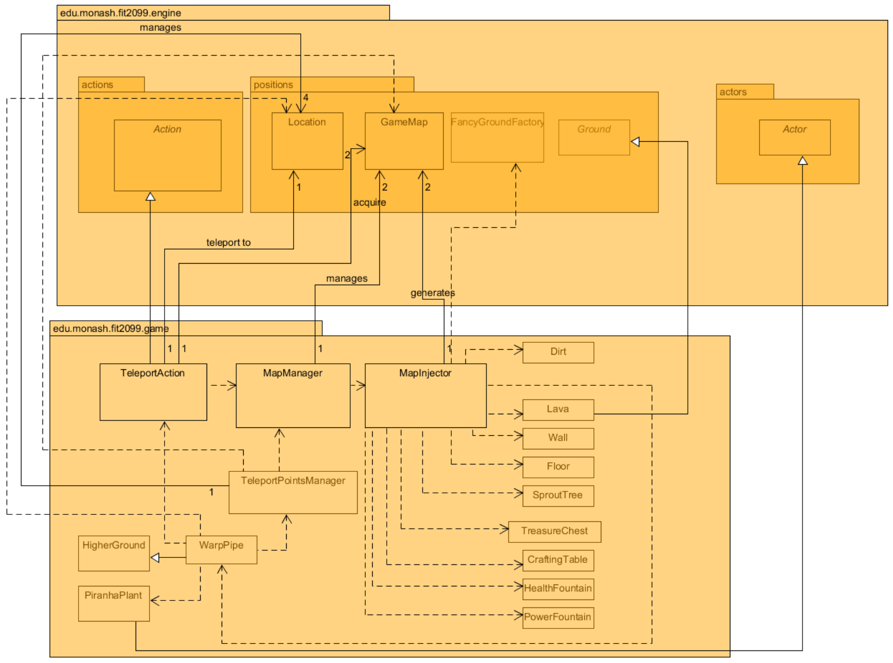
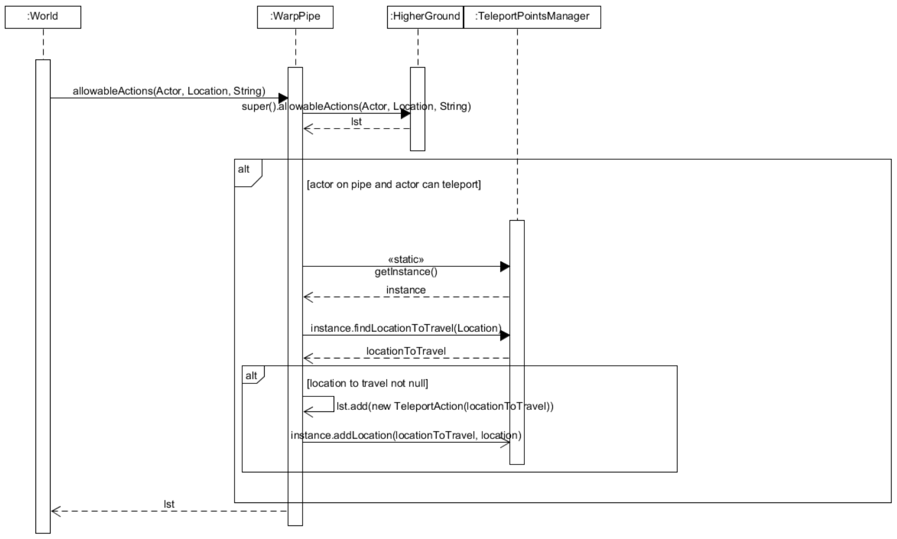
REQ 1: Lava Zone - Design Rationale

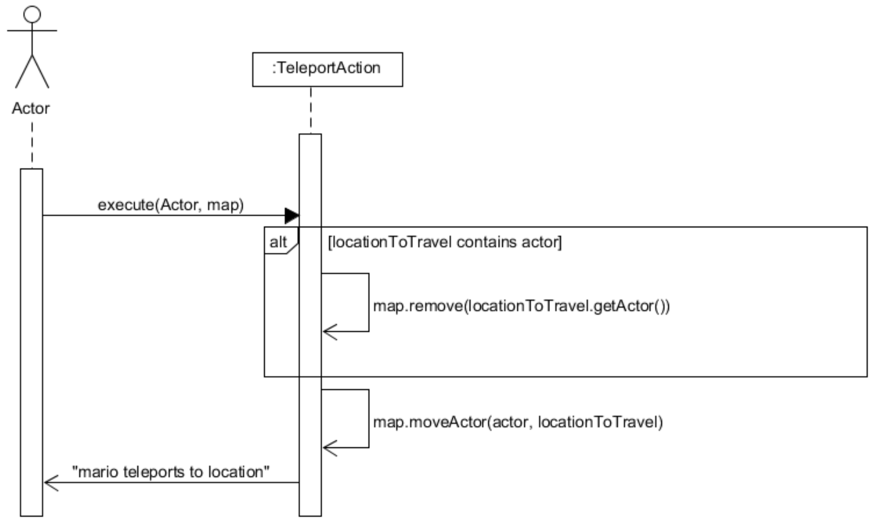
For reference, here are the class diagrams and sequence diagrams.

Class Diagram



Sequence Diagram


:World
:WarpPipe
:HigherGround
:TeleportPointsManager
allowableActions(Actor, Location, String)
super().allowableActions(Actor, Location, String)
lst
alt
[actor on pipe and actor can teleport]
«static»
getInstance()
instance
instance.findLocationToTravel(Location)
locationToTravel
alt
[location to travel not null]
lst.add(new TeleportAction(locationToTravel))
instance.addLocation(locationToTravel, location)
lst


Actor
:TeleportAction
execute(Actor, map)
alt
[locationToTravel contains actor]
map.remove(locationToTravel.getActor())
map.moveActor(actor, locationToTravel)
"mario teleports to location"

Rationale

I designed the REQ such that there exists a TeleportPointsManager and MapManager where both are singleton. This will ensure that every object in this file shares the manager and will be updated when one object updates it. This concept is very useful as maps and teleport points should be stored in one class so that classes such as Warp Pipe or Teleport Action can acquire it easily when required.

Essentially, I designed it so that WarpPipe is not required to have so many responsibilities to find where to teleport but instead, have a tp manager to let it know where to teleport to. The tp manager then passes a location to be teleported to and warp pipe then passes it to teleport action where it will move the actor to the correct location. This all follows the SRP.

TeleportAction Class

Purpose: Moves the actor to the new location when called
Single Responsibility Principle: Followed, since it only moves the actor
Open-Closed Principle: Followed, since we only extended from the Action abstract class and did not modify any of the source code
Liskov Substitution Principle: None
Interface Segregation Principle: None
Dependency Inversion Principle: None

TeleportPointsManager Class

Purpose: A global Singleton manager that stores and adds teleport points throughout the map
Single Responsibility Principle: Followed, since it only manages all the teleport points
Open-Closed Principle: None
Liskov Substitution Principle: None
Interface Segregation Principle: None
Dependency Inversion Principle: None

MapManager Class

Purpose: A global Singleton manager that manages all the map and map names respectively
Single Responsibility Principle: Followed, as it stores maps and map names and provides them when any class requires it
Open-Closed Principle: None
Liskov Substitution Principle: None
Interface Segregation Principle: None
Dependency Inversion Principle: None

MapInjector Class

Purpose: It reduces dependencies for the MapManager class and creates all the game maps, and their names respectively!
Single Responsibility Principle: Followed, it's only responsibility is to generate game maps, and their names as well.
Open-Closed Principle: None
Liskov Substitution Principle: None

Interface Segregation Principle: None
Dependency Inversion Principle: None

WarpPipe Class

Purpose: It will create and provide the teleport action as a possible action to the player when there is a location to teleport to
Single Responsibility Principle: Followed, it's only responsibility is to add teleport action as a possible action to player if there is a location to teleport to
Open-Closed Principle: Followed, it is extended from HigherGround class and does not modify any of the code from HigherGround class. In addition, if it overrides a method, it calls the super of it and adds new features to it, following the open for extension, close for modification principle.
Liskov Substitution Principle: None
Interface Segregation Principle: None
Dependency Inversion Principle: None

Lava Class

Purpose: New grounds for the player to interact with that deals damage when they step on it
Single Responsibility Principle: deals damage when the player steps on it
Open-Closed Principle: None
Liskov Substitution Principle: None
Interface Segregation Principle: None
Dependency Inversion Principle: None

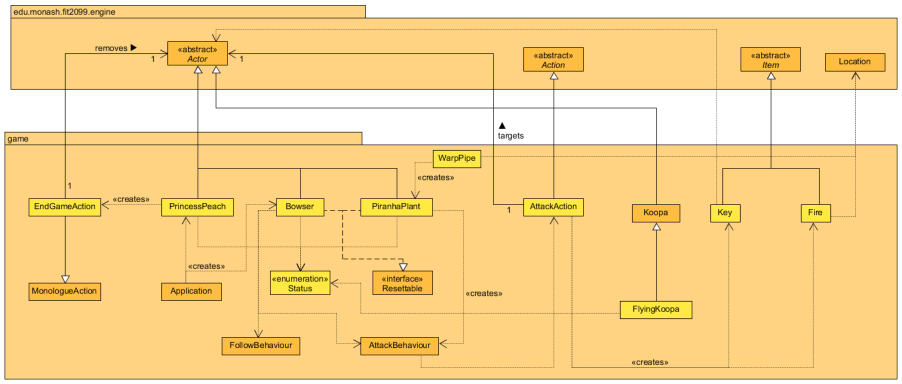
Changes from Assignment 2 to Assignment 3

For the map generation, instead of creating it in the Application class, it is created in the MapInjector class. This allows our Application to have lesser responsibilities. Furthermore, our maps are files instead of being in the class as arrays. It makes it easier to see as well. Our Maps can also be accessed easily through the MapManager depending on which map we would like to use by putting the respective map enum as the key.

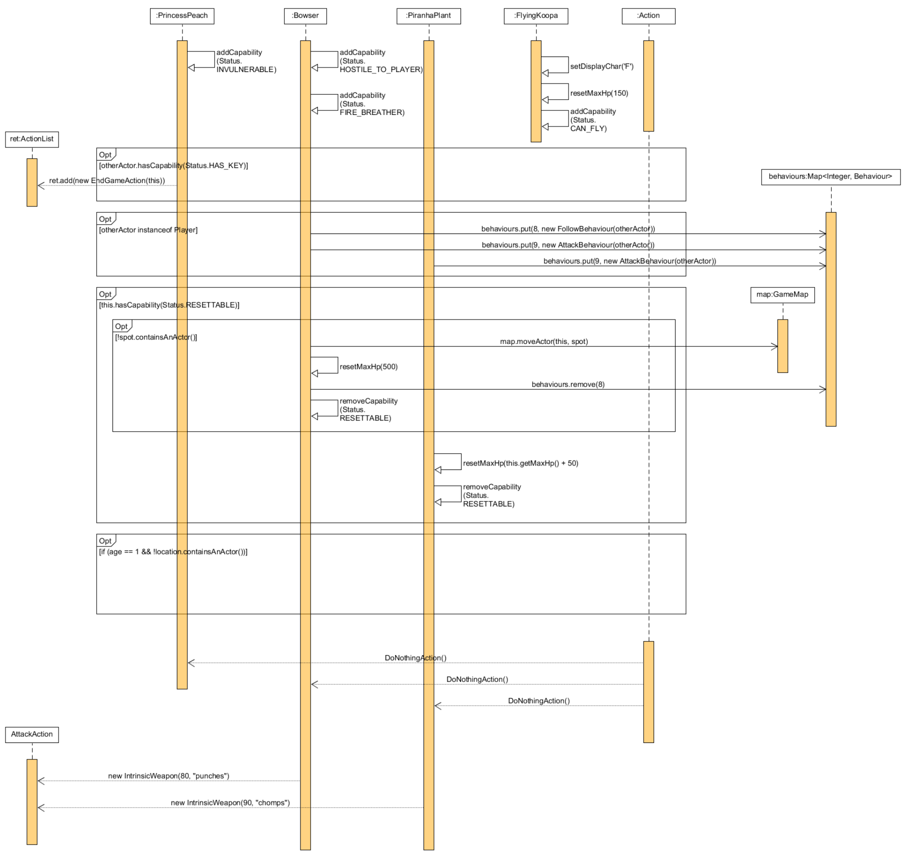
REQ 2: More allies and enemies! Design Rationale

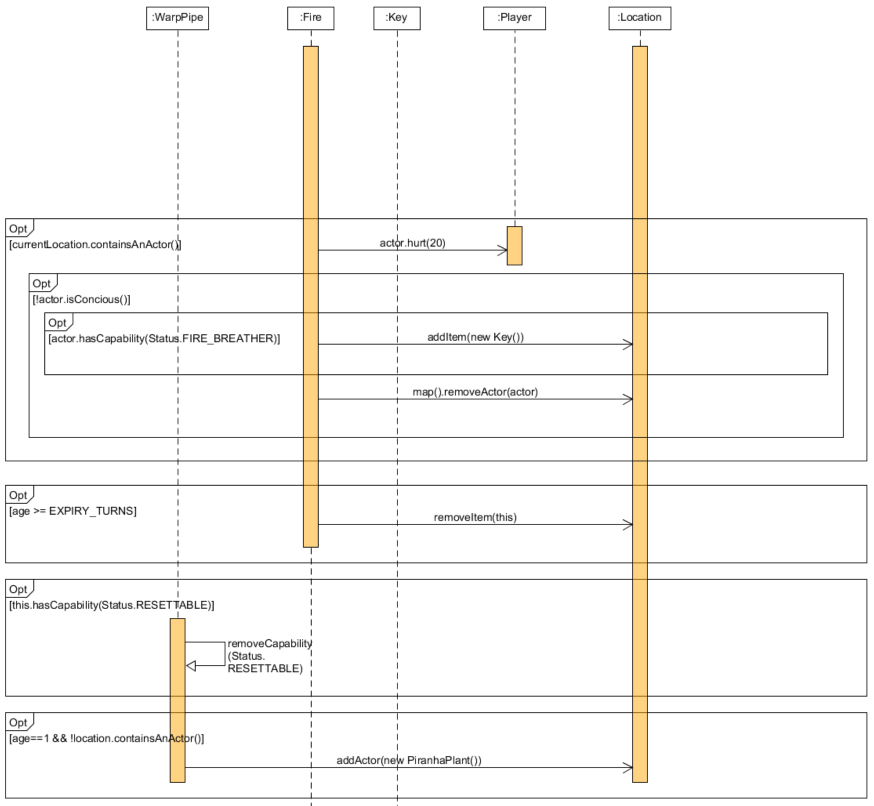
For reference here are the class and sequence diagrams.

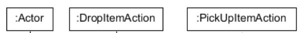
Class Diagram



Sequence Diagrams


:PrincessPeach
:Bowser
:PiranhaPlant
:FlyingKoopa
:Action
addCapability (Status. INVULNERABLE)
addCapability (Status. HOSTILE_TO_PLAYER)
addCapability (Status. FIRE_BREATHER)
setDisplayChar('F')
resetMaxHp(150)
addCapability (Status. CAN_FLY)
ret:ActionList
Opt
[otherActor.hasCapability(Status.HAS_KEY)]
ret.add(new EndGameAction(this))
behaviours:Map<Integer, Behaviour>
Opt
[otherActor instanceof Player]
behaviours.put(8, new FollowBehaviour(otherActor))
behaviours.put(9, new AttackBehaviour(otherActor))
behaviours.put(9, new AttackBehaviour(otherActor))
Opt
[this.hasCapability(Status.RESETTABLE)]
map:GameMap
Opt
[!spot.containsAnActor()]
map.moveActor(this, spot)
resetMaxHp(500)
behaviours.remove(8)
removeCapability (Status. RESETTABLE)
resetMaxHp(this.getMaxHp() + 50)
removeCapability (Status. RESETTABLE)
Opt
[if (age == 1 && !location.containsAnActor())]
DoNothingAction()
DoNothingAction()
DoNothingAction()
AttackAction
new IntrinsicWeapon(80, "punches")
new IntrinsicWeapon(90, "chomps")


:WarpPipe
:Fire
:Key
:Player
:Location
Opt
[currentLocation.containsAnActor()]
actor.hurt(20)
Opt
[!actor.isConcious()]
Opt
[actor.hasCapability(Status.FIRE_BREATHER)]
addItem(new Key())
map().removeActor(actor)
Opt
[age >= EXPIRY_TURNS]
removeItem(this)
Opt
[this.hasCapability(Status.RESETTABLE)]
removeCapability (Status. RESETTABLE)
Opt
[age==1 && !location.containsAnActor()]
addActor(new PiranhaPlant())



Rationale

Since Princess Peach cannot attack or be attacked, I decided to give her an 'INVULNERABLE' status which allows the program to check if the target has this status before attacking it. On her turn, since she cannot move, I had her always return a new DoNothingAction() using the playTurn method. As for my implementation of what happens when you interact with her, one of my approaches was to have the program loop through the Player's inventory and look for an instance of the Key, however this approach uses too much code, hence I went with my current approach, which was to have the program check the Player for a 'HAS_KEY' capability and end the game, this approach is a lot cleaner and simpler.

The game ends with the EndGameAction class called, where Princess Peach thanks the Player, a victory message appears, and the Player is removed from the map, which ultimately ends the game.

One of my approaches for implementing the fire dropped by Bowser is by having the fire damage the actor in AttackAction, however this would give the AttackAction class too much responsibility as it already handles all the attacks dealt by the Player and all the enemies in the game, thus violating the Single Responsibility Principle. My new approach was to let it have its own class to contain all the code, instantiate it in the AttackAction class, and damage the Player using the tick method. The method checks whether the current location of the fire contains an actor and if so, burns the actor. Other than that, the method also checks if Bowser is killed by his own fire to ensure that he drops the Key, allowing the Player to end the game. The method also keeps count of how many turns have passed to ensure that the fire only lasts for 3 turns.

The way I reset Bowser was by utilising the existing reset classes from REQ 7 in the previous assignment. Other than having the program check if Bowser had the 'RESETTABLE' status on his turn, I also had the program check if Bowser's original spawn location contained an actor which would most likely be himself if he didn't move. Then, I moved Bowser back to his original position, reset his HP and removed the FollowBehaviour from him to make sure he did not follow the Player even after reset unless the Player is within his attack range again.

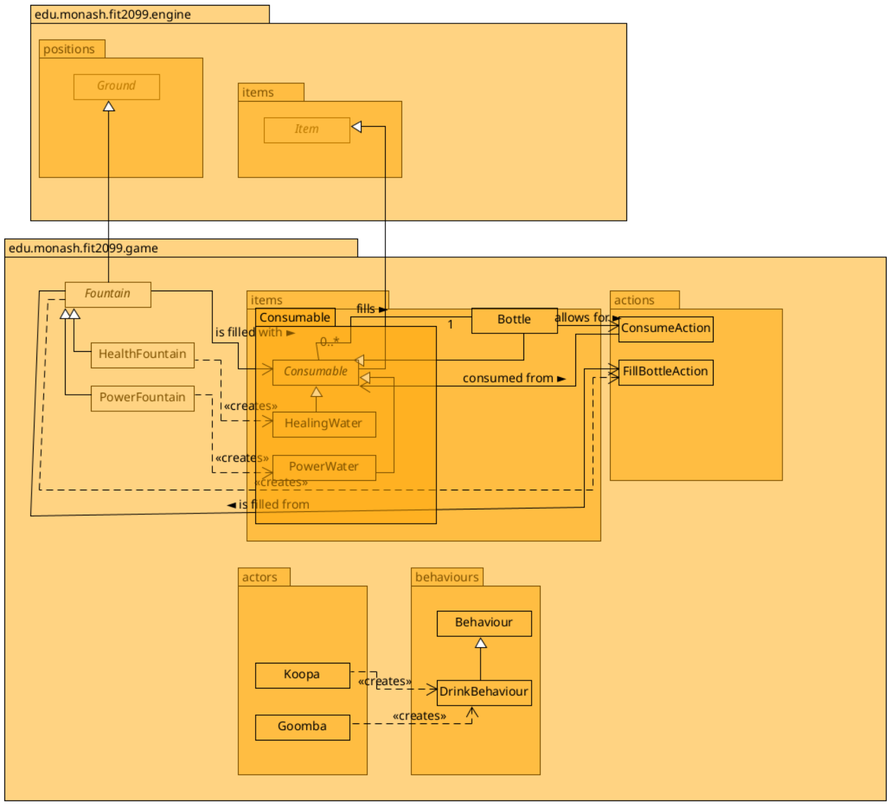
How I made Piranha Plant spawn was by using the tick method to keep track of the "age" (turns passed) of the Warp Pipe and spawn it on the second turn in the Warp Pipe class. The same method is also used to reset the age of the Warp Pipe to 0 if the Warp Pipe has the 'RESETTABLE' status and if there is no Pirahna Plant in it, to ensure that it respawns on the second turn again after reset. Otherwise, the Pirahna Plant gains 50 hit points which is done by once again checking if it has the 'RESETTABLE' status in its own class and resetting its hp to its own max hp plus 50, effectively healing it back to maximum health in the process as well.

To prevent violation of the Liskov Substitution Principle, I had the FlyingKoopa class I created extend the Koopa class, inheriting all of Koopa's attributes and methods and not overriding any of the methods. All the FlyingKoopa class contains is the changing of its display character, its max hp and its name. Other than that, I've added the capability 'CAN_FLY' to Flying Koopa to have the program allow for actors with this status to enter any higher grounds. I have also edited the MatureTree class to allow for a 50% chance of spawning either type of Koopa after a successful 15% spawn rate. To deal with when FlyingKoopa dies and to not violate the DRY principle by repeating code similar to the death of Koopa, I've changed a line in the AttackAction class that checks if the target's name is Koopa to if the target's name contains Koopa.

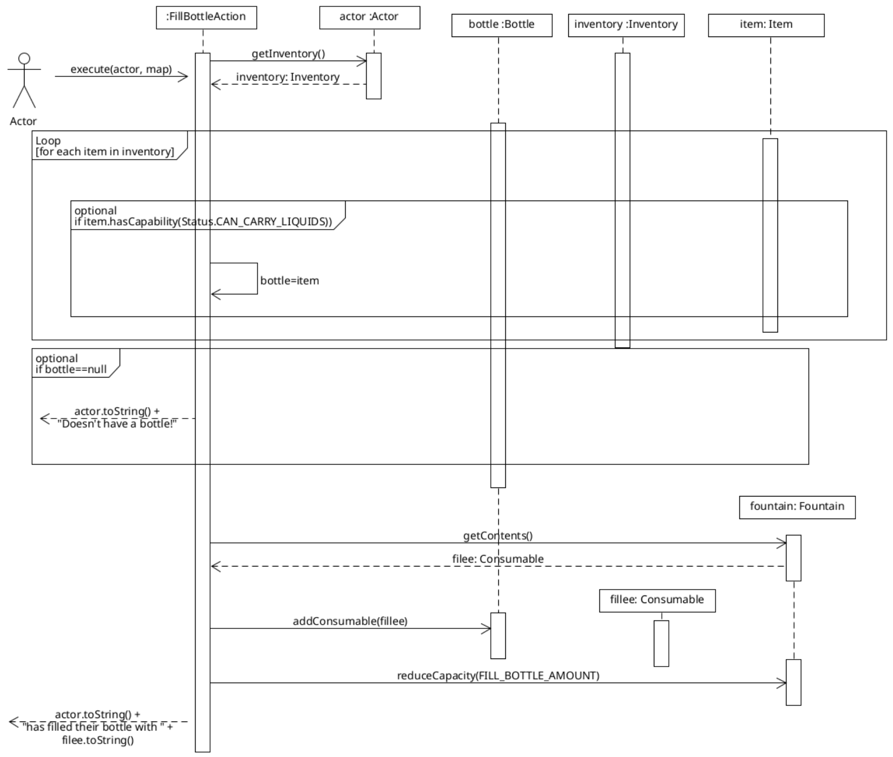
REQ 3: Fountains

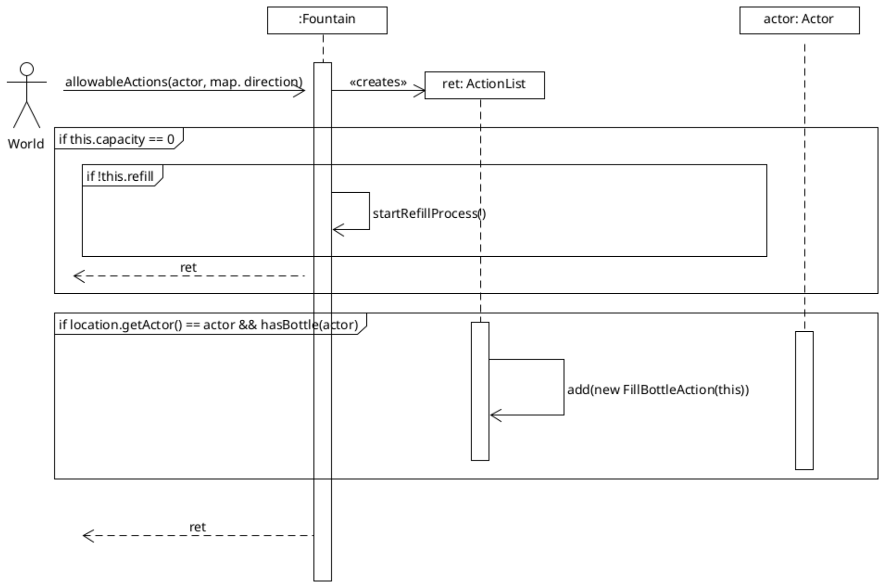
For reference, here are the class diagrams and sequence diagrams.

Class Diagram



Sequence Diagram




:Fountain
actor: Actor
World
allowableActions(actor, map. direction)
«creates»
ret: ActionList
if this.capacity == 0
if !this.refill
startRefillProcess()
ret
if location.getActor() == actor && hasBottle(actor)
add(new FillBottleAction(this))
ret

Rationale

I have also implemented the optional challenges, hence the Player does not start with a Bottle and instead Mario has to obtain the Bottle from Toad by talking to him (Select "Player talks with Toad").

Water classes

The `Consumable` class already existed and had similar properties as the new `Power Water` and `Healing Water` would have: They're both something you can "pick up", they have an effect when you "consume" them, they have similar menu descriptions for the action of consuming (so we can re-use `ConsumeAction` as well), etc.

It was then very simple to have the new Waters extend from the `Consumable` class with minimal changes to `Consumable` itself (Had to modify from Assignment 2, so that Consumables do not necessarily give the drinker an "effect"/capability anymore). With that change, it follows the open-clsoed principle where if I were to want to add another type of drink, just as `PoisonWater` that hurts the player, it would be very easy to add to the game by extending `Consumable` , and then it would work with my other classes like `Bottle` and `Fountain` s seamlessly without modifying existing classes.

A `Bottle` class (which is an Item that is place into the player's inventory) could then have a stack of `Consumable` s in it. Following the dependency inversion principle, it does not rely on the two specific Water classes, instead containing a stack of the lower-level class instead, so that if other types of water were to be added in the future, it would easily be extended just by creating a new class extending `Consumable` , without needing to touch other classes like `Bottle` , allowing for better maintainability. Drinking from a `Bottle` would then simply pop off the top of the stack of `Consumable` s it has, and call that consumable's `consume` function to do the appropriate action, no matter what type of `Consumable` it is.

One downside is that in theory, it would be possible for a programmer using this code to accidentally allow for putting food such as a into a Bottle, since the container is for a `Consumable` . An alternative approach would be to have a `Drinkable` abstract class implement `Consumable` , and then use that instead for the Waters, but the sacrifice of having a deeply nested inheritance tree (`Healing Water` extends `Drinkable` extends `Item` implements `Consumable` , etc.) deemed it not worth it.

Fountains

There was no existing class similar enough to `Fountains` as `Water` was to `Consumable` , and `HealthFountain` and `PowerFountain` had a lot of similar characteristics (both have a "capacity" and fill up your "bottle", both are types of "Ground" etc.). To follow the DRY (Don't Repeat Yourself) Principle, an abstract class was created (`Fountain`) f or the two fountains to extend.

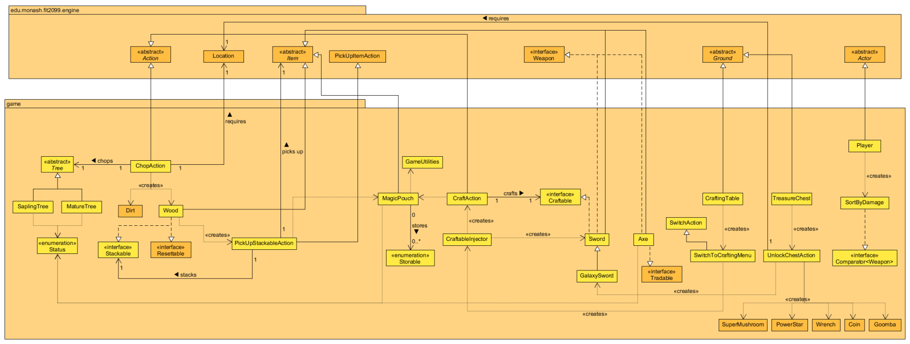
Drinking Behaviour

Drinking is added to certain enemies like Goombas and Koopas. If they happen to be on a fountain ground, they would have an 80% chance ot drinking it. Since the `ConsumeAction` and `consume` function depends on the abstraction of `Actor` instead of specifically relying on a `Player` class, it could easily substitute the `Actor` parent class with the child `Koopa` or `Goomba` class (Liskov's substitution principle) and have the same effect as if it were done on an Actor, for example, it would heal a Koopa or increase their base intrinsic damage, without having any special code for Koopas/Goombas.

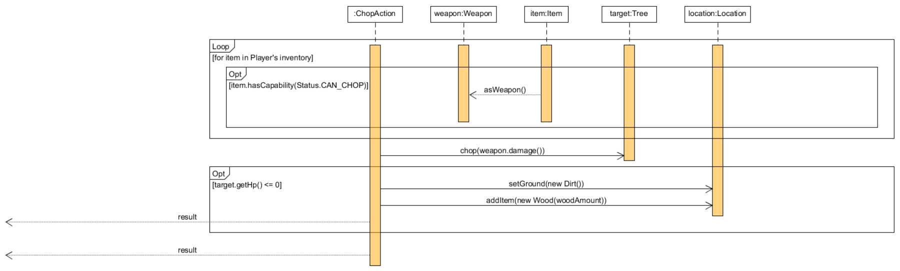
REQ 4: Minecraft Design Rationale

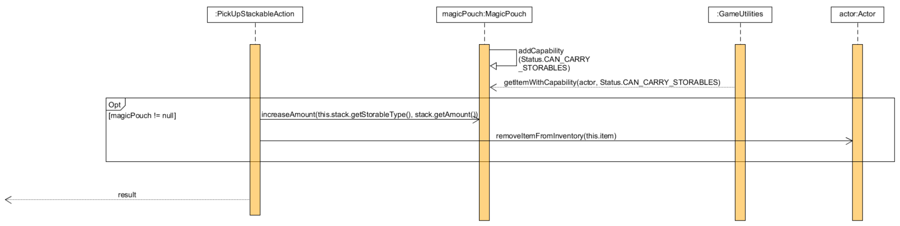
For reference here are the class and sequence diagrams.

Class Diagram

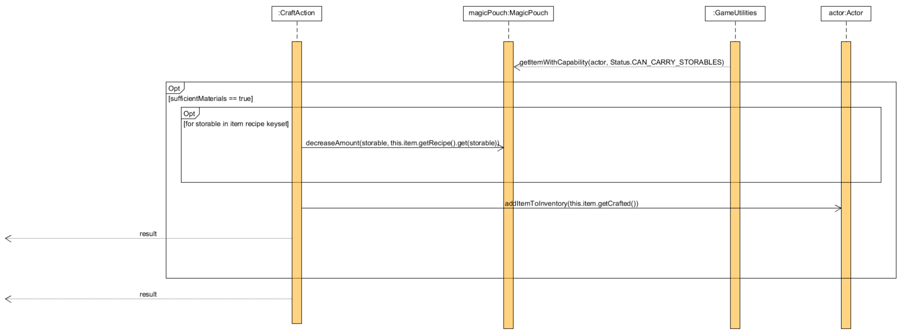


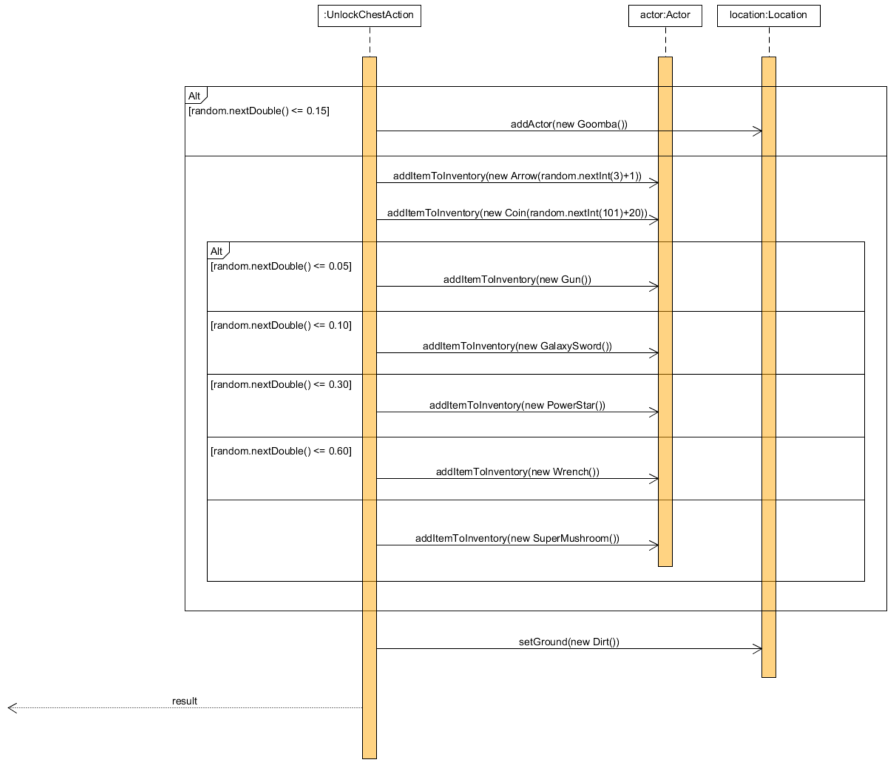
Sequence Diagrams

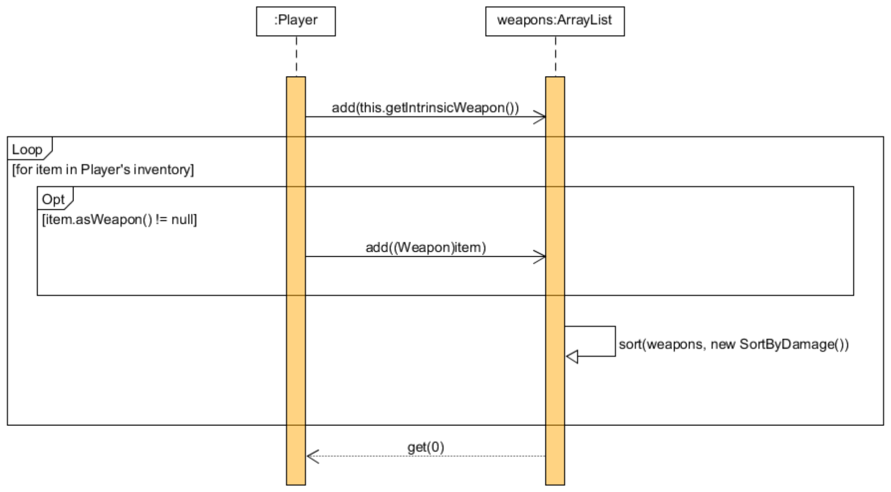




:PickUpStackableAction
magicPouch:MagicPouch
:GameUtilities
actor:Actor
addCapability (Status.CAN_CARRY _STORABLES)
getItemWithCapability(actor, Status.CAN_CARRY_STORABLES)
Opt
[magicPouch != null]
increaseAmount(this.stack.getStorableType(), stack.getAmount())
removeItemFromInventory(this.item)
result


:CraftAction
magicPouch:MagicPouch
:GameUtilities
actor:Actor
getItemWithCapability(actor, Status.CAN_CARRY_STORABLES)
Opt
[sufficientMaterials == true]
Opt
[for storable in item recipe keyset]
decreaseAmount(storable, this.item.getRecipe().get(storable))
addItemToInventory(this.item.getCrafted())
result
result


:UnlockChestAction
actor:Actor
location:Location
Alt
[random.nextDouble() <= 0.15]
addActor(new Goomba())
addItemToInventory(new Arrow(random.nextInt(3)+1))
addItemToInventory(new Coin(random.nextInt(101)+20))
Alt
[random.nextDouble() <= 0.05]
addItemToInventory(new Gun())
[random.nextDouble() <= 0.10]
addItemToInventory(new GalaxySword())
[random.nextDouble() <= 0.30]
addItemToInventory(new PowerStar())
[random.nextDouble() <= 0.60]
addItemToInventory(new Wrench())
addItemToInventory(new SuperMushroom())
setGround(new Dirt())
result


:Player
weapons:ArrayList
add(this.getIntrinsicWeapon())
Loop
[for item in Player's inventory]
Opt
[item.asWeapon() != null]
add((Weapon)item)
sort(weapons, new SortByDamage())
get(0)

Rationale

Chopping

To implement chopping trees into the game, we've allowed trees to have health points. Since we've designed it so that Sprouts do not drop Wood while Saplings and Mature Trees do, We've made it so that Saplings and Mature Trees have the capability "CAN_BE_CHOPPED" to allow the program to check if the tree can be chopped before applying ChopAction(a new class) on it.

ChopAction is the class used to actually chop the trees. The class utilises the execute method which initialises the Player's punch as the main weapon for chopping, then it checks the Player's inventory for an Axe, which is the only item in the game that has the new capability "CAN_CHOP", if it exists, then the Player's main chopping tool switches to the Axe. The method then chops the tree, proceeding to check if the tree's health points are below or equal to 0, and if so, will set the ground to Dirt and drop Wood.

Wood

As for the implementation of Wood, it was noticed that it was conceptually very similar to Coins. Both can be spawned by Trees, both are "Items" you pick up that increases a "counter" (how much coins/wood you have), both are "spent", either by selling or by crafting. In the previous assignment, the Coin system wasn't abstracted out, so we took this as an opportunity to make an abstraction, the `Stackable` interface, so that it would be easy to make similar items in the future in the future. (e.g: REQ5 actually uses `Stackable` again for `Arrow` s.).

Similarly, `PickUpCoinAction` needed to be refactored to a more generic

`PickUpStackableAction` to allow for any stackable to be picked up. This was done via having a parameter `Stackable` in the constructor of the action, and because of the Liskov Substitution Principle, one can create this action and pass in any Stackable like a Coin or a Wood item. Executing this action will then remove add to this stack as needed and remove it from the ground if it was picked up from the ground.

Because of the different types of Stackables to be carried with similar logic, I took inspiration from the `Bottle` class in REQ3, but for Stackables instead of Water. Like Bottles, it is an item placed into an Actor's inventory, and with it, they can "pick up" stackable items and it'll be added to the counter. In this way, any actor with this "MagicPouch" can have a counter of stackables in their inventory, and not just for `Player` .

The downside of this is that in order to use the `MagicPouch` , you will have to iterate over the inventory of an actor until you find the magic pouch (checked by seeing if the item has the capability of "storing stackables").

One way around this is to have a `MagicPouchManager` class (which would either be a Singleton or purely static class since there would need to be only one instance of it in the game) with a hashmap key being the `Actor` , and the value being the `MagicPouch` . Picking up a stackable would then use this class to see if the actor picking it up has a magic pouch, and if it does, to add it in. The reason why this approach didn't go was because of wanting to reduce the amount of global variables as possible, which a singleton would be similar to. This would also need really fit the conceptual idea of a "Magic Pouch" being an item that you can pick up or drop.

Crafting

As for the implementation of crafting, we've created a CraftingTable ground that is placed next to Toad that can be used through the CraftAction class. To avoid violating the DRY principle, all the craftable items implement the Craftable interface which contains methods used to get the recipes for crafting said items, get the crafted items and their names.

The CraftAction class utilises the aforementioned Magic Pouch and checks if the Player has sufficient materials to craft the item from the recipe. If so, the method will craft the item and add it to the Player's inventory, otherwise it will display a message saying the Player's does not have enough materials.

As the menu will be more congested with options for the player to choose, we have decided that a separate menu for the crafting would be useful and easier for the user to choose the options.

Chests

The TreasureChest class contains a Treasure Chest ground that can be "unlocked" by the Player to gain surprises. It utilises the UnlockChestAction class where it takes a random generated probability and adds the actor or item that is within the range of the generated probability to the location of the Treasure Chest, the Treasure Chest is that set to Dirt.

Weapons

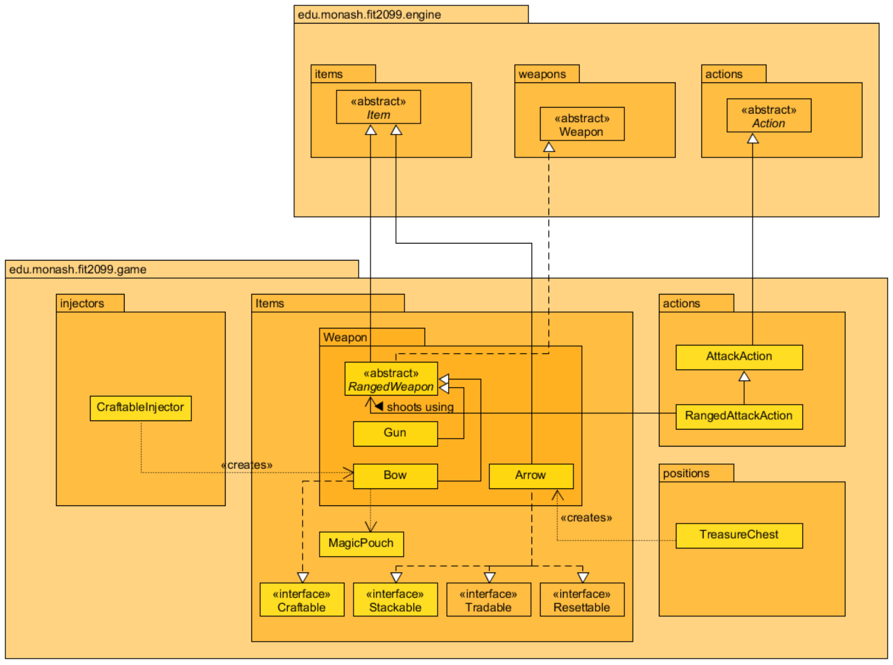
For the GalaxySword class, to avoid violating the DRY principle, I've made the class extend the Sword class, the only differences being the change in display character, returned damage and its name.

Since we've added a few weapons (including the ones in creative REQ5) into the game, we've also changed the way the program access the Player's held weapon using the getWeapon method. What we've done is first override the getWeapon method in the Player class, create an ArrayList to store all the weapons possessed by the Player including his Intrinsic Weapon (punch), then utilising a Comparator to sort the ArrayList by weapon damage in decreasing order, and finally choosing the weapon at the top of the ArrayList, which would ultimately be the weapon with the highest damage.

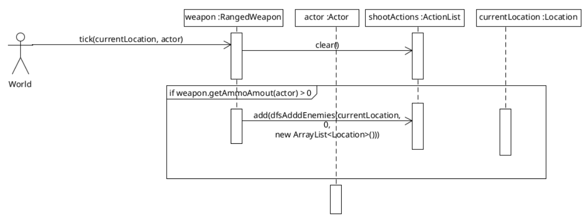
REQ 5: Ranged Weapons

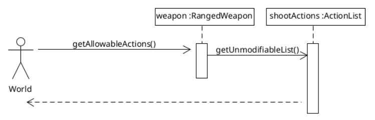
For reference, here are the class diagrams and sequence diagrams.

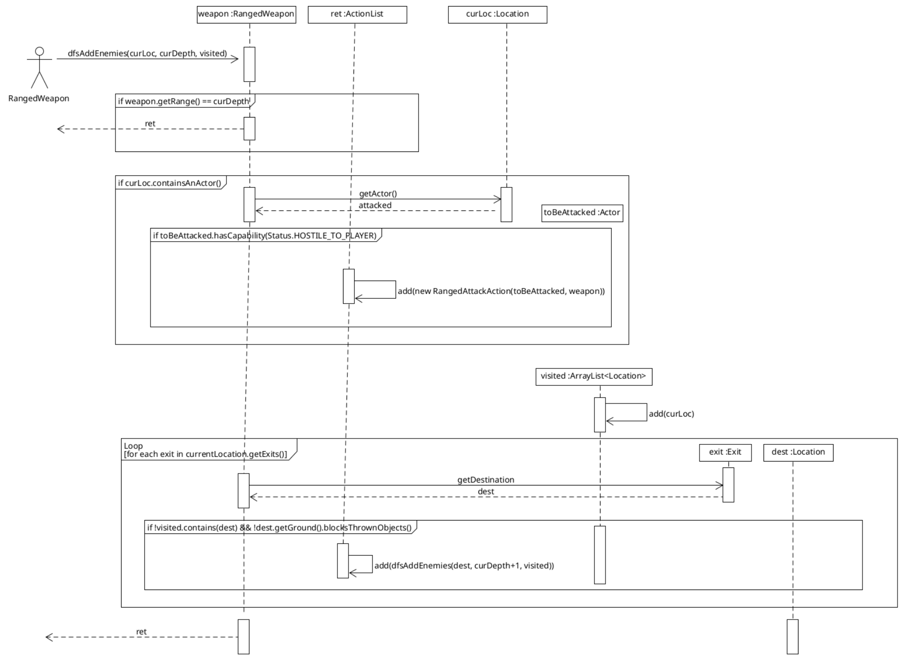
Class Diagram



Sequence Diagram


weapon :RangedWeapon
actor :Actor
shootActions :ActionList
currentLocation :Location
World
tick(currentLocation, actor)
clear()
if weapon.getAmmoAmout(actor) > 0
add(dfsAdddEnemies(currentLocation,
0,
new ArrayList<Location>()))




weapon :RangedWeapon
ret :ActionList
curLoc :Location
RangedWeapon
dfsAddEnemies(curLoc, curDepth, visited)
if weapon.getRange() == curDepth
ret
if curLoc.containsAnActor()
getActor()
attacked
toBeAttacked :Actor
if toBeAttacked.hasCapability(Status.HOSTILE_TO_PLAYER)
add(new RangedAttackAction(toBeAttacked, weapon))
visited :ArrayList<Location>
add(curLoc)
Loop
[for each exit in currentLocation.getExits()]
exit :Exit
dest :Location
getDestination
dest
if !visited.contains(dest) && !dest.getGround().blocksThrownObjects()
add(dfsAddEnemies(dest, curDepth+1, visited))
ret

Rationale

A new abstract `RangedWeapon` class was created that extends `Item` and implements `Weapon`, and all ranged weapons would then extend from this `RangedWeapon` class.

The reason for this is that ranged weapons have a common functionality between them, that is, searching for enemies within `WEAPON_RANGE` tiles to shoot. The code for this is somewhat involved, using a Depth-First-Search to find enemies within its range that it can reach (meaning no walls or something that can block items) and by "hiding" this code (information hiding for unnecessary things is one of the goals of good software design) inside the parent class as a private method, it reduces the complexity of the code since other code does not need to concern itself with it. This DFS is the main reason for the class to exist, and so it follows the Single-Responsibility Principle, where the only reason to touch this class would be to modify this algorithm.

Other goals of good software design includes performance and simplicity. Although having a recursive depth-first-search algorithm may be slightly more complex than something like checking all the tiles in the map and checking if it's within a certain distance from the player, this would have had the performance impact of a) Checking every tile on the map instead of just the ones you could reach and nothing is blocking the ammo and b) Calculating the distance for every tile. So the trade-off is considered worth it.

Classes like `Bow` and `Gun` extend this class, and don't need to worry about the code for "finding" enemies within range, they just have to provide things like the range of the weapon, the name, etc.

Following the Open-Closed Principle, "classes should be open to extension but closed to modification", `RangedAttackAction` extends `AttackAction` , a previous type of `Action` that we used for regular (melee) attacking. Here, we wanted all the functionality of `AttackAction` , but wanted to extend it so that it could see enemies from further away as well.

Arrows are similar to `Wood` and `Coin` stacks. We can have many of those arrows, and just need a counter of how many arrows we need, and don't specifically need specific references to those Arrow objects like `Bottle` s need to with their Power or Healing Water. As such, we can simply make Arrows implement `Stackable` and re-use the existing code that allows for Stackables to pick up arrows and increment the counter for that stackable, following the "DRY" (Don't repeat yourself) principle.

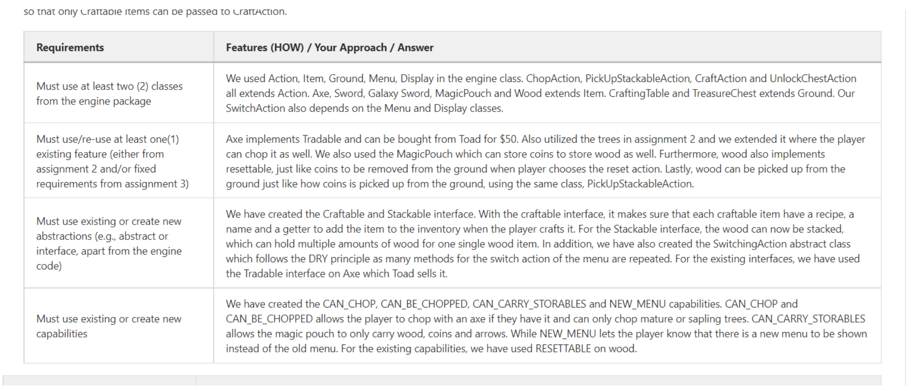
Requirement 4

Title: Minecraft

Description: Mario can now punch trees and collect wood just like Minecraft! With wood, mario can also craft tools to defend himself such as a sword from the crafting table! On top of that, there are treasures around the map which drop good loot when you unlock it such as a galaxy sword! With the menu piling up with many options, the crafting menu comes in handy to provide easier focus on what the player would like to do.

Explanation why it adheres to SOLID principles (WHY):

- All classes follows the SRP and will only have one sole responsibility. For instance, the chopping of a tree, the player may have an axe(to chop the tree), a chopping action, and a PickUpStackableAction when the player wants to pick up the wood.
- The crafting of an item also follows the SRP where there is a crafting table(which the player can craft items at), and a craft action which then the item is added to the inventory!
- Open Closed Principle is also followed in the Switching of Menus. All classes which extended from the SwitchingAction does not modify the base code but instead just extends to it. For instance, the switching action can be switched to either shop or crafting menu.
- Interface Segregation Principle is followed. There are items that can be bought and items that is stackable. However, there are some items that only can be bought and not stackable, such as Axe. Thus, we split the 2 interfaces.
- Lastly, DIP is also followed. The CraftAction only needs to know the Craftable methods and not all the item's methods. Therefore, the interface Craftable was created to follow this principle so that only Craftable items can be passed to CraftAction.



so that only Craftable items can be passed to CraftAction.

Requirements	Features (HOW) / Your Approach / Answer
Must use at least two (2) classes from the engine package	We used Action, Item, Ground, Menu, Display in the engine class. ChopAction, PickUpStackableAction, CraftAction and UnlockChestAction all extends Action. Axe, Sword, Galaxy Sword, MagicPouch and Wood extends Item. CraftingTable and TreasureChest extends Ground. Our SwitchAction also depends on the Menu and Display classes.
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	Axe implements Tradable and can be bought from Toad for $50. Also utilized the trees in assignment 2 and we extended it where the player can chop it as well. We also used the MagicPouch which can store coins to store wood as well. Furthermore, wood also implements resettable, just like coins to be removed from the ground when player chooses the reset action. Lastly, wood can be picked up from the ground just like how coins is picked up from the ground, using the same class, PickUpStackableAction.
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	We have created the Craftable and Stackable interface. With the craftable interface, it makes sure that each craftable item have a recipe, a name and a getter to add the item to the inventory when the player crafts it. For the Stackable interface, the wood can now be stacked, which can hold multiple amounts of wood for one single wood item. In addition, we have also created the SwitchingAction abstract class which follows the DRY principle as many methods for the switch action of the menu are repeated. For the existing interfaces, we have used the Tradable interface on Axe which Toad sells it.
Must use existing or create new capabilities	We have created the CAN_CHOP, CAN_BE_CHOPPED, CAN_CARRY_STORABLES and NEW_MENU capabilities. CAN_CHOP and CAN_BE_CHOPPED allows the player to chop with an axe if they have it and can only chop mature or sapling trees. CAN_CARRY_STORABLES allows the magic pouch to only carry wood, coins and arrows. While NEW_MENU lets the player know that there is a new menu to be shown instead of the old menu. For the existing capabilities, we have used RESETTABLE on wood.

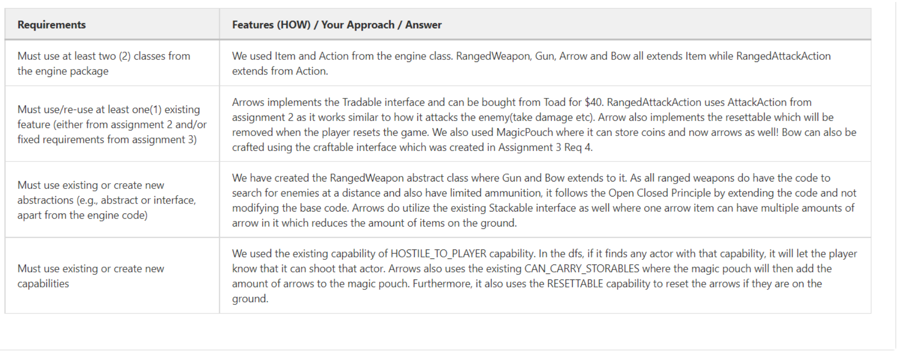
Requirement 5

Title: Ranged Combat

Description: Bows, arrows and guns now exist in the game! They utilize dfs to search through the enemies and finish them! Bows have a range of 5 and have arrows to shoot while guns have unlimited POWER(unlimited range and unlimited bullets)!! With these weapons, we're confident mario can one shot Bowser(who needs Peach when we got guns).

Explanation why it adheres to SOLID principles (WHY):

- All classes follow SRP. When the player wants to shoot an enemy, it must have a bow or gun and calls the RangedAttackAction which attacks the enemy. All classes mentioned have their single responsibility which meets the principle.
- Open Closed Principle is followed as well, where Bow and Gun extends to RangedWeapon without modifying the source code and only adds new features.

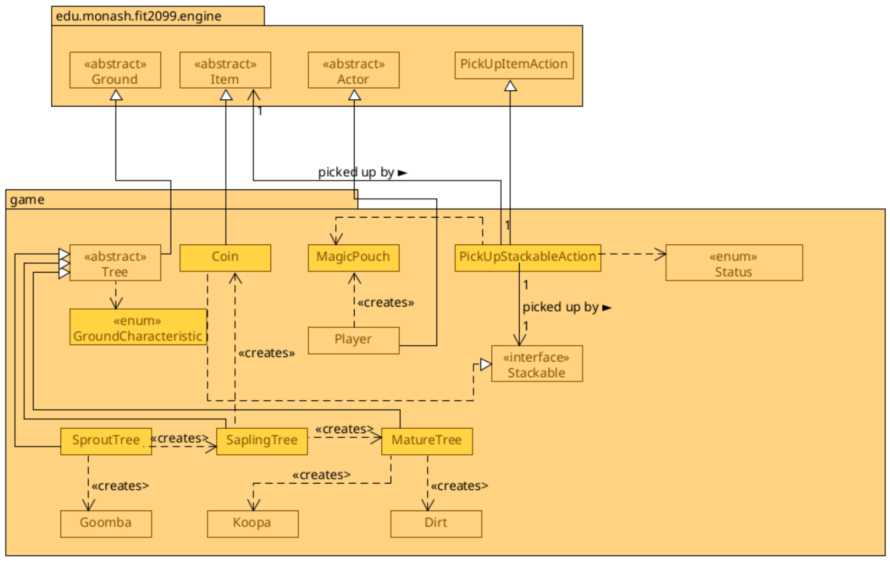


Requirements	**Features (HOW) / Your Approach / Answer**
Must use at least two (2) classes from the engine package	We used Item and Action from the engine class. RangedWeapon, Gun, Arrow and Bow all extends Item while RangedAttackAction extends from Action.
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	Arrows implements the Tradable interface and can be bought from Toad for $40. RangedAttackAction uses AttackAction from assignment 2 as it works similar to how it attacks the enemy(take damage etc). Arrow also implements the resettable which will be removed when the player resets the game. We also used MagicPouch where it can store coins and now arrows as well! Bow can also be crafted using the craftable interface which was created in Assignment 3 Req 4.
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	We have created the RangedWeapon abstract class where Gun and Bow extends to it. As all ranged weapons do have the code to search for enemies at a distance and also have limited ammunition, it follows the Open Closed Principle by extending the code and not modifying the base code. Arrows do utilize the existing Stackable interface as well where one arrow item can have multiple amounts of arrow in it which reduces the amount of items on the ground.
Must use existing or create new capabilities	We used the existing capability of HOSTILE_TO_PLAYER capability. In the dfs, if it finds any actor with that capability, it will let the player know that it can shoot that actor. Arrows also uses the existing CAN_CARRY_STORABLES where the magic pouch will then add the amount of arrows to the magic pouch. Furthermore, it also uses the RESETTABLE capability to reset the arrows if they are on the ground.

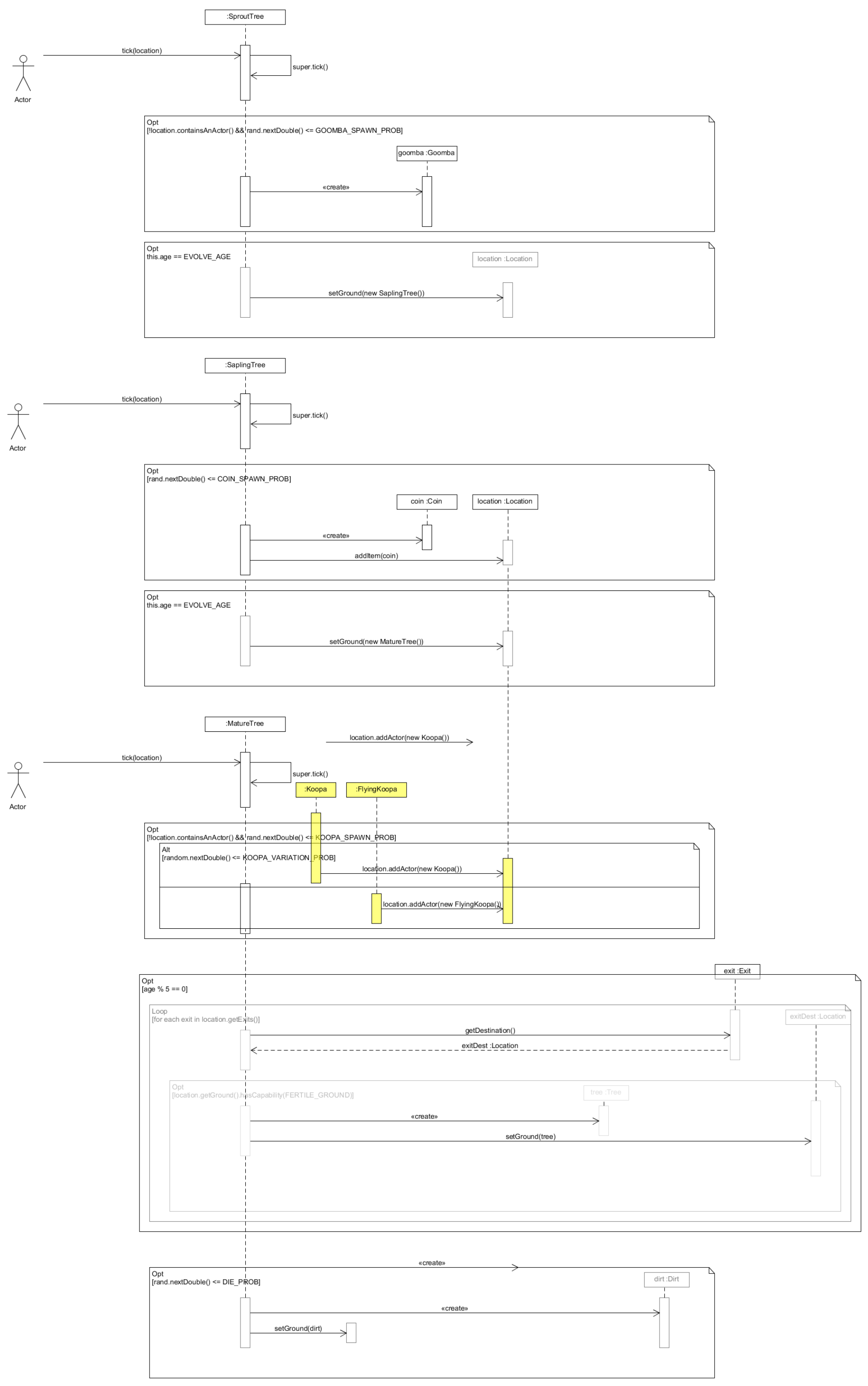
REQ1:Let it grow! 🌳 Design Rationale

For reference, here are the class diagrams and the sequence diagrams.

Class Diagram



Sequence Diagram (Tree's tick method)


:SproutTree
tick(location)
super.tick()
Actor
Opt
[!location.containsAnActor() && rand.nextDouble() <= GOOMBA_SPAWN_PROB]
goomba :Goomba
«create»
Opt
this.age == EVOLVE_AGE
location :Location
setGround(new SaplingTree())
:SaplingTree
tick(location)
super.tick()
Actor
Opt
[rand.nextDouble() <= COIN_SPAWN_PROB]
coin :Coin
location :Location
«create»
addItem(coin)
Opt
this.age == EVOLVE_AGE
setGround(new MatureTree())
:MatureTree
location.addActor(new Koopa())
tick(location)
super.tick()
Actor
:Koopa
:FlyingKoopa
Opt
[!location.containsAnActor() && rand.nextDouble() <= KOOPA_SPAWN_PROB]
Alt
[random.nextDouble() <= KOOPA_VARIATION_PROB]
location.addActor(new Koopa())
location.addActor(new FlyingKoopa())
exit :Exit
Opt
[age % 5 == 0]
Loop
[for each exit in location.getExits()]
exitDest :Location
getDestination()
exitDest :Location
Opt
[location.getGround().hasCapability(FERTILE_GROUND)]
tree :Tree
«create»
setGround(tree)
«create»
Opt
[rand.nextDouble() <= DIE_PROB]
dirt :Dirt
«create»
setGround(dirt)

Note that the `<<creates>>` relations aren't necessarily needed in the diagram, as stated on edstem that since it isn't the main focus and simply creates it with `new Goomba()` , but I added it in for clarity.

Also note that Goomba and Koopa extends Actors, and Dirt extends Ground. I omitted that because that portion wasn't relevant to explanation of this design.

Rationale

WalletKeeper is in REQ1 because REQ1 is the place where you pick up coins to earn money. The Trading REQ5 expands on this by using this interface for buying/selling items.

A Coin extends an Item, like in the real world, it is a physical "thing" that we can *see* and *pick up*. In the engine, this helps us be able to re-use logic to display the item, and pick it up, without having to worry about the implementation details of that. Picking it up is handled by `PickUpStackableAction` , which is explained in Ass 3 REQ 4 section.

GroundCharacteristic

The reason for a GroundCharacteristic enum is to check for fertile grounds on the surrounding. In the Dirt class (not pictured), the constructor would add the capability of it being FERTILE. This way, it would be easy to extend in the future, if we have other types of ground besides dirt that we want to be fertile enough to grow trees on.

Changes from Assignment 1 to Assignment 2

Design of Tree

It was decided to go with the alternative approach with having separate classes, each extending the "Tree" class, for the different stages of the tree. The drawbacks are outweighted by the simplicity this new design has by leveraging a Single Responsibility Principle, with each class being able to concentrate on its own logic for that specific stage of the tree. For example, now although there are more classes, there are now less dependencies on a single class. Before, the Tree class needed to have a dependency on `Koopa` , `Goomba` , and `Coin` because it needed to spawn them. Now, the dependency can be kept on the specific stage that is required, for example only the `SproutTree` class needs to have a dependency on `Goomba` . The sequence diagrams for the tick() functions are a lot shorter and simpler now.

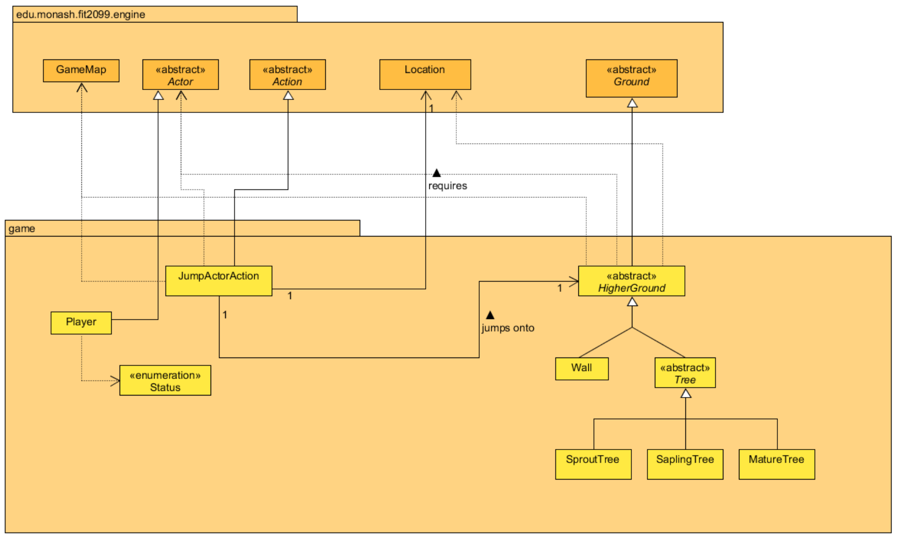
Changes from Assignment 2 to Assignment 3

`PickUpCoinAction` was replaced by `PickUpStackableAction` , which is explained about in Ass 3's REQ 4 section.

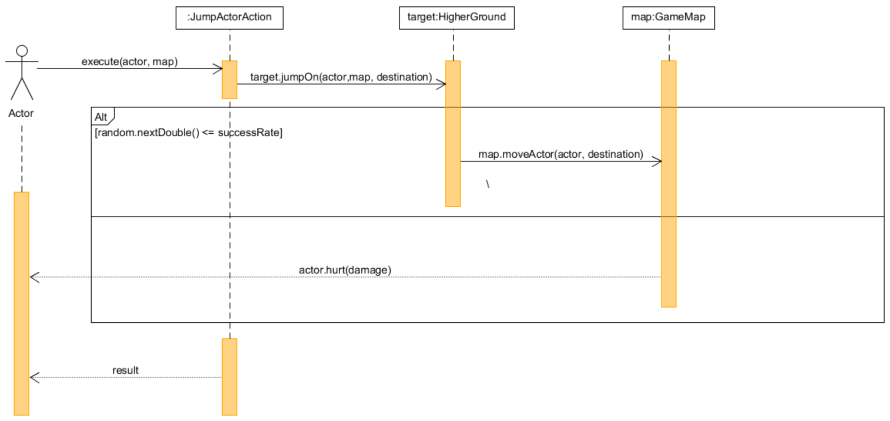
REQ 2: Jump Up, Super Star! Design Rationale

For reference, here are the class diagrams and sequence diagrams.

Class Diagram



Sequence Diagram


:JumpActorAction
target:HigherGround
map:GameMap
Actor
execute(actor, map)
target.jumpOn(actor,map, destination)
Alt
[random.nextDouble() <= successRate]
map.moveActor(actor, destination)
actor.hurt(damage)
result

Rationale

My previous approach for this was to have the program check the Player's surroundings for high grounds and give the Player to option to jump to those high grounds. However, that approach resulted in a lot of messy and repeating code which would have heavily violated the DRY principle.

Thus, my new approach for this was to instead have the classes Tree and Wall extend an abstract class I created called HigherGround so the program could identify both as one single higher ground entity which would negate the repetition of code. The rest of the approach is similar to my previous one wherein when the Player chooses to jump to the high ground in the menu, the program will generate a random probability and compare it to the success rate of jumping to that specific high ground. If the generated probability is within the range of the success rate, the Player will successfully jump onto the high ground, otherwise the Player will fail to jump onto the high ground and take damage instead. All the comparisons and results occur in one method that is contained inside the HigherGround class called jumpOn which is called in the JumpActorAction class.

The JumpActorAction class is a class used to allow the Player to jump onto high grounds. It extends the Action abstract class so that it can override the methods inside to allow the Player to perform a jump.

The JumpActorAction class should have 2 dependencies:

1. GameMap, which is used the jumpOn method to move the Player to the specified high ground.
2. Actor, which contains many methods that the JumpActorAction class can use to get and modify information about the Player.

The Location class is used by the jumpOn method to move the Player to the high ground if the generated probability is within the range of the success rate.

The HigherGround class is a base class extended by the Tree and Wall classes to allow the program to identify both classes as one single higher ground entity. It contains the jumpOn method which moves the Player to the specified high ground if the generated probability is within the range of the success rate of jumping to the specified high ground. If the Player has the status TALL which is granted if the Player consumes a Super Mushroom, the Player will have 100% success rate of jumping and no fall damage.

The Player is given a status called CAN_JUMP that is used by the Tree and Wall classes allow actors to jump onto them depending on whether the actors have the CAN_JUMP status.

Changes from Assignment 1 to Assignment 2

None.

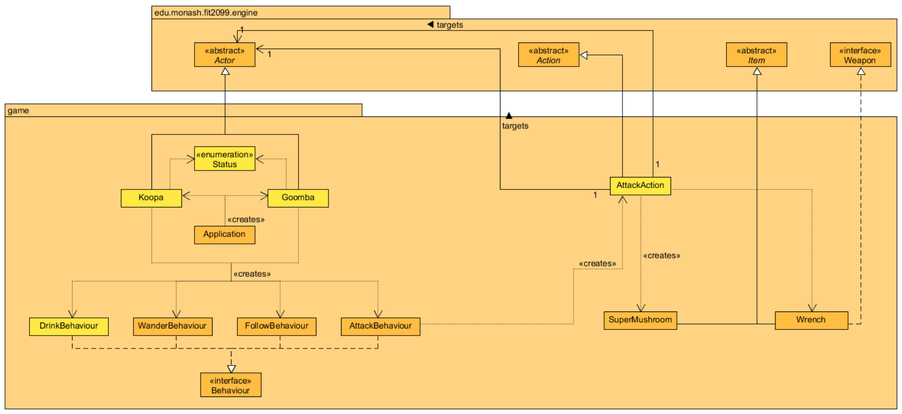
Changes from Assignment 2 to Assignment 3

None.

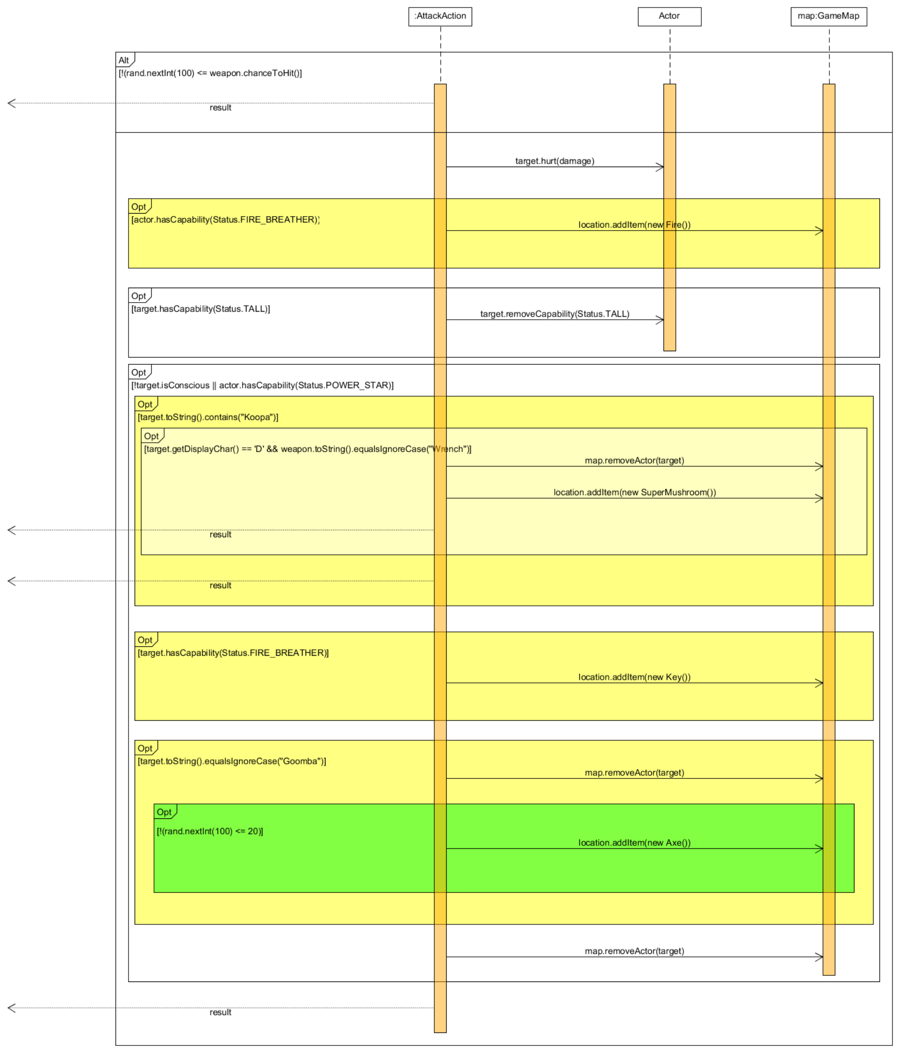
REQ 3: Enemies Design Rationale

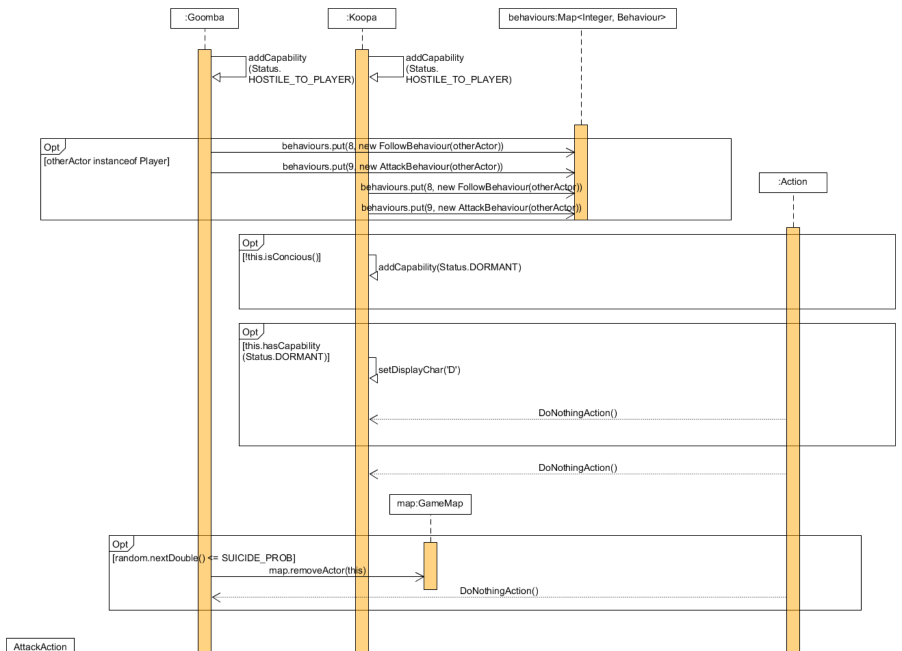
For reference, here are the class diagrams and sequence diagrams.

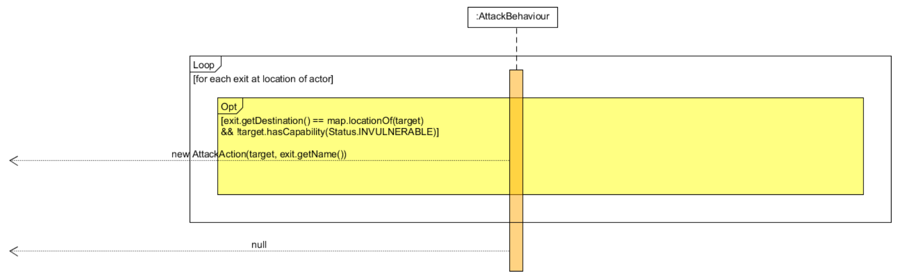
Class Diagram



Sequence Diagram


:AttackAction
Actor
map:GameMap
Alt
[!(rand.nextInt(100) <= weapon.chanceToHit()]
result
target.hurt(damage)
Opt
[actor.hasCapability(Status.FIRE_BREATHER)]
location.addItem(new Fire())
Opt
[target.hasCapability(Status.TALL)]
target.removeCapability(Status.TALL)
Opt
[!target.isConscious || actor.hasCapability(Status.POWER_STAR)]
Opt
[target.toString().contains("Koopa")]
Opt
[target.getDisplayChar() == 'D' && weapon.toString().equalsIgnoreCase("Wrench")]
map.removeActor(target)
location.addItem(new SuperMushroom())
result
result
Opt
[target.hasCapability(Status.FIRE_BREATHER)]
location.addItem(new Key())
Opt
[target.toString().equalsIgnoreCase("Goomba")]
map.removeActor(target)
Opt
[!(rand.nextInt(100) <= 20)]
location.addItem(new Axe())
map.removeActor(target)
result


:Goomba
:Koopa
behaviours:Map<Integer, Behaviour>
addCapability (Status. HOSTILE_TO_PLAYER)
addCapability (Status. HOSTILE_TO_PLAYER)
Opt
[otherActor instanceof Player]
behaviours.put(8, new FollowBehaviour(otherActor))
behaviours.put(9, new AttackBehaviour(otherActor))
behaviours.put(8, new FollowBehaviour(otherActor))
behaviours.put(9, new AttackBehaviour(otherActor))
:Action
Opt
[!this.isConcious()]
addCapability(Status.DORMANT)
Opt
[this.hasCapability (Status.DORMANT)]
setDisplayChar('D')
DoNothingAction()
DoNothingAction()
map:GameMap
Opt
[random.nextDouble() <= SUICIDE_PROB]
map.removeActor(this)
DoNothingAction()
AttackAction


:AttackBehaviour
Loop
[for each exit at location of actor]
Opt
[exit.getDestination() == map.locationOf(target)
&& !target.hasCapability(Status.INVULNERABLE)]
new AttackAction(target, exit.getName())
null

Rationale

My approach for this was to create a new status in the Status enumeration class and add it as a capability to both Koopa and Goomba to have them attack the Player similar to the status HOSTILE_TO_ENEMY used by the Player. Previously, I had the Koopa and Goomba utilise the PunchAction and KickAction classes respectively which are instantiated in the AttackBehaviour class to attack the Player. However, this violates the DRY principle, hence I scrapped both and instead utilised the existing AttackAction class instead. I overrided the getIntrinsicWeapons() method in both the Goomba and Koopa classes to give them their own unique attack values. The AttackAction class is instantiated in the AttackBehaviour class now, replacing the previously used KickAction and PunchAction classes.

The Koopa and Goomba are enemies that appear in the game that can attack the Player. The Koopa and Goomba classes instantiate the following classes:

1. WanderBehaviour: It is used to allow the enemies to wander around the map.
2. FollowBehaviour: It is used to make the enemies follow the Player if the Player is standing next to them.
3. AttackBehaviour: It is used to make the enemies automatically attack the Player if the Player is standing next to them.

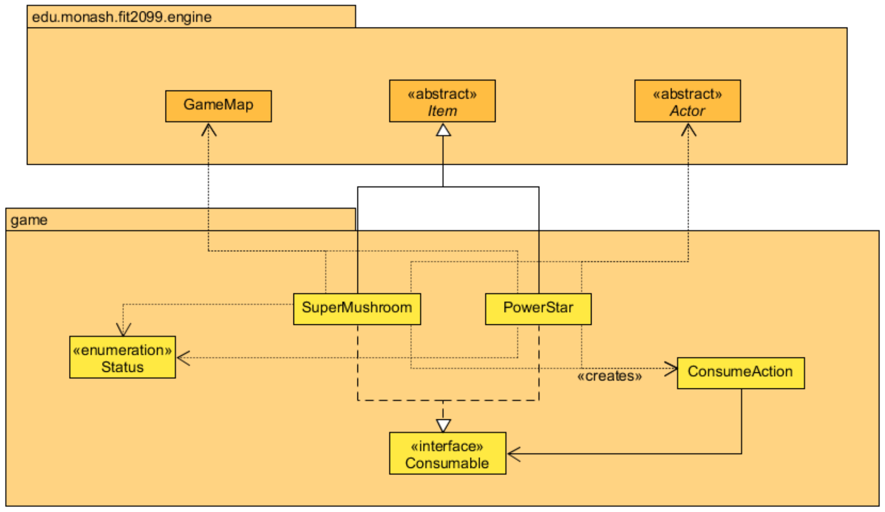
The AttackBehaviour class instantiates an AttackAction class used by enemies to attack the Player.

The AttackAction class which is used by the Player to attack enemies will drop create a new SuperMushroom object after the Player breaks a Koopa shell with a Wrench.

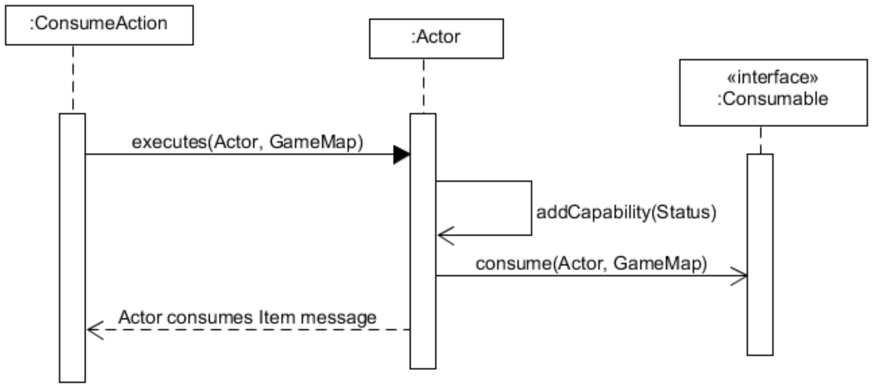
REQ4: Magical Items 🍄 Design Rationale

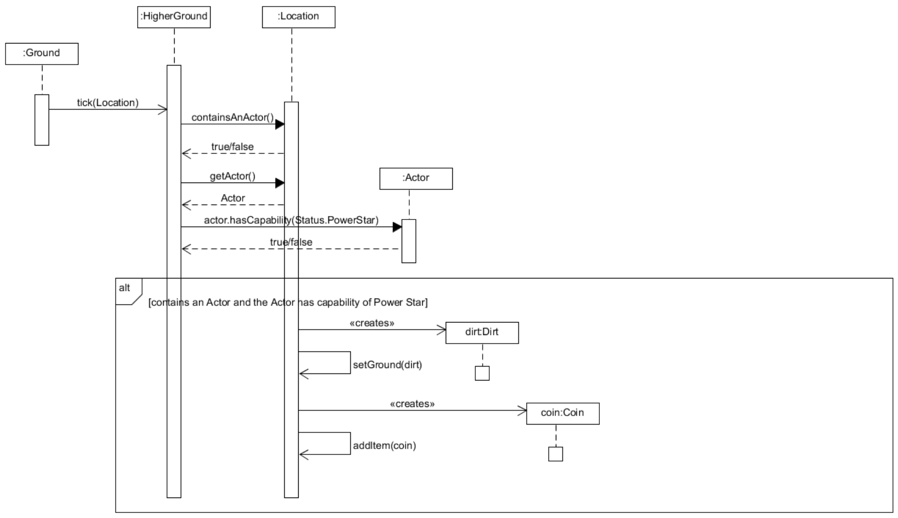
For reference, here are the class diagrams and the sequence diagrams.

Class Diagram



Sequence Diagram





Rationale

The Consumable interface will have one method that need to be implemented.

2. consume(Actor, GameMap) - This will execute when the item is eaten. The reason for having this is that we want to allow for custom code to be run depending on the class implementing it. For example, PowerStar not only inflicts a status (perhaps called a `POWER_STAR` status, or perhaps an INVINCIBLE status), but also heals the player. A SuperMushroom doesn't only inflict a status (The TALL status given in the code), but also lets you increase the *max* HP, which is different from increasing just the *HP* like what PowerStar does.

A ConsumeAction was created as well to let the player eat the item. It has an association with the Consumable that it is meant to eat.

This association with the interface follows the dependency inversion principle. If the ConsumeAction class were to have associations instead with PowerStar and SuperMushroom, if any other consumable were to be created, the code in ConsumeAction would have to be updated as well, which is not ideal because we want to minimising changing code from other classes when making such a change.

An alternate design would be to have a ConsumableItem class, that itself derives from Item. Then, PowerStar and SuperMushroom would have to derive from this ConsumableItem class. In fact, using this method, we can have the effect be a protected attribute of this class instead of a method. The reason we didn't go with this approach is because

1. It would limit the Consumables to only be Items. If we were to decide

to have a consumable Actor in the future (Eating dead turtles and mushrooms?), we would have to re-design the code to account for this.
2. Not have deeply nested inheritance trees, e.g: A SuperMushroom extending a Consumable extending an Item has 3 levels deep, and can go deeper if you want a `SuperDuperMushroom` extending a SuperMushroom.

Changes from Assignment 1 to Assignment 2

None.

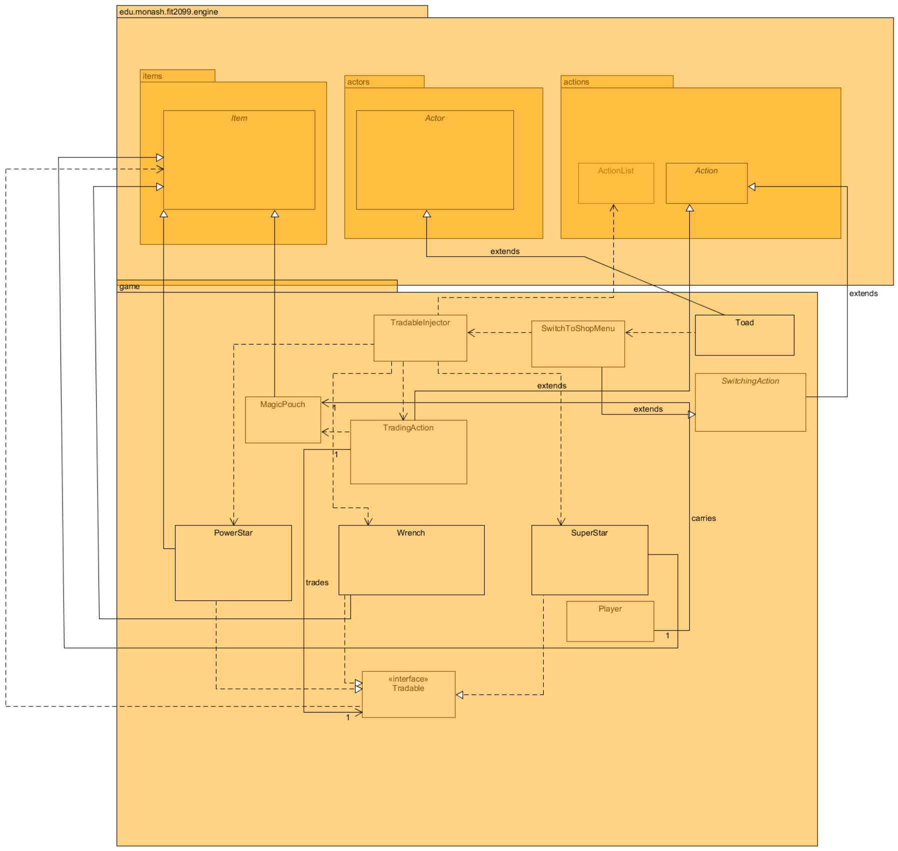
Changes from Assignment 2 to Assignment 3

- Consumables no longer enforce an effect to be placed on the actor. This is because of new consumables added to the game that don't force an effect on the actor.

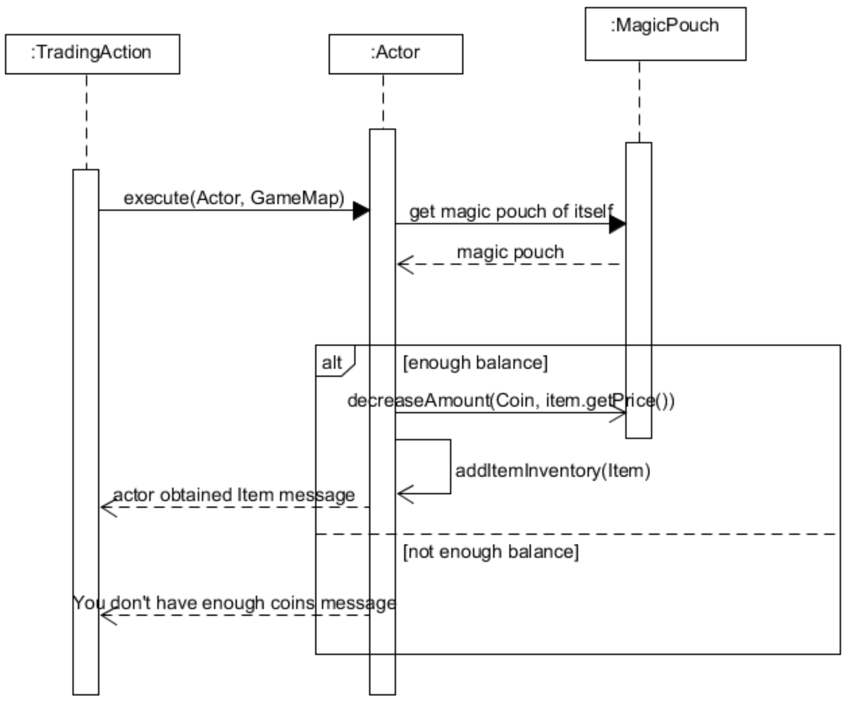
REQ 5: Trading Design Rationale

For reference, here are the class diagrams and sequence diagrams.

Class Diagram



Sequence Diagram



Rationale

When designing the code, I have decided that Toad will have the code to perform trading with the Player instead of the Player having the code as it will have too many responsibilities. By creating a dependency injector for the Toad class called TradableInjector, I managed to reduce the dependencies on Toad. I have also created a Tradable interface as to follow the Dependency Inversion Principle so that the TradingAction class is only required to know the methods in the Tradable interface and not every single method in the Item class. Then, all the items which can be tradable with the TradingAction will be added to Toad such that Toad do not need to know that all those Actions are TradingActions but simply Actions.

Regarding the buying of items in this code, I have designed a Wallet system which is an interface called WalletKeeper. This allows me to deduct the player's coins preventing the need to use instanceOf for the player and simply deduct using the WalletKeeper's method.

Since all the items which can be traded are of the same code, I followed the principle Don't Repeat Yourself(DRY) and only created one Trading class where it can be used by all the Tradable items. For this code, I also made it such that the Player can keep buying the same item and they will be added to the inventory.

TradingAction Class

Purpose: An action to be performed when the player decides to trade with Toad
Single Responsibility Principle: It's sole responsibility is to provide the buying option to the player if they decide to choose to buy the item from Toad and perform the execution of it by checking and deducting the coins from the player!
Open-Closed Principle: This is followed since it will be easy to just add more methods to it, for example: adding a discount to certain items
Liskov Substitution Principle: None
Interface Segregation Principle: None Dependency Inversion Principle: I created the Tradable interface so to follow this principle. The main reason behind it is that, instead of this class having to depend on a full Item instance where it does not utilize the entire methods of it. It can instead be dependent on an Interface which only requires the certain methods of the item. For example, getting the price of the item!

Toad Class

Purpose: To perform trading with the Player
Single Responsibility Principle: Perform trading with player and is not required to know what items to trade or what actions to be added to its allowableActions
Open-Closed Principle: This is followed as we extended from the Actor class without modifying the source code
Liskov Substitution Principle: None
Interface Segregation Principle: No interface created
Dependency Inversion Principle: None

TradableInjector Class

Purpose: It's a Dependency Injector class to instantiate tradable items and actions for the Toad class which also reduces its dependencies to Toad Single Responsibility Principle: Creating new TradingAction instances for each Tradable item
Open-Closed Principle: It certainly can be added more TradingAction instances for new Tradable items in the future and to be passed to Toad.
Liskov Substitution Principle: None, it cannot extend from any class due to its different usability
Interface Segregation Principle: None
Dependency Inversion Principle: None

Changes from Assignment 1 to Assignment 2

I decided that a Tradable interface and TradingInjector class was required after Week 8. The Tradable interface follows DIP which allows the TradingAction class to only use those methods in the Tradable. This is good design since it will prevent TradingAction class to access from unnecessary methods in the Item instance.

For the TradingInjector class, I learned from Week 8 that it is good to lift off dependencies from Toad. As more Tradable items appear in the game, it is only wise to create these Item objects in another class instead of Toad and be dependent on that class instead. Therefore, I have created a Dependency Injector class to be used for Toad class.

Changes from Assignment 2 to Assignment 3

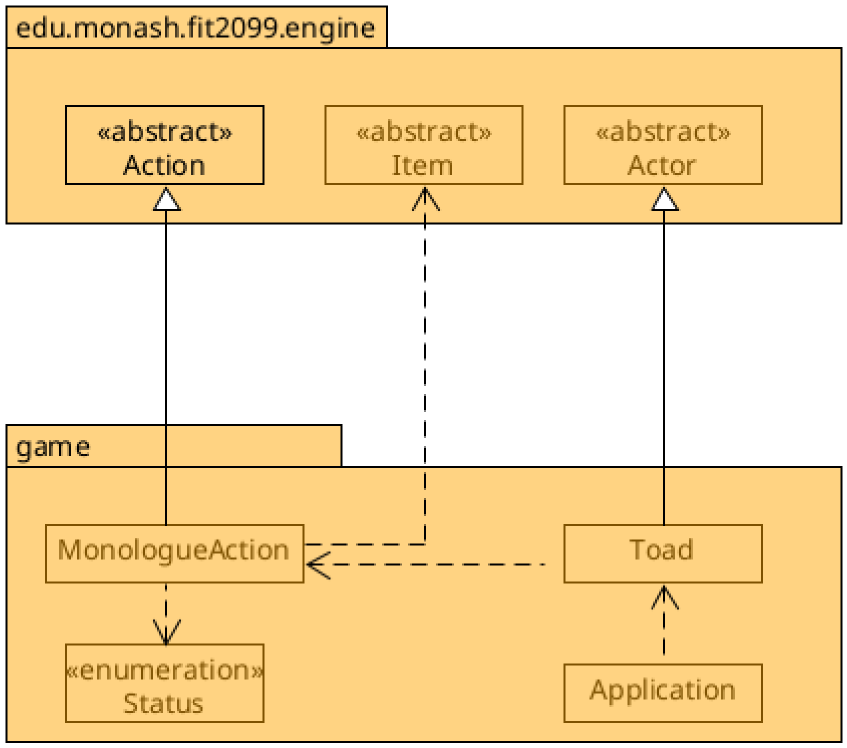
Instead of the WalletKeeper interface, we have now created a MagicPouch which holds many storable items such as coins. By doing this, we do not need to implement the keeping of balance in player, which follows the SRP principle!

In addition, we do also have the SwitchToShopMenu which makes it easier for the player to see which action it would like to perform.

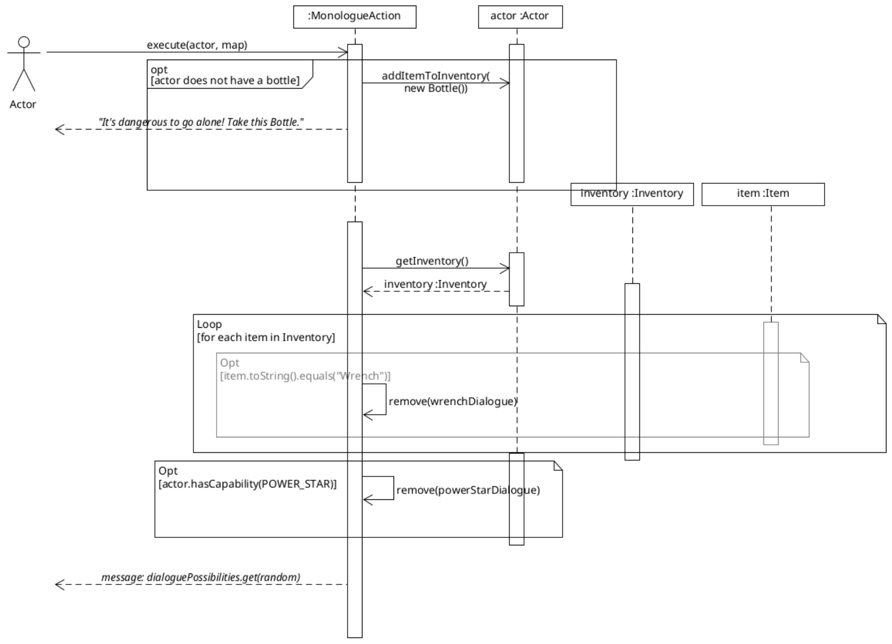
REQ6: Monologue 💬 Design Rationale

For reference, here are the class diagrams and the sequence diagrams.

Class Diagram



Sequence Diagram


:MonologueAction
actor :Actor
Actor
execute(actor, map)
opt
[actor does not have a bottle]
addItemToInventory(
new Bottle())
"It's dangerous to go alone! Take this Bottle."
inventory :Inventory
item :Item
getInventory()
inventory :Inventory
Loop
[for each item in Inventory]
Opt
[item.toString().equals("Wrench")]
remove(wrenchDialogue)
Opt
[actor.hasCapability(POWER_STAR)]
remove(powerStarDialogue)
message: dialoguePossibilities.get(random)

Rationale

A Toad is an interactable entity that has a physical manifestation on the game map. It extends the Actor class so that it can re-use logic and methods elsewhere that uses the subclass Actor; For example, the engine's World class, which uses a list of the subclass Actor to continuously update/print out the entity's logic per game tick. which follows the DRY Don't repeat yourself principle, as otherwise we would have to re-implement functionality like setting up a character for how the Toad should look like, its menu description for talking to the Toad, etc.

One downside of this is that not everything in the Actor base class is applicable to Toad. For example, the Toad doesn't do any fighting at all, and rather, acts as a static NPC. As such, some methods/attributes from the subclass Actor like the health, hitPoints, heal methods and others aren't applicable to Toad.

The dialogue options are currently hard-coded into MonologueAction, which admittedly isn't ideal because it would be nice to have a design where different instances of Toad (or other talkable actors) would have different dialogues and conditions for certain dialogue to appear. For example, perhaps having two separate Toads where one only talks about Enemies, and another only about Enemies.

An alternative design would have MonologueAction not hard code the dialogue, and instead have an "addDialogue" method to add a line of dialogue. In this way, in the Toad class we can set up up the action to talk about specific things, depending on what it was created for. The problem with this approach is that we need to set certain conditions for certain lines of dialogue to appear, and that is difficult to set up when adding the line of dialogue.

For example, if we were to only want to add a line of dialogue when the actor doesn't have a specific status, we can make the method `addDialogue(dialogueString, status)` . So for talking about invincibility for example, `addDialogue("Eat a power star to become invincible!", Status.POWER_STAR)` . But what about if we want a condition in that an actor needs to not have an item in an inventory? Create a new `addDialogue(dialogueString, item)` method? What about more advanced logic, such as "You need item X but not item Y and also have status Z for this status to appear", or if we want to add other conditions not reliant on inventory and status, such as if we want to add a condition based on the Actor's location, we would need to have another `addDialogue` method. This is the reason that it is currently coded in MonologueAction.

Changes from Assignment 1 to Assignment 2

None.

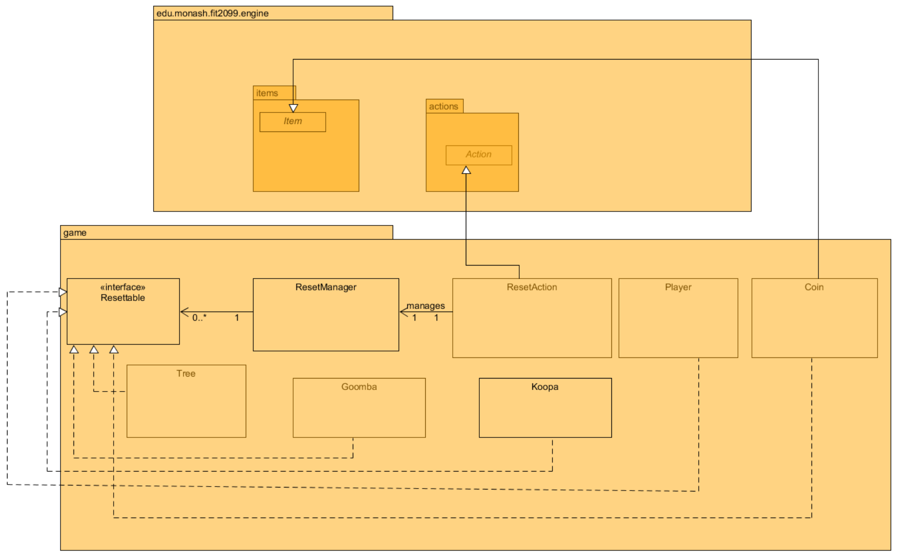
Changes from Assignment 2 to Assignment 3

Toad will now give the Player a Bottle if they do not already have one.

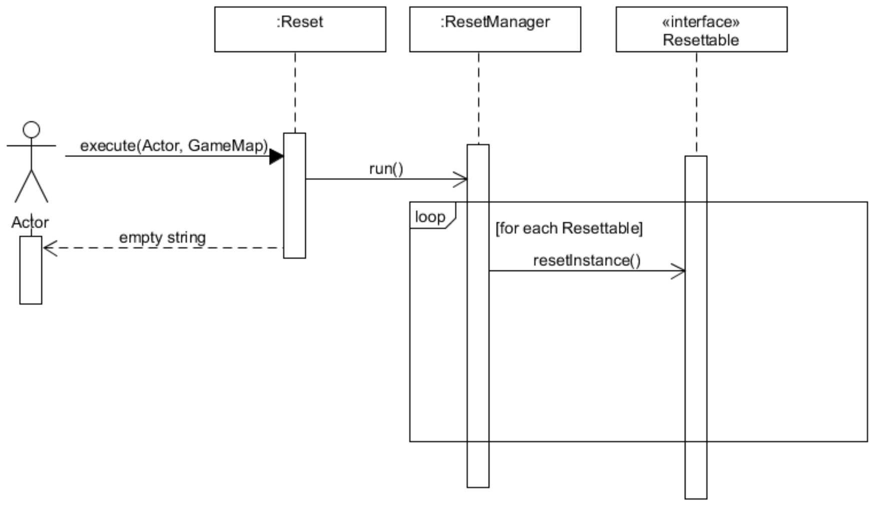
REQ 7: Reset Game Design Rationale

For reference, here are the class diagrams and sequence diagrams.

Class Diagram



Sequence Diagram



Rationale

I have created an extra class called ResetAction which extends to an Action. To keep track of the number of times the Player can reset, I have created a counter. When Reset is run, it will reduce one on the attribute making sure that the Reset action is not available for the Player anymore. As other classes which are required to reset their attributes, I have extended the Resettable interface to each of them, and all the classes will perform their resetting in their own implementations. This allows our ResetManager code to not have complicated code to reset every enemy, item, and so on. This provides extensibility for the code as the future code is only required to implement the Resettable interface and do their own logic in the class and not Reset Manager.

A major design flaw for this REQ is that ResetAction does not really utilize the meaning of Action. In the execute function, it doesn't even use the parameters given to it but instead just calls the ResetManager to perform the resetting of instances. However, it does not seem that there is another way to design this class since we need this Reset to be an Action so it could be registered in the Player's action as one of the actions available and able to reset the game. The advantage of creating an additional class of ResetAction is that it follows the single responsibility principle whereby the Reset Manager only manages the reset while the ResetAction class is an Action which runs the Reset Manager.

ResetManager Class

Purpose: A global Singleton manager that does soft-reset on the instances.
Single Responsibility Principle: It manages all the Resettable instances and resets them when the reset action is called
Open-Closed Principle: None
Liskov Substitution Principle: None, it cannot extend from any class due to its different usability
Interface Segregation Principle: None
Dependency Inversion Principle: This principle is followed as we will call resetInstance method on multiple classes, actions, actors and more! With a layer of abstraction, this class will only need to depend on the Resettable interface and not the instances themselves to be reseted!

ResetAction Class

Purpose: It's an action class which is used when the reset option is chosen by the player
Single Responsibility Principle: It's only responsibility is to reset the entire game
Open-Closed Principle: It is extended from Action and can be modified to suit its usability!
Liskov Substitution Principle: None
Interface Segregation Principle: None
Dependency Inversion Principle: None

Changes from Assignment 1 to Assignment 2

None.

Work Breakdown Agreement

Assignment 1

Dates are YYYY-MM-DD.

Naavin Ravinthran

- `REQ6: Monologue` by 2022-04-08
- `REQ1: Let it grow!` by 2022-04-08
- `Assignment 3 - REQ3: Magical fountain` by 2022-05-20

WBA Acceptance I accept this WBA

Yi Zhen Nicholas Wong

- `REQ5: Trading` by 2022-04-08
- `REQ7: Reset Game` by 2022-04-08
- `Assignment 3 - REQ1: Lava Zone` by 2022-05-20

WBA Acceptance I accept this WBA

Yu Zhang Ooi

- `REQ2: Jump Up, Super Star!` by 2022-04-08
- `REQ3: Enemies` by 2022-04-08
- `Assignment 3 - REQ2: More allies and enemies!` by 2022-05-20

WBA Acceptance I accept this WBA

Team tasks

- `Assignment 3 - REQ4: Minecraft` by 2022-05-20
- `REQ4: Magical Items` by 2022-04-08
- `Assignment 3 - REQ5: Ranged Combat` by 2022-05-20

WBA Acceptance Naavin Ravinthran - I accept this WBA
Yi Zhen Nicholas Wong - I accept this WBA
Yu Zhang Ooi - I accept this WBA