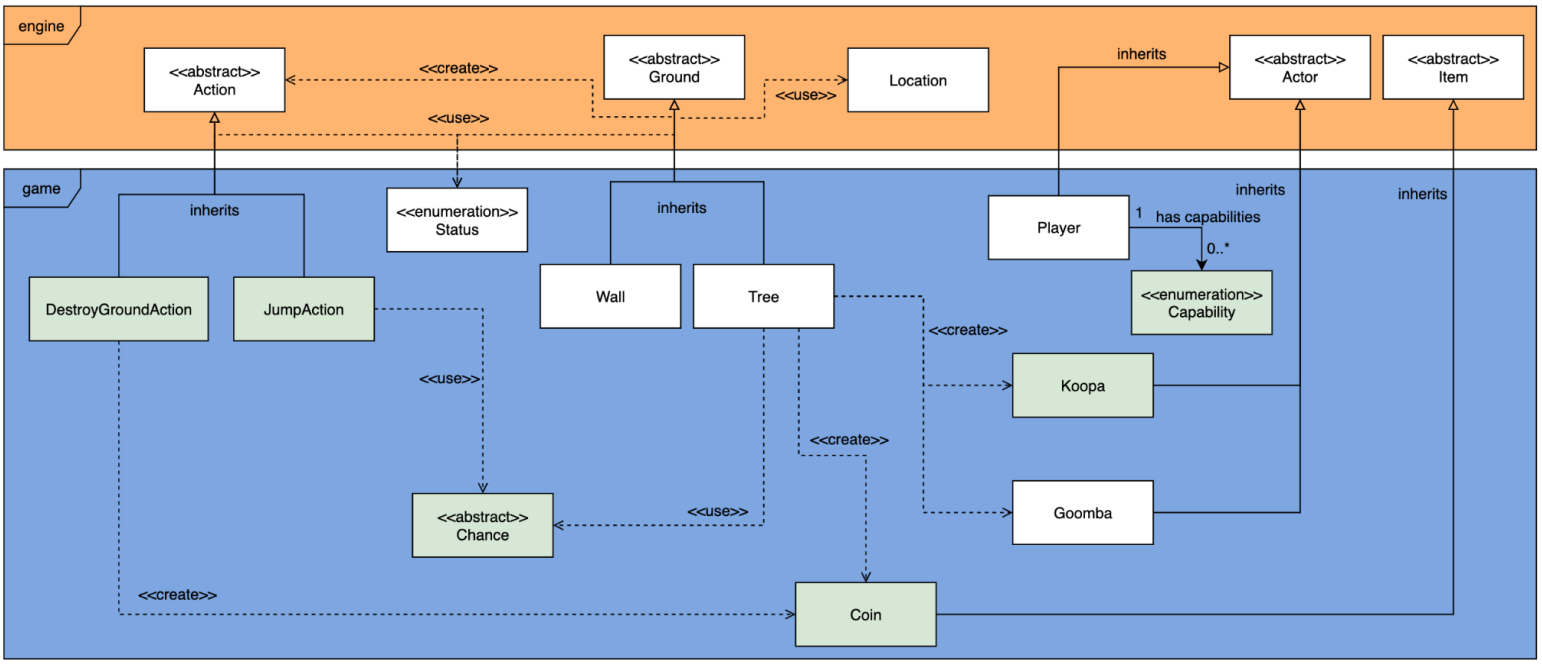
FIT2099 Assignment 1: Design Rationale

Assignment 3 updates found under each section

Extending Tree and Adding Jump Feature



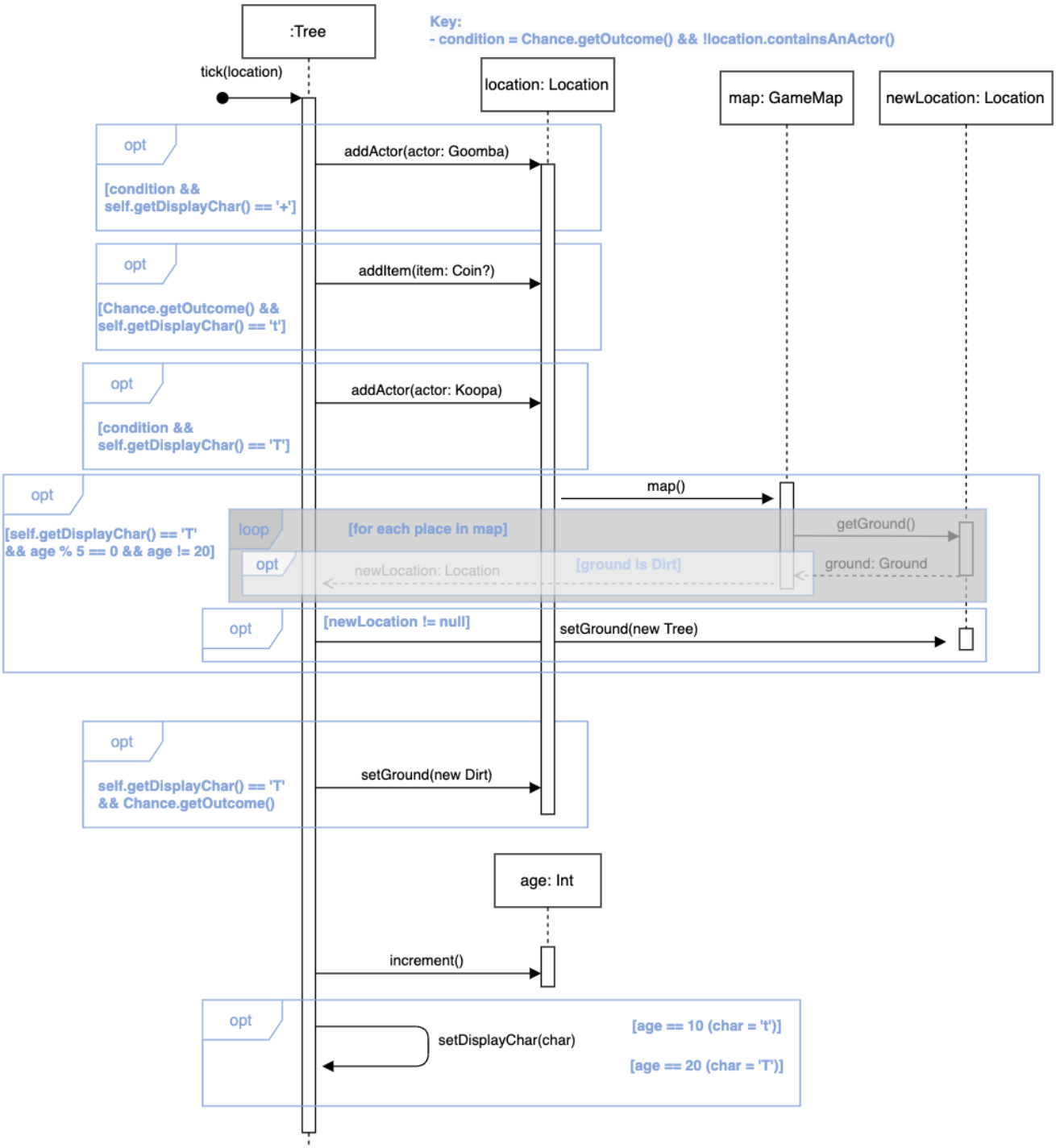
Chance Class: a new abstract class with a static method *getOutcome* that uses *Random* to return a boolean based on how likely an event is. This class has been added to increase abstraction and reduce repetition of having to calculate if an event will go ahead or not depending on its percentage change of success. Thus helping the design conform to the Single Responsibility Principle.

Tree class : this class will be extended to accommodate the new features required. This involves overriding the *tick* method from the *Ground* to let it experience time. Inside this method, the *Chance* class will be used to determine what the *Tree* decides to do on a given turn, with an *age* attribute to maintain how many turns the *Tree* has been alive for. This class will depend on *Chance, Koopa, Goomba* and *Coin* classes in order to perform the required actions. Refer to the sequence diagram below for how the *Tree* decides what to do on each turn.

Update:

There is no longer a Chance class in our design. Since the purpose of Chance class was to only calculate the success of an event which could be easily done with *Math.random()*100 <= (the probability of the event occurring),* the Chance class is no longer necessary in our design.

Sprout, Sapling and Mature classes were created that inherit the Tree class which became abstract. This is due to the fact that all of the types of trees have very similar functionality with only a few key differences. By adding these extended classes, each class is responsible for only what happens to itself (rather than having Tree be responsible for each of the three types).



Capability enumeration : a new enumeration to store capabilities that an *Actor* can have, in this design, this enumeration will be used to give *Player* a *JUMP Capability* in order to be able to jump when approaching high ground or trees. This enumeration has been added as it reduces the use of ‘magic strings’ and whilst at the moment it only contains one ‘element’ it can be easily extended if more functionality is required in future developments. This

enumeration helps the design conform to the Liskov Substitution Principle as it eliminates the need for *instanceOf* or *getClass().getName()* in order to determine if the specific *Actor* can jump.

JumpAction class : a new class that extends *Action* that will be used to give an *Actor* the ability to jump to higher ground. The implementation of this class is very similar to the *MoveActorAction* class, with the main differences being in the *execute* method. This method will use the *Chance* class to determine if the jump is successful, and moves / hurts the *Actor* depending on the result. This class will be called inside the *Wall* and *Tree* classes, where the *canActorEnter* will be altered to return true and the *allowedActions* method from *Ground* will be overridden to add an instance of *JumpAction* to the *ActionList* before returning it.

DestroyGroundAction class : another new class that extends *Action* that is used when approaching high ground only when the *Actor* has a power star (status is *Status.POWER*). This status will be checked for in the *allowedActions* method in *Tree* and *Wall*, where an instance of this class will be returned instead of *JumpAction*. The setup of this class will be the same as *JumpAction*, with the only exception being the *execute* method, where the current location's *Ground* will be set back to dirt, money will be dropped and the player will get moved.

These two *Action* classes have been added to separate the functionality, where each of these classes has one responsibility delegated to it, the fact that they extend action mean that they can work in the system without having to change it and can be easily adapted to for any further extensions of this project.

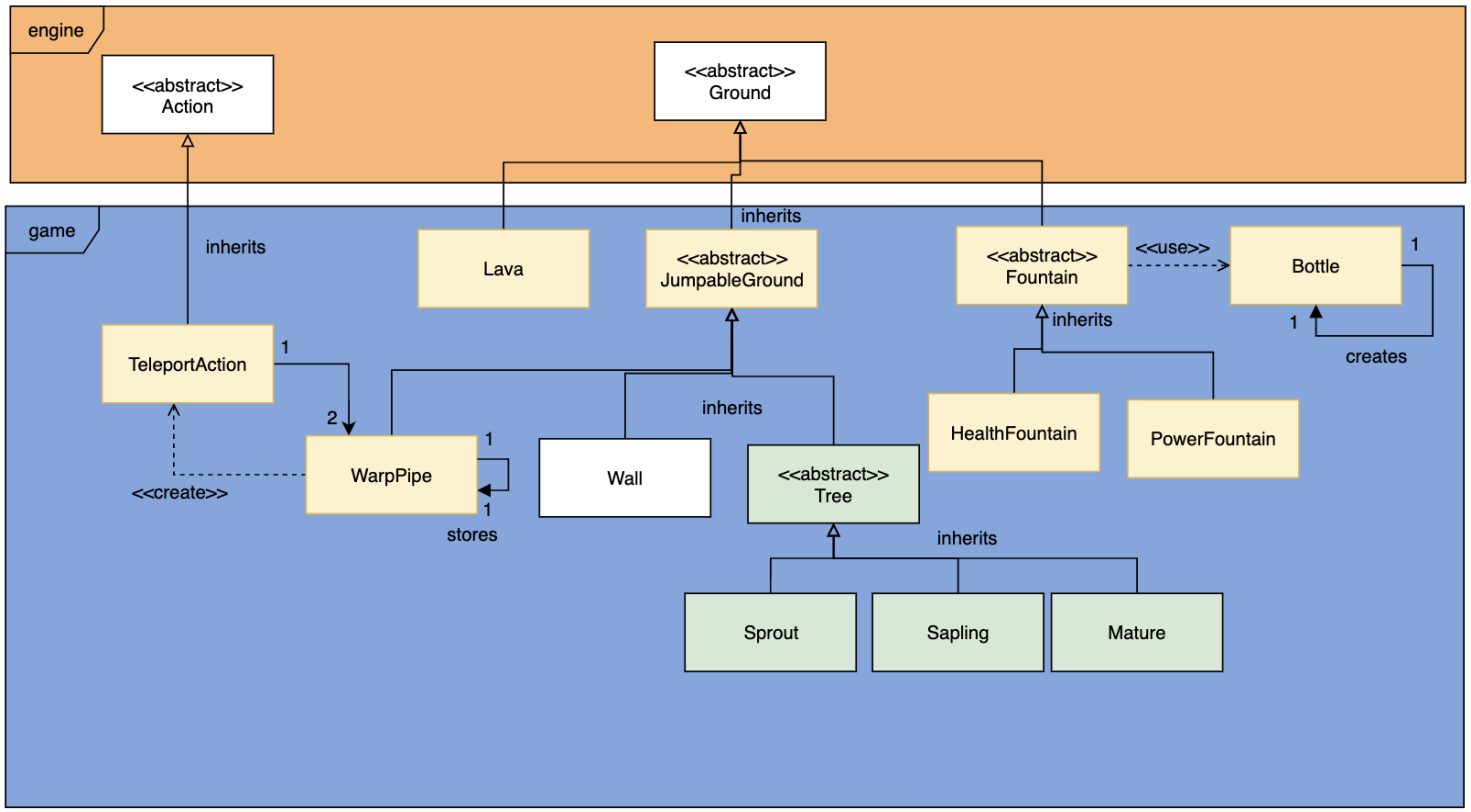
Update:
Added JumpableGround interface to deal with all of the Ground subclasses that can be jumped upon by actors. This interface allows the JumpAction to not have to know exactly which Ground the Actor is jumping over, just that it is jumpable. This allows much easier addition of a new Ground subclass that is jumpable, as it simply just needs to follow the interface and add a JumpAction and no changes will be needed to be made to the JumpAction class itself. This helps our design adhere to the Open / Closed principle.

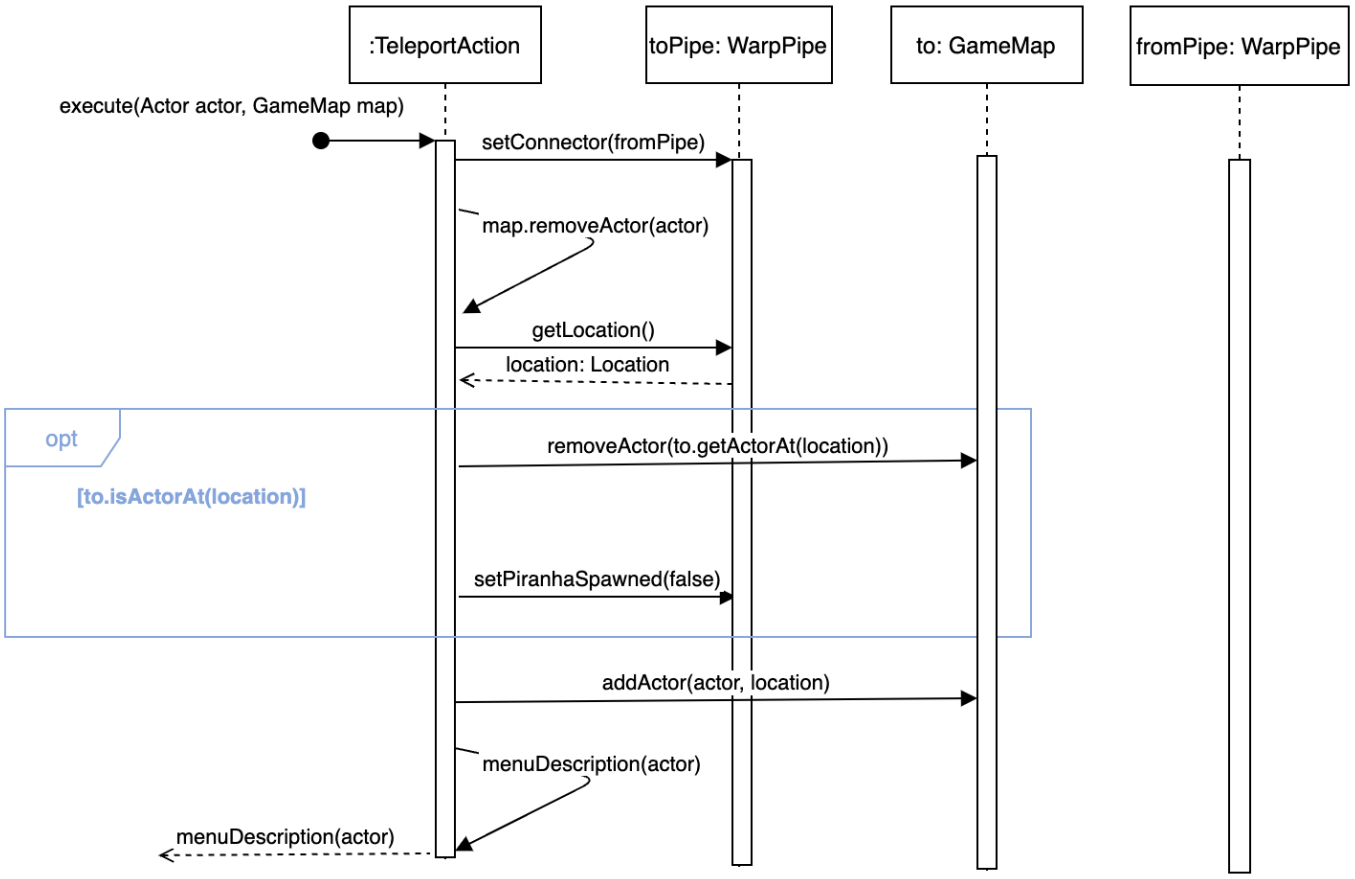
Assignment 3 Update:

Lava added as a new Ground child to model in-game lava, with overrides on the tick method to hurt an Actor (that is not an Enemy) if it is stepped on. By extending Ground this design follows the Open-Closed principle, as new features were added without having to modify any existing code.

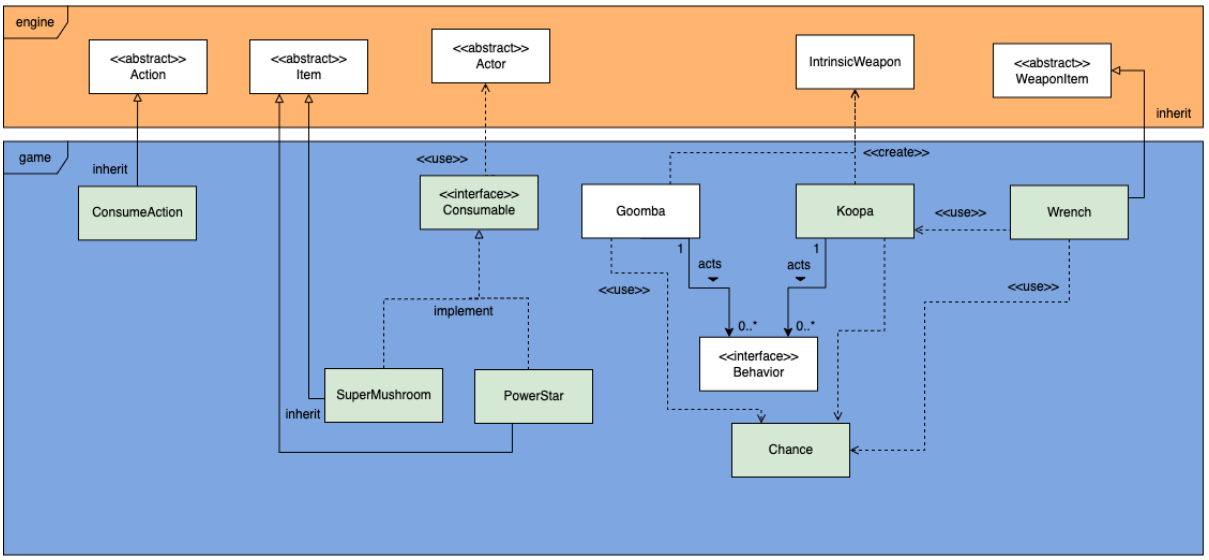
JumpableGround has been converted to an abstract class inheriting from Ground, this is due to the fact that there was repeating code inside Tree and Wall in the allowableActions method. By making this class abstract this repetition of code was enabled to be removed.

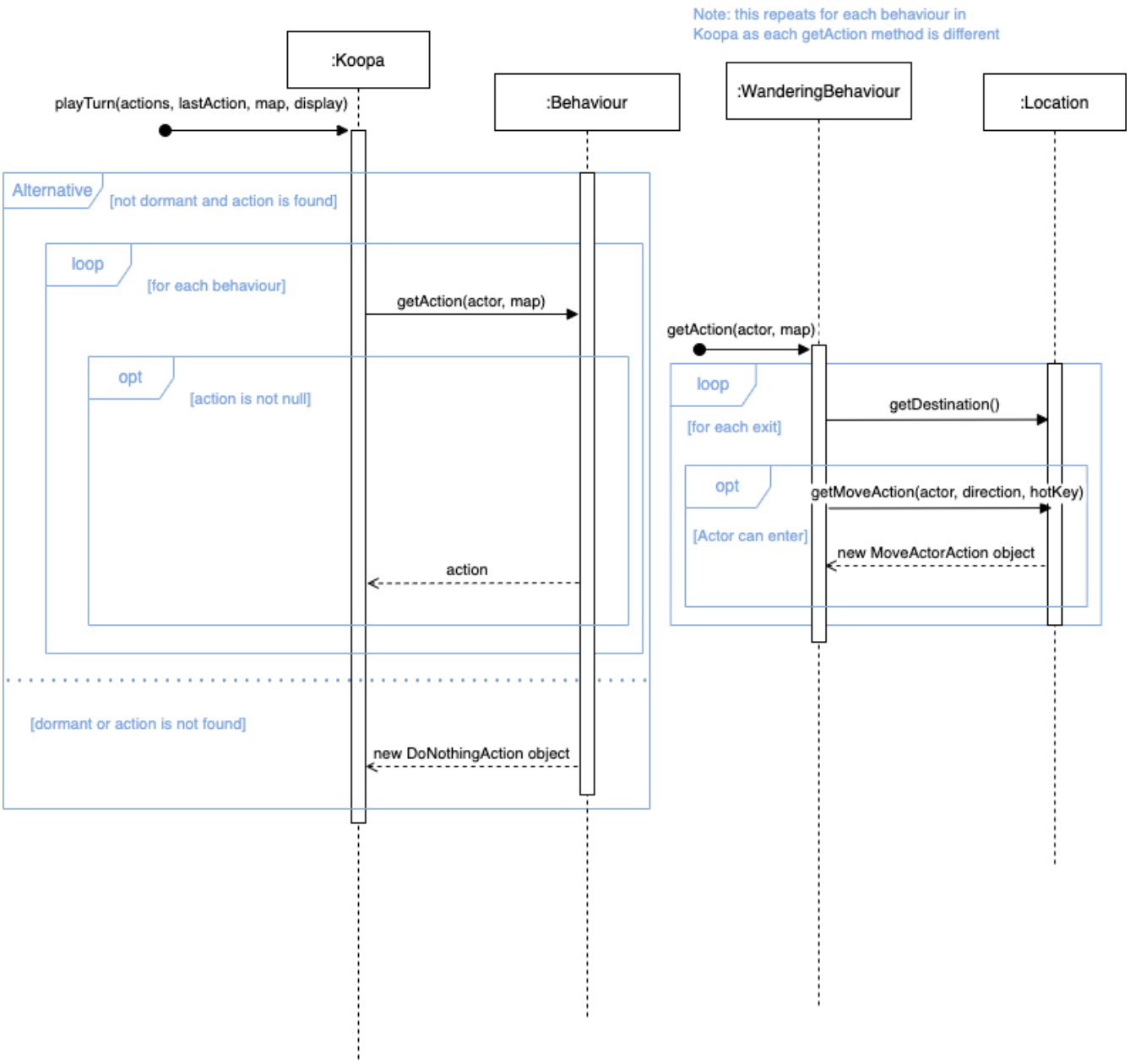
A new WarpPipe class was created to model the warp pipe that allows teleportation between maps. This class extends JumpableGround, to allow it to have the features of being able to be jumped on. The WarpPipe class stores another instance of itself inside it, this enables two WarpPipes to be connected. This design decision ensures that travelling to the Lava Zone and back will always bring you back to the WarpPipe you originally teleported from. To handle the teleportation, TeleportAction was added as a child of Action. This enables the teleportation to work within the already existing system without having to change the system itself, thus adhering to the Open-Closed principle. A new sequence diagram illustrating the teleportation process is shown below.





Enemies and Items





Goomba class: Since Goombas attack with a kick that deals 10 damage with 50% hit rate, this class will override the getIntrinsicWeapon method from the Actor class, which will return a new IntrinsicWeapon object with the following kick characteristics. The Goomba also has a 10% chance of dying and will immediately be removed from the map once defeated. Hence, a Chance class is used to determine if the Goomba will be removed from the map by suicide, and a parameter of type GameMap is needed to remove the Goomba from the map if it is unconscious.

Koopa class: This new class is a subclass of Actor class that represents all of the Koopas in the game. Similarly to Goomba class, this class contains a HashMap of type Integer and

Behaviour that lists all of the behaviours of a Koopa (WanderBehaviour, FollowBehaviour, AttackBehaviour). Koopa also overrides the getIntrinsicWeapon method from the Actor method because Koopa attacks with a punch that deals 30 damage with a 50% hit rate.

Wrench class: This new class is a subclass of WeaponItem class that represents all of the wrench items in the game. Since a wrench is the only weapon that can damage a Koopa's shell, a new method is added in Wrench class that damages the Koopa's shell that deals 50 damage with an 80% hit rate. And because there is a 80% hit rate, this class also uses Chance class to determine if the wrench successfully hits the target. This part of the design conforms with the Single Responsibility Principle since the Wrench class has its own responsibility of being a type of WeaponItem that causes damage.

Interface *Consumable:* Consumable is a new interface that is implemented by both the SuperMushroom and PowerStar class used to distinguish consumable items from the general items in the game. In this interface, a new method is created for the items to be consumed by an Actor. Another method is also added in this interface for the Player to gain a ConsumeAction for the user to choose which item to consume. This part of the design applies to the Open-Closed Principle by extending the functionality specifically for consumable items without modifying the Item class.

ConsumeAction class: This is a new child class of Action class that is a special action for Actors to consume an item. ConsumeAction class contains an attribute of type Item of the item that is being consumed, which shows an association relationship between ConsumeAction and Item. And because this class needs to know which Actor is consuming the item, there is also a method that passes an Actor as a parameter into the consumedBy method from the interface Consumable.

SuperMushroom class: SuperMushroom is a new child class of Item class that implements the Consumable interface which represents all of the super mushroom items in the game. In order to know which Actor has consumed the SuperMushroom and needs to be upgraded, the class changes the status of the Actor to *TALL* from the enum status, the max HP of the Actor increases by 50, and the JumpAction class ensures that the Actor jumps with 100% success rate and no fall damage.

PowerStar class: PowerStar is a new child class of Item class that implements the Consumable interface which represents all of the power star items in the game. Because PowerStar only lasts for 10 turns, this class overrides the tick method from Item class to take count of the number of turns before the power star disappears. Another status called *POWER* is added into enum Status to signify that an Actor consumed the power star, which notify other Actors when attacking or being attacked by that Actor with the *POWER* status.

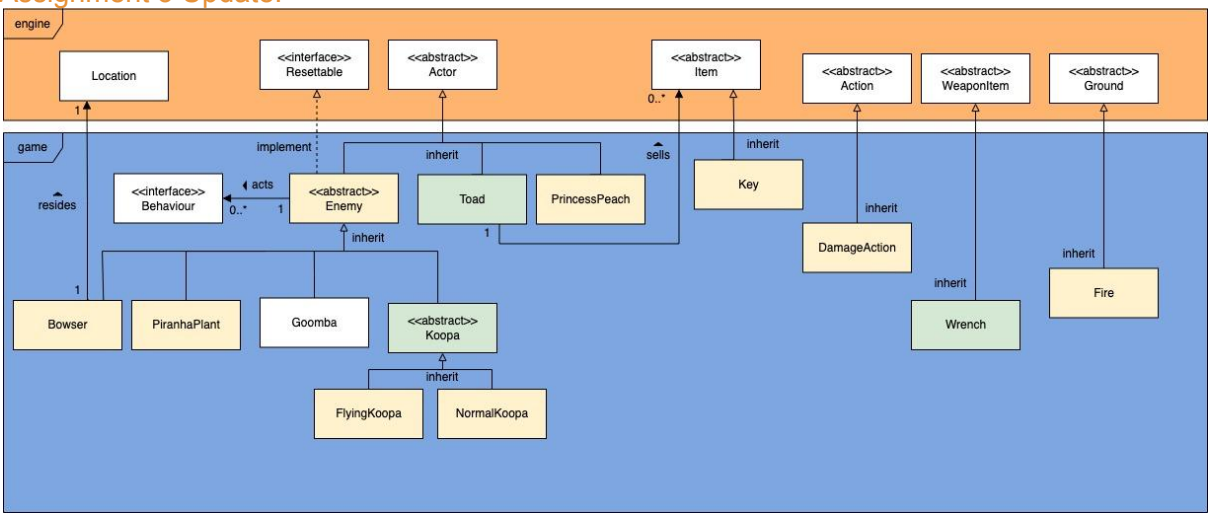
Update:

There is a new abstract class called Enemy in our design which Koopa and Goomba inherits. Because Koopa and Goomba possess similar attributes with different values and perform the same actions but in different ways, polymorphism is applied in this updated design by having a class that all types of enemies can relate to and inherit.

The class DamageAction that inherits Action class was also created in the design. Since Koopa has a special state of being dormant and it is only possible to destroy the shell by damaging it with a Wrench, a new class was created for the action of damaging the Koopa's shell with a Wrench. This applies to the Single Responsibility principle as this helps separate the concern of dealing with the specific case of damaging a Koopa's shell to DamageAction class.

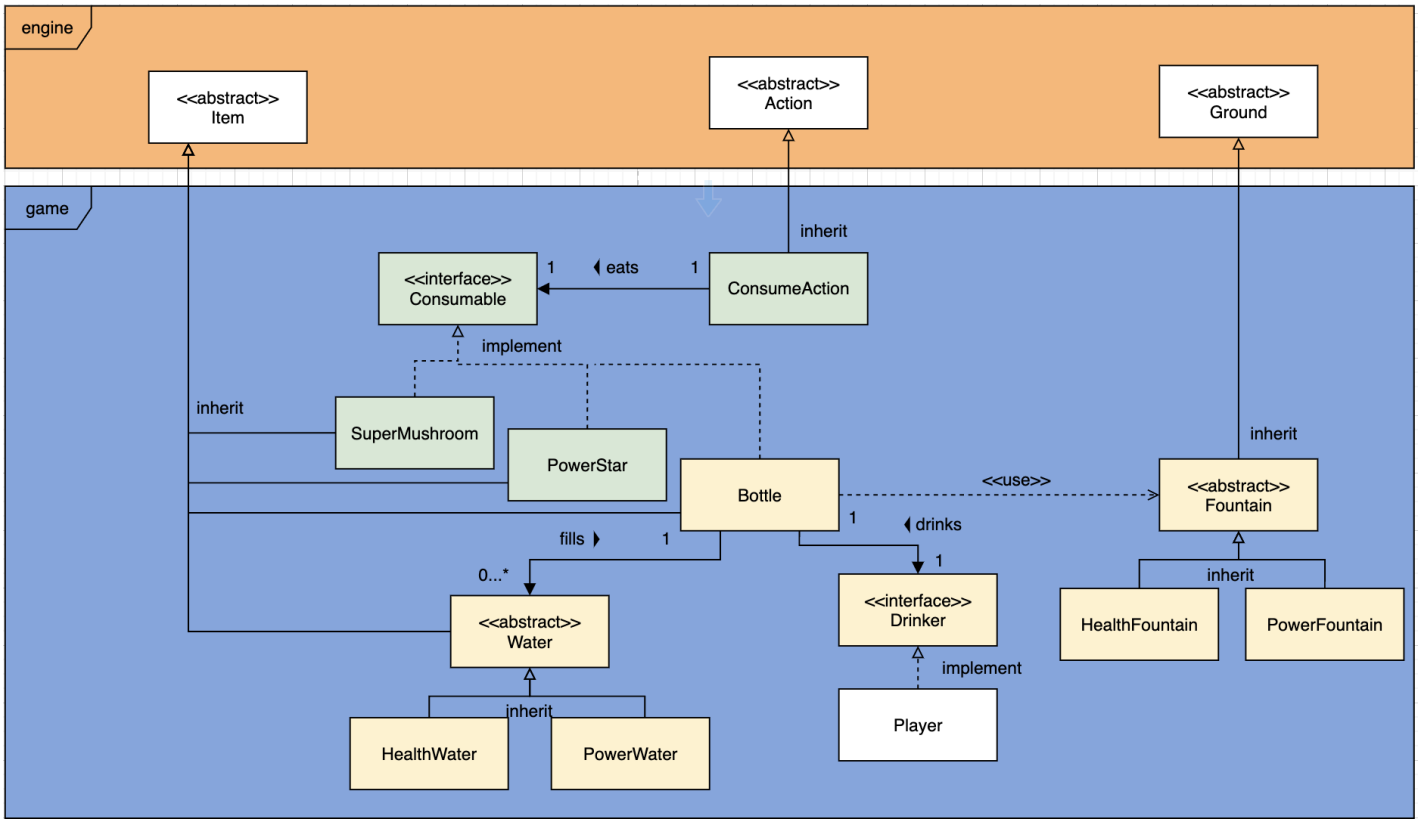
The enumeration BehaviourPriority was added to the updated design. BehaviourPriority is used to provide meaning to the values by representing them as priorities of each behaviour. If more behaviours were to be created as further implementation, the priority for the new behaviours can be easily set. Furthermore, if modifications need to be made to the priority of a behaviour, it can be also easily done in BehaviourPriority.

Assignment 3 Update:



The addition of the classes Bowser, and PiranhaPlant were added that are inherited from the abstract Enemy class, which was created during the implementation in Assignment 2. They were added as child classes of Enemy because they are considered new enemy characters of the game and can override and add new methods that are appropriate for the different types of enemies. This helps add new functionality without editing existing code (adhering to the Open-Closed principle). Since Enemy class implements Resettable, each class of an enemy does not need to implement Resettable individually. When resetting the game, since Bowser needs to return to its original location, there is an attribute of type Location in the Bowser class, which creates an association relationship between Location and Bowser. Furthermore, because there are two types of Koopas - a flying Koopa and a normal Koopa -, the Koopa class is made abstract and two new classes called FlyingKoopa and NormalKoopa extend the abstract Koopa class. Implementing this design prevents more modification inside the Koopa class and instead extends its functionality by having different classes represent different types of Koopas. Hence, this adheres to the Single Responsibility principle. PrincessPeach class is also added as a new character in the game that extends Actor.

The Key class is created as a singleton that extends Item because there is only one instance of key which is used to save Princess Peach and have her say the victory message, indicating that Mario wins. The Fire class is added which extends Ground as it hurts anyone who steps on it. Although it can be seen as a weapon that harms any Actor that walks over it, since it can't be picked up or dropped and is placed on the ground, Fire inherits from Ground instead of WeaponItem. Creating the Fire class takes care of the additional feature of dropping a fire whenever Bowser attacks, which adheres to the Single Responsibility principle.



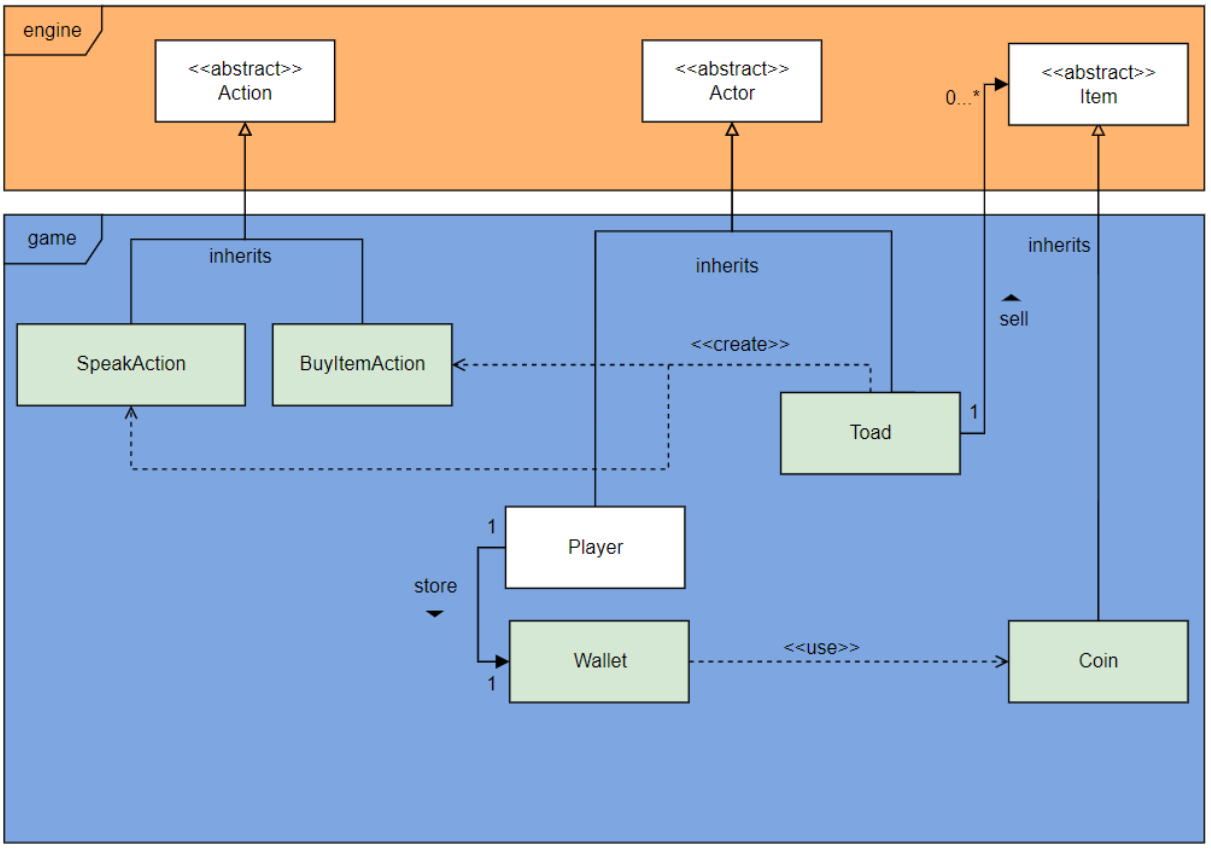
The Bottle class created was made into a singleton because in this game, only the Player can have a bottle that can be refilled and consumed. In the Bottle class, there is an attribute of type Drinker that represents the Drinker that is consuming the bottle and gaining the effects of the water. The class also has another attribute of type interface Water which is an ArrayList that indicates the type of water that is filled in the bottle. The interface Drinker is added because it allows all Actors that can drink to implement the methods that are needed. This helps conform to the Interface-Segregation principle as this feature of having some actors be drinkers is isolated. This also helps conform to the Dependency-Inversion principle as the Bottle doesn't have to rely on the concrete class Player, instead relying on the interface.

The abstract class Fountain is added that inherits from Ground to allow it to be treated as such in the system. With the additional features, no existing code needs to be updated in order for this class to become part of the system, thus maintaining the design's Open-Closed principle. The two types of fountains - HealthFountain and PowerFountain - inherit from Fountain because both fountains, whilst having very similar fountain-like features, differ slightly. An original thought was to have one fountain and have an attribute for the type of fountain, and actions were to be based on this type. However, our group decided that this did not conform to

the Single Responsibility principle and therefore two separate classes were added and Fountain was created as abstract. This also allows Bottle to rely on an abstraction rather than a concrete class, adhering to the Dependency-Inversion principle.

To accompany the Fountain, a new abstract Water class with two children was created. This abstract class was discussed to be an interface instead, due to the fact that Water only declares an abstract method, however this was decided against as Water is an Item and in the old case, HealthWater and PowerWater were both extending Items. Adding this as abstract adds another layer of abstraction, enabling further implementation to be easily added if the general Water class needed to have extra attributes or if the Water class needed to interact with the Item abstraction. If this was the case with the old design, both HealthWater and PowerWater would have to interact with the Item abstraction, creating repeated code. Thus Water was made abstract. Having each case of HealthWater vs PowerWater instead of just an attribute of Water being the type of water helps the design adhere to the Single-Responsibility principle.

Trading and Monologue

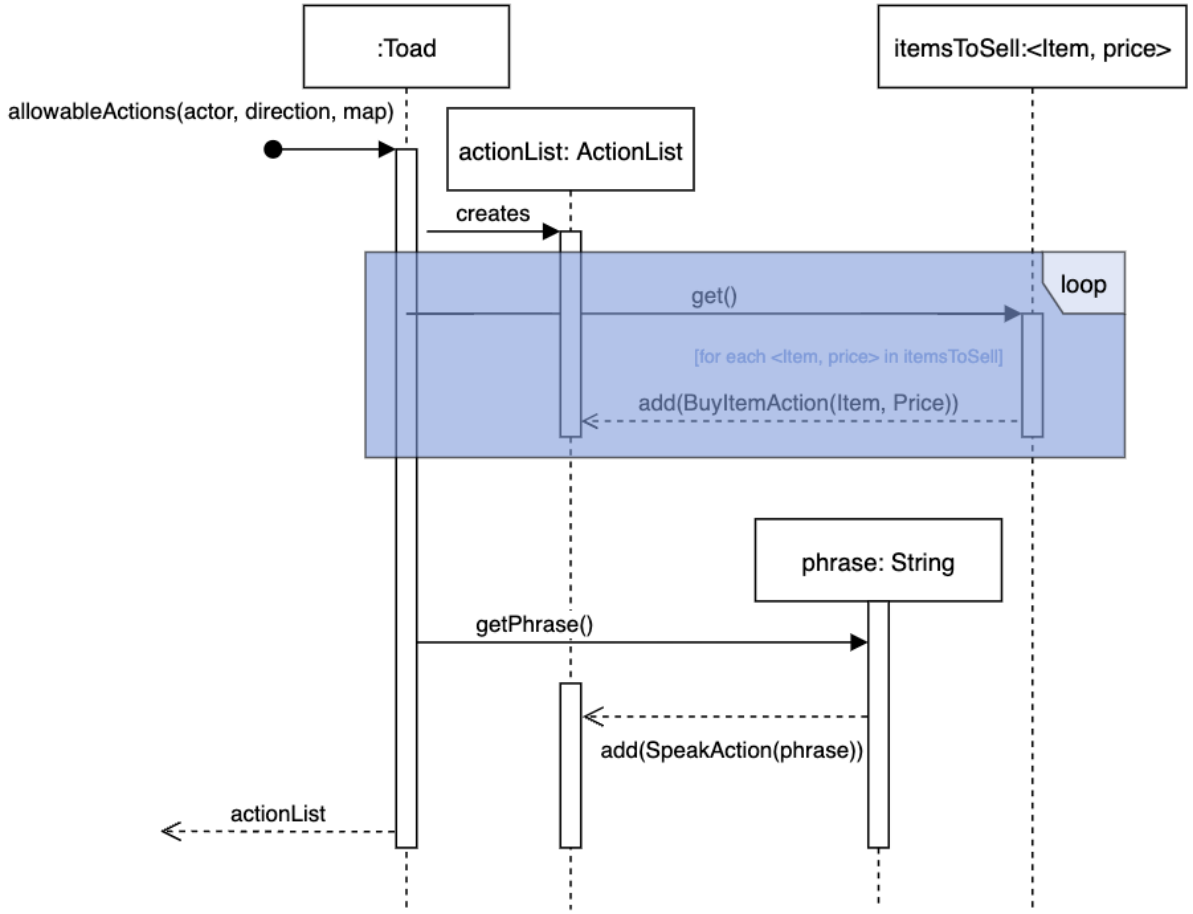


Coin class : a new class that extends the *Item* class that is portable and has an added attribute to store the value of the *Coin* in dollars ($). Picking up the *Coin* is different from any other item as it requires an instant change in the *Actor*'s *Wallet*. For this to be done a new capability *WALLET_STORED* will be added to the *Capability* enumeration with this *Capability* getting added to the *Coin* in the constructor, then the *addItemToInventory* method must be overridden in *Player* to check if the *Item* can be stored in the wallet using the enumeration, and if so update the *Wallet*, otherwise add the item to the *inventory* as normal. This part of the design helps to conform to the Liskov Substitution Principle as it eliminates the need for *instanceOf* or *getClass().getName()* to determine if the *Item* can be added to the *Wallet*. This class acts to allow the user to add to their *Wallet* and allows the *Item* to be dropped during various parts of the game.

Wallet class : a new *Wallet* class to store the money for the *Actor* and handle all of the money events (getting current balance, adding to the balance, reducing the balance and checking if the balance is sufficient. This is created to allow all actions related to money to be stored in a single place and enables easy further expansion of the game if there was to be more than one *Actor* that could have a *Wallet* thus. This *Wallet* will be an attribute of the *Player* class, with the *Player* class having a getter for this *Wallet*.

Toad class : a new class that extends *Actor* that is used to represent Toad. This class will have two attributes, an *ArrayList* of possible phrases spoken, and a *HashMap*, containing both the *Item* and the integer price of that *Item*. To be able to buy an *Item*, this class will override the *allowableActions* method to iterate over the *HashMap* and for each *Item* adding a *BuyItemAction* to the initially empty list of actions. To be able to speak to Toad, a method will be created that will determine by random which phrase will be spoken (taken into consideration the player's inventory), once the phrase is chosen it is passed into a new *SpeakAction* which is again added to the list of actions. The *allowableActions* method is detailed in a sequence diagram below. The design of this class and phrase selector method

allows *Toad* to be responsible for what it says and all of its information inside its own class, helping the design conform to the Single Responsibility Principle.



BuyItemAction class : a new class that extends *Action* that represents the action to buy an item when closeby to *Toad*. This class has two additional attributes that are the price of the *Item being bought and the Item* itself. The *execute* method in this class will be overridden to check if the *Actor* can buy the product, and if so add it to *inventory* and subtract funds from the *Wallet*. If the *Actor* has insufficient funds then this method will return a meaningful statement informing the user of this.

SpeakAction class : another new class that also extends *Action* that represents the action for *Toad* to speak when prompted to by the user. This class will contain an attribute for the phrase that will be spoken and its *execute* method will simply print the phrase.

These two new *Action* childs are helpful to separate the specific actions that the user can perform whilst also maintaining the current system.

Update:

There is a new abstract class WalletItem which extends Item. Coin inherits this class, giving it the capability WALLETABLE. This lets us check if the item being added to inventory has this capability and thus lets us add the item to inventory or to the Wallet accordingly (only Coin is WALLETABLE as of now). This also lets us avoid downcasting and using instanceOf as we can check if the Item can be accepted to the wallet using enumeration.

Assignment 3 Update;

Deletion of the WalletItem class, instead a new PickUpCoinAction that extends PickUpItemAction has been added to manage adding the coin to the wallet when it is picked up. This allows the picking up of the coin to work within the system without having to downcast or edit existing code. Thus adhering to the Open-Closed principle. Wallet is now WalletManager, since 'Wallet' is more of an item when in actual fact the class was just a manager for the wallet value.

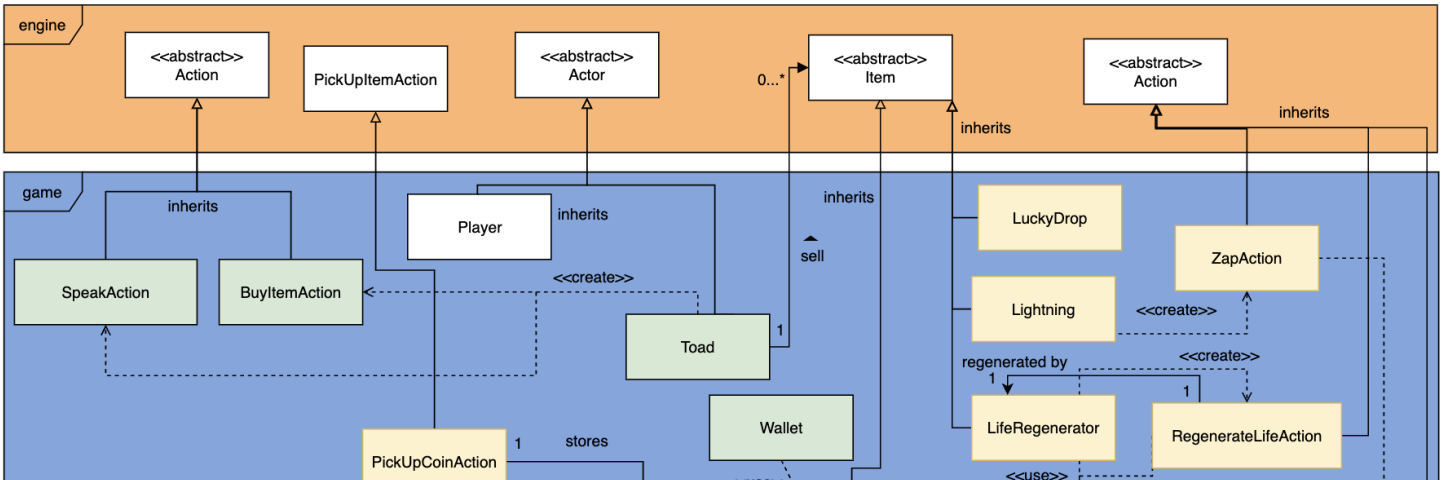
The Lightning class has been added as a new sellable item, that when used, decreases enemies HP by half. This is done by creating a new Zappable interface used to represent the objects in the game that can be zapped by lightning. This new interface helps with Interface Segregation as it only covers one specific feature of the game (rather than adding this feature to an existing interface). A ZapManager class has also been added in order to store all instances of Zappable (similar to the reset feature), this class is assigned the responsibility of performing the zap on all of the Zappable objects, helping the design adhere to the Single Responsibility principle. To perform the 'zap' the Lightning item adds a new ZapAction class, this action calls upon the ZapManager to 'zap the map' and since ZapAction is a child of

Action, it is able to integrate into the system smoothly without any modifications needing to be made. Therefore adhering to the Open-Closed principle.

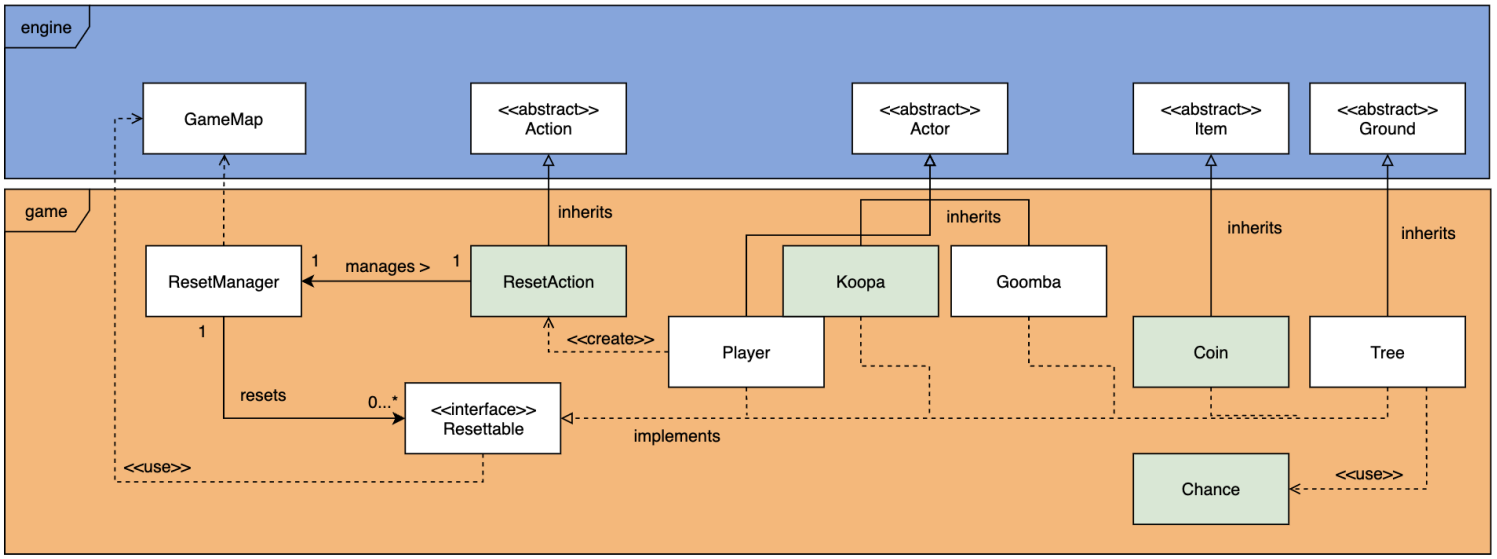
A new LifeManager has been added to add a new feature that allows Mario to have lives. This feature also prompted a new DieAction class that is used to do the required actions once an actor dies. This DieAction helps to reduce repetitive code and also helps the design adhere to the Single Responsibility principle as the AttackAction class now no longer has to deal with what happens when an Actor dies, that task is delegated to DieAction. This DieAction interacts with the LifeManager when the Actor has lives and removes a life (or kills the Actor as normal if they have no lives left). The LifeManager class is solely responsible for storing the lives of the player, again helping to adhere to the Single Responsibility principle.

This LifeManager is used with a new Item that Toad sells, the LifeRegenerator. As the name suggests this Item can be bought and used by the Actor to regenerate a life. To do this is used a new RegenerateLifeAction class which again, since it extends Action, can be easily implemented in the current system, thus adhering to the Open-Closed principle.

A third and final new Item has been added to the list of items that Toad sells, the LuckyDrop. Much like the question mark box in Mario Kart, the LuckyDrop can be obtained (through purchase) and then consumed to get a random item from the list of items that Toad sells. We decided to put this functionality into its own class rather than to have it all in Toad to help adhere to the Single Responsibility principle, it also declutters and simplifies the design. When Toad creates this item, it passes in all of its current items that it buys so that the LuckyDrop can pick between these items. To consume this item, we reuse the Consumable interface created as part of Assignment 2, this helps adhere to the Open-Closed principle as we are adding this new functionality of being able to consume the LuckyDrop and the effects of it without needing to change the current design.



Resetting the Game



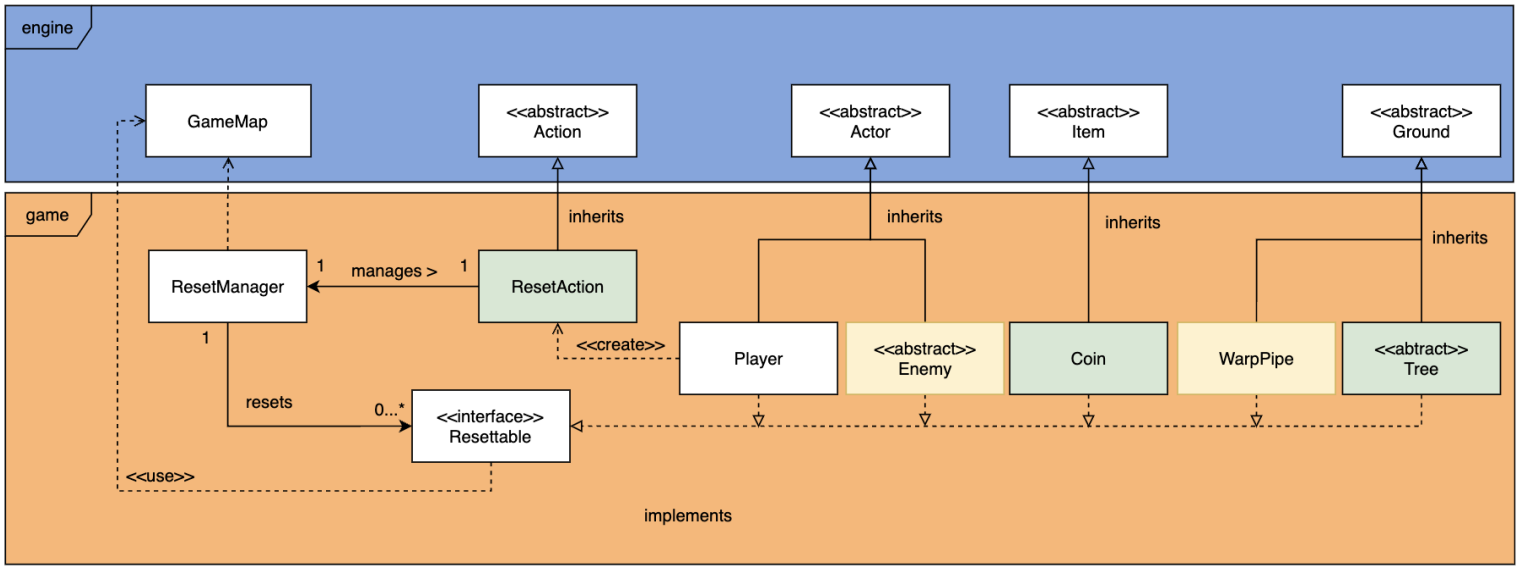
ResetAction class : a new class that extends *Action* that enables a *Player* to reset the game at any time. This will be done by adding this to the *ActionList* inside the *Player* class' *playTurn* method. This class will override the *execute* method to create (or obtain) the instance of *ResetManager*, and iterate over the *resettableList* and call *resetInstance* on each object inside this list. By creating this class the delegation of the actual action of resetting is held solely by this class, and since it extends *Action* it can be used inside the system as such.

Implementation : each of *Player, Goomba Koopa, Coin and Tree* will need to implement the *Resettable* interface and provide their own implementation for *resetInstance* to do the necessary tasks in order to complete the reset. These classes will also need to add themselves to the *resettableList* by calling the default method *registerInstance* in their constructor. Each class has been chosen to implement the interface and provide their own implementation so that each class is responsible for only the actions that the particular class is involved with, therefore the *ResetManager* doesn't have to know about any of these classes or how they work, just that they are implementations of *Resettable*, therefore conforming to the Single Responsibility Principle.

Assignment 3 Update:

See Enemies and Items section for the addition of the Enemy class.

WarpPipe was added to the list of classes that implement Resettable in order to perform the required actions when it is reset.



WBA

- All the team members in FIT2099 Team 6 will divide the work as equally as possible. Each of the members will be responsible for the reviewing and testing of each other's work.
- If a team member is struggling to complete their task, they are more than welcome to ask for help. However, they can't ask another team member to complete the task for them.
- Due dates for the deliverables will be discussed when allocating the tasks, if a team member is unable to complete their task by the due date, they must give adequate notice to allow the other team members to sort out the necessary added parts that they must do.

Delegated Tasks:

Samir Gupta: REQ 1 and REQ 2

Li Ting Desiree Sng: REQ 3, REQ 4, REQ 7 (with the assistance of Samir Gupta and Niloy Shehrin)

Niloy Shehrin: REQ 5 and REQ 6

- All of the team members will work on the class diagrams, interaction diagrams, and design rationale together during meetings based on each of the member's design documentations for the tasks that have been assigned.

Meetings: Thursday (7/4/2022), Sunday (10/4/2022)

Assignment 3 Delegated Tasks:

- Samir Gupta: REQ 1 (and help with REQ 2 & 3) and implementation for both Creative Mode features
- Li Ting Desiree Sng: REQ 2, 3 and some implementation for one Creative Mode feature
- Niloy Shehrin: Creative Mode tables

- All of the team members will work on the class diagrams, interaction diagrams, and design rationale together during meetings based on each of the member's design documentations for the tasks that have been assigned.
    - Meetings: Monday (16/5/2022), Sunday (22/5/2022)

Signatures

Samir Gupta 3/4/22

Li Ting Desiree Sng 3/4/22

Shehrin Niloy 3/4/22