enough for the item)
- 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 Sapling class), that can be added to the Wallet item of a player, or to the inventory of the Toad.

2. Relationship with other classes:
Inherits from Item abstract class (extends this class).
Wallet has an association with Coin class.

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 to the Player's wallet instead of its inventory. AddToWalletAction should also remove the coin object from the map position.
- returns the AddToWalletAction object

AddToWalletAction
1. Class Overall Responsibility:
This class is used to represent an adding item to Wallet Action. Note that this class will only ever be used for adding coins to the wallet.

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

3. Attributes:
private final Item item, as inherited from PickUpItemAction class

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 into the Actor's wallet item.

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 and who sells several items (for now: Wrench, SuperMushroom, PowerStar) to players.

2. Relationship with other classes:
TradeAction has a dependency with this class.
Extends Actor abstract 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.
In constructor, set String name='Toad'; String displayChar = 'O';.

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

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

purchaseItem(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.

Tradeable

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

2. Relationship with other classes:
TradeAction has an association with Tradeable.
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.

Design Rationale

1: Purchasing an item: TradeAction and Wallet class

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 extends Item. This would allow an actor who has a Wallet to use their Wallet item to perform this tradeAction action.

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: **Wallet ---<<create >>---> TradeAction** (association), and **Player ---<<stores>>---> Wallet** (association). The Wallet is the item object that gives the Actor (i.e., the Player) an action to purchase an item. Alternatively, we can add an attribute called totalBalance in Player class to account for the total amount he has received from collecting coins.
However, doing this will require the Player to instantiate a tradeAction in its “playTurn” method in order to add it to the list of actions it can do. By doing so, this would generate an additional dependency for Player to TradeAction, and also “playTurn” would have to do added functionality such as checking if the balance is enough for purchase. This goes against the principle of “Reduce Dependency” as well as “single-responsibility”, as it would have more dependencies and the Player class would be responsible for purchasing an item, which it shouldn’t be as it shouldn’t be responsible for the functionality of the coins/items it has.

Hence, we discard this alternative. Our final design has now aligned with the Reduce Dependency Principle and Single-Responsibility Principle. The application checks for actions allowed for the Wallet item stored in Player’s inventory, instead of Player adding the trading action to the action list inside the Player's own playTurn() function.
Both designs are alright and do not break any object-oriented principles.

We decided to create a Wallet item and add it to the Player's inventory because we ensure that the meaning of each class is clear and they can be reused and allows extensibility in other scenarios in the future (ie: if there is another actor who can store a Wallet item other than Player) (adheres to the DRY principle & SRP principle).

2: Coin, Wallet and AddToWalletAction
Here, a Wallet item is added to the player's inventory as a non-portable item. Whereas coin is added to the Wallet of the player rather than added to
the player's inventory.
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. Thus 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 coins and 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, this action class will be useful for future extensibility, if we have other forms of currency that can be added to the Wallet of the player other than coins.
This is in line with the "Open-Closed Principle", where this AddToWalletAction class can be extended to be used and to accomodate more functionality in the future, without modifying the class. Also by adding coins to the wallet, we don't destroy the coins straight away (ie: just keep the balance). Even though keeping the coins is not part of the requirement, we believe that keeping these coins as "items" in the wallet can help for future extensions, if the coins are to be used for other functionality besides just trading. (ie: a scoring system)

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

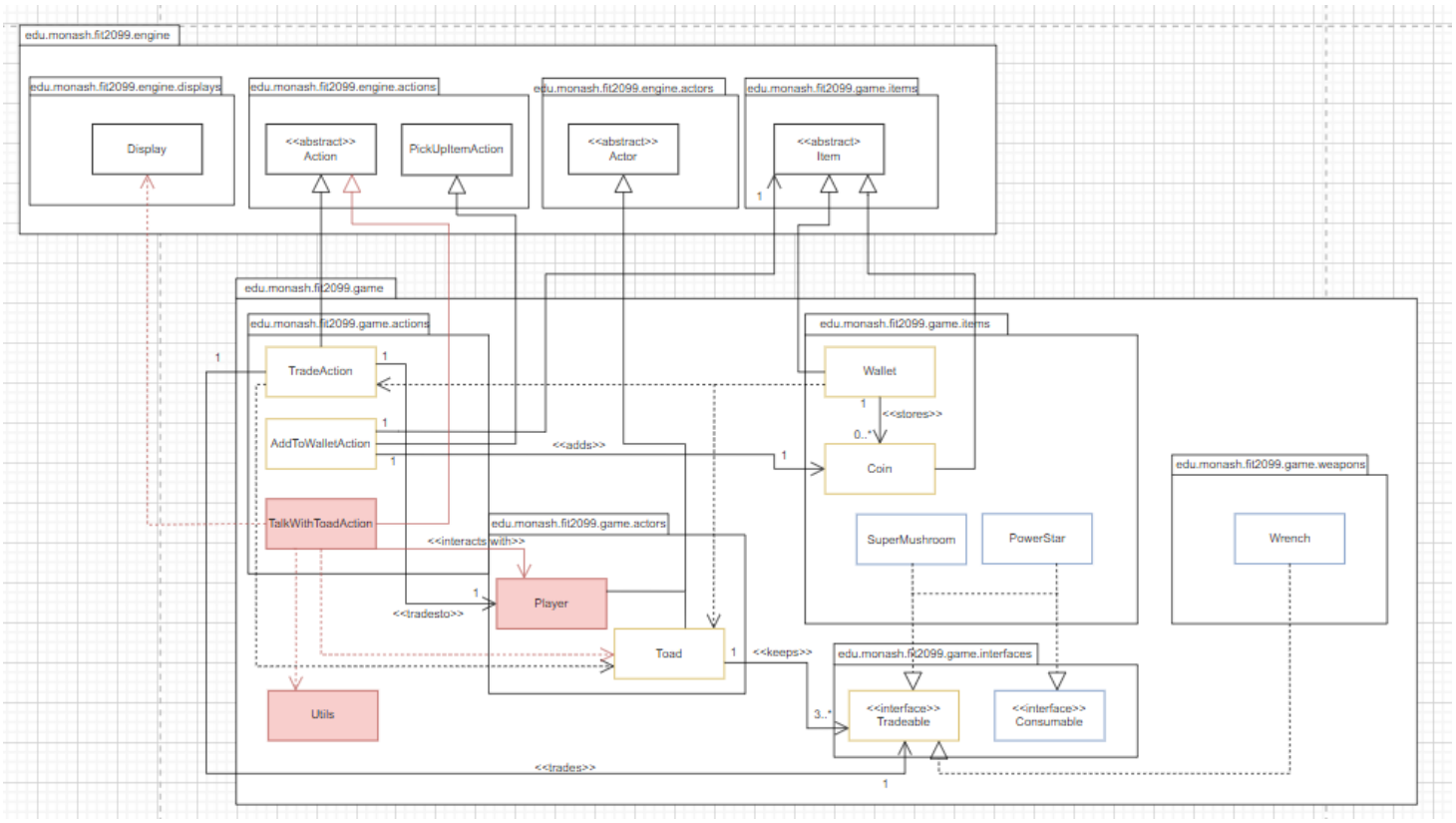
3: Toad and Tradeable
Tradeable is an interface by which the Toad singleton class keeps an arraylist of, and that the items or weapons that are tradeable will
implement this interface.

An alternative to doing this is for Toad to have attributes of Wrench, SuperMushroom and PowerStar (one of each type), to storethe current items he can trade with the player. However, this causes the Toad to have 3 associations (strong dependency), one to each of the tradeable items. This goes against the principle of “Reduce Dependency”, as Toad class would have more dependencies.

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" and the "Dependency Inversion Principle", where we are reducing the amount of dependencies/associations for Toad class (RDP) and 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.
Therefore, the above rationale is why we have the Tradeable interface.

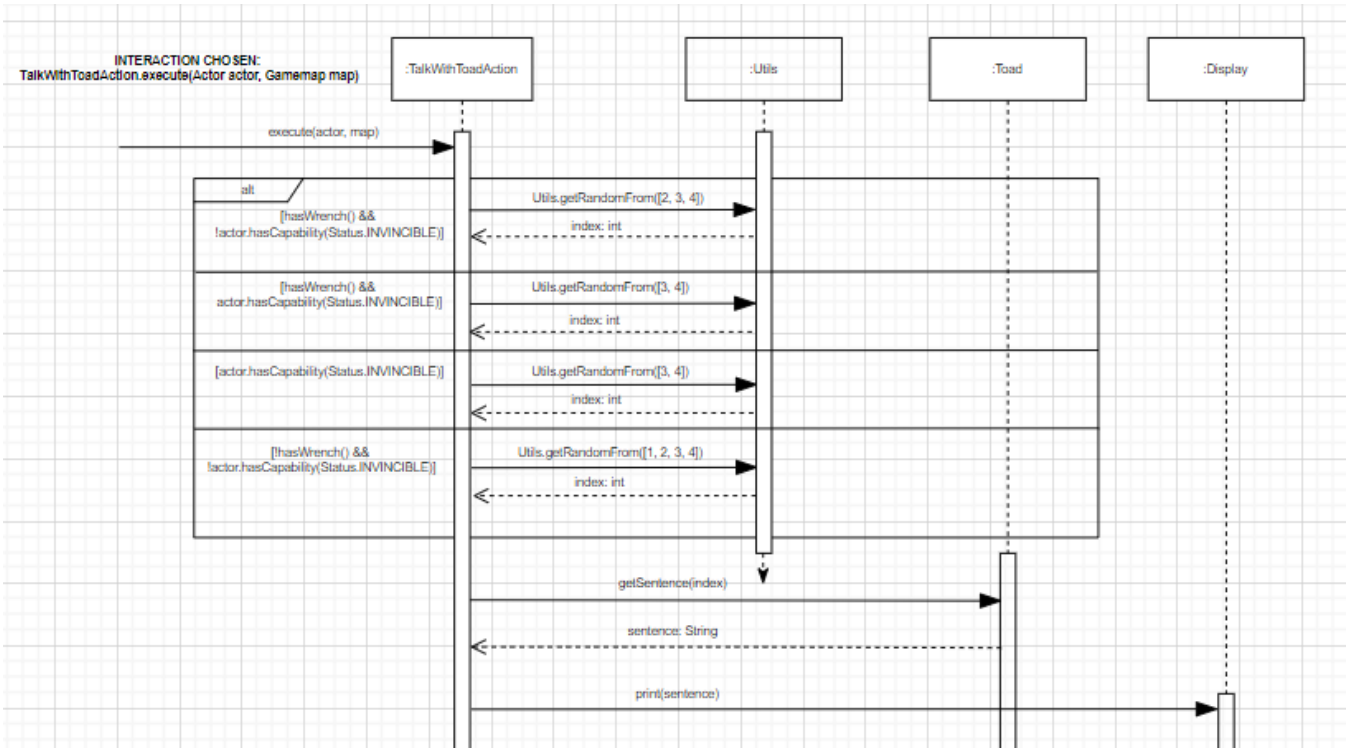
REQ6:

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



Overall Class Responsibilities
REQ 6 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.

Player
1. New Attributes:
ActionList additionalActions;
2. Constructor:
Modify constructor to:
- instantiate ActionList for additionalActions attribute.
- instantiate TalkWithToadAction & add this action to the additionalActions ActionList.

3. In playturn() method:
Modify to add additionalActions list attribute to the passed in ActionList argument.

4. Methods:
hasWrench() to return true if the player has a wrench in inventory, else return false.

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

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

TalkWithToadAction
1. Class Overall Responsibility:
This class is used to represent talking with a Toad Action. Note that 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 Player player;

4. Constructor:
creates instance of TalkWithToadAction with the player attached to it.

5. Methods:
overrides Action's execute method.

overrides execute() method:

- check if Player has a wrench.
- check if Player has powerstar effect. (player.hasCapability(INVINCIBLE)).

if player'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.

overrides Action's getHotKey() method to return the specific hotkey, which is: "d".

Utils
1. Class Overall Responsibility:

This class is used to provide utility methods which other classes can use.

2. Relationship with other classes:

None atm.

3. Attributes:

private Player player;

4. Constructor:

creates an instance of TalkWithToadAction with the player attached to it.

5. Methods:
overrides Action's execute method.

overrides Action's getHotKey() method to return the specific hotkey, which is: "d".

overrides execute() method:

- check if Player has wrench.
- check if Player has powerstar effect. (player.hasCapability(INVINCIBLE)).
if player'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.

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.

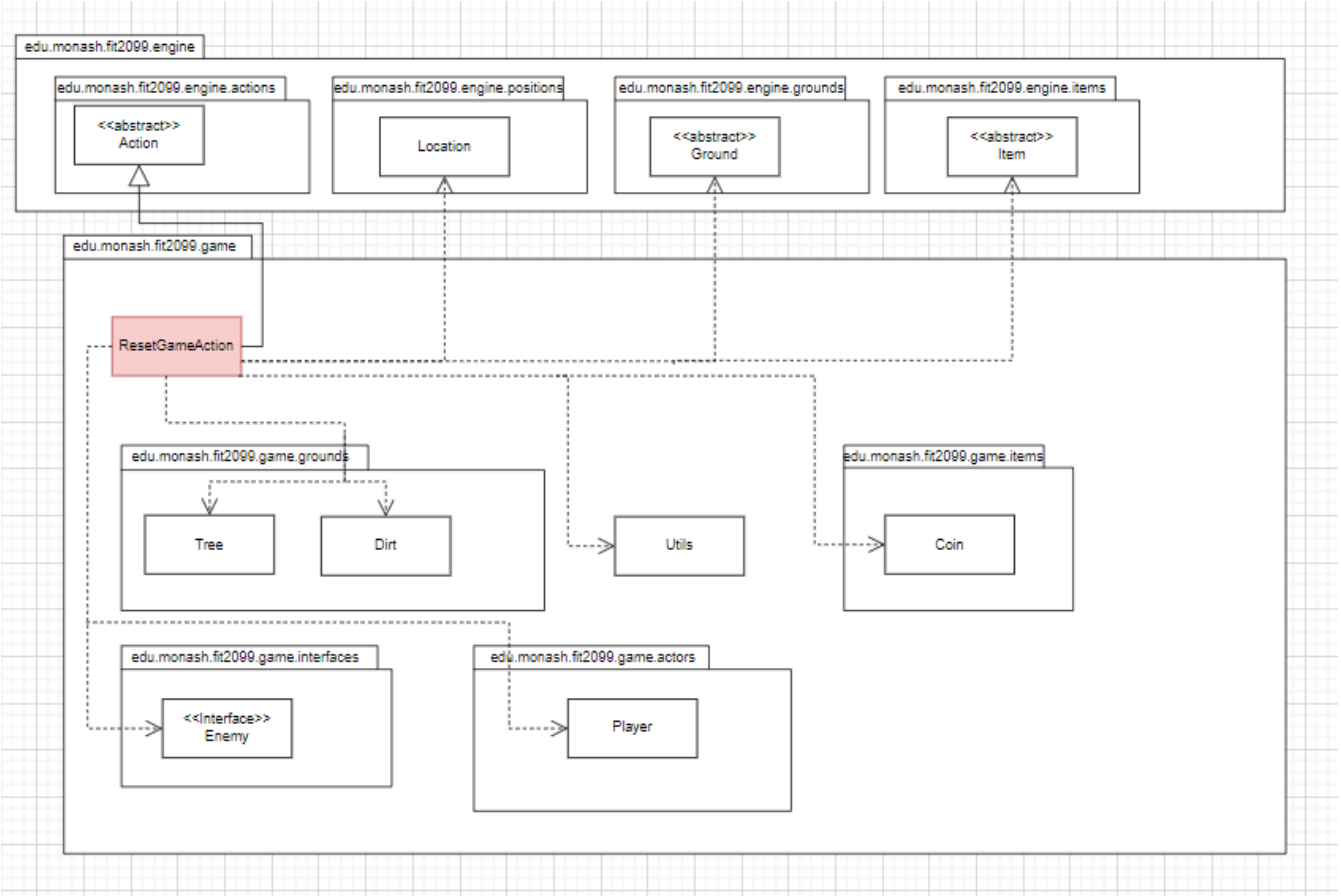
In this game, the ideal way to interact with the object is by attaching an appropriate action to its corresponding object (aligns with the meaning of "object-oriented").
Thus, this is shown with this: **Player ---<<uses>>---> Action** (dependency, not shown in the class diagram for REQ6 as it was already existing in the basecode), and
TalkWithToadAction ---<<interacts with>>---> Player (association).
Here we are adding an instance of TalkWithToadAction to the actionsList via the playTurn method in the Player class. 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, however it would cause Player to have an association with behaviour class, which is an added strong dependency for the Player class. Thus, would go against the "Reduce Dependency Principle." Note that both designs are alright and do not break any object-oriented principles.

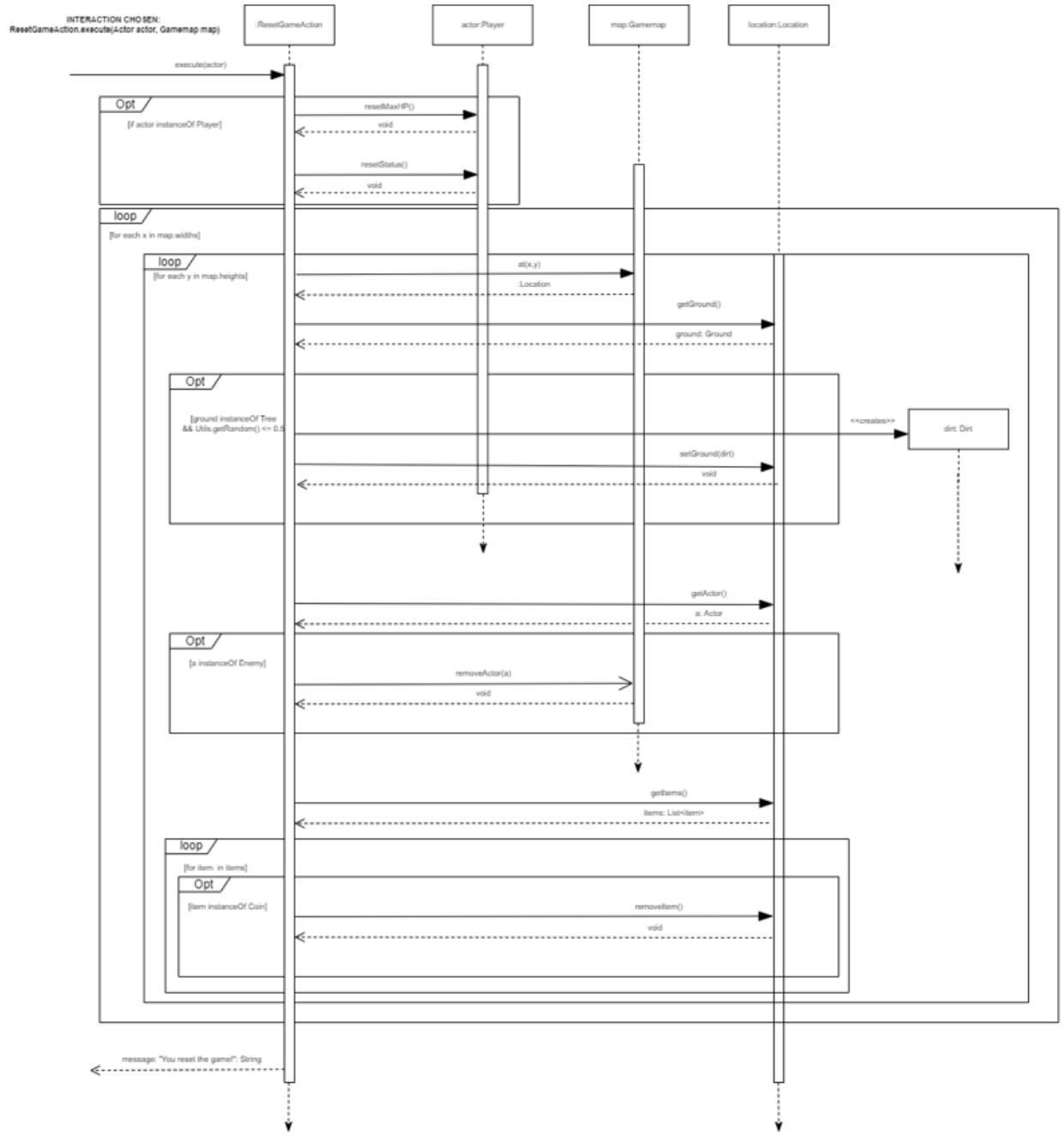
Thus, with our current design, we discard the alternative since it goes against the Reduce Dependency Principle, and also because we should not need a list of behaviours for the player to choose from (not in the requirements). So, in our design only TalkWithToadAction has an association to Player and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

REQ7:

Class Diagram



Sequence Diagram



Overall Class Responsibilities

Player class has added attributes and methods.
ResetGameAction is a new class.

Player

1. Attributes:
resetStatus boolean, true if game has been reset, false if not.
2. Methods:
resetPlayerStatus, to reset the player's status & remove any effect from SuperMushroom and PowerStar

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's resetStatus is false.

2. Relationship with other classes:
Dependencies on several classes (Enemy, Tree, Dirt, Player, Utils, Coin)

3. Attributes:
None

4. Constructor:
creates instance of ResetGameAction.

5. Methods:
overrides Action's execute method
overrides Action's getHotKey() method to return the specific hotkey, which is: "r".

in public execute() method:

- resets the Player's HP
- resets the Player's status
- get the width and height of the Gamemap map argument.
- loop through width and height to get each location in the map.
- for each location:
  if there is a Tree, convert it to Dirt based on a 50% chance
  if there is an Enemy, remove it from the map (kill it)
  for all coins, remove it

overrides Action's getHotKey() method to return the specific hotkey, which is: "r".

<u>Design Rationale</u>

1: ResetGameAction

To be able to reset the game, the player needs to have an option to reset the game.
Thus, we have the ResetGameAction that extends the abstract Action class.

Note that currently, ResetGameAction has dependencies to a few classes (Tree, Utils, Dirt, Player, Coin, Enemy). Thus, this may not be optimal, however we had decided to go with this approach due to these 2 benefits:
1. Straightforward approach.

For the execute() method to check for all trees and all locations, it needs to iterate through the map's width and height to get all the locations in the map. Then, for each location we will check for the Trees, the Enemys and the Coins in each location to perform the require resetting for each of them if they exist at the location.
Thus, note that it allows for a straightforward approach if the execute() method does this.
2. Single Responsibility Principle.
Currently, we want the resetting of the game to be done by a single class, and therefore, we have the ResetGameAction class that is added to the player's actions list if the player has not yet reset the game. Thus, we believe having the checking and dependencies within the ResetGameAction allows for the responsibility to be completed in a single class.

Therefore, due to the rationale above, we believe that this approach even with its introduced dependencies make the implementation straightforward and supports the single responsibility principle and thus we decided to go with this implementation.