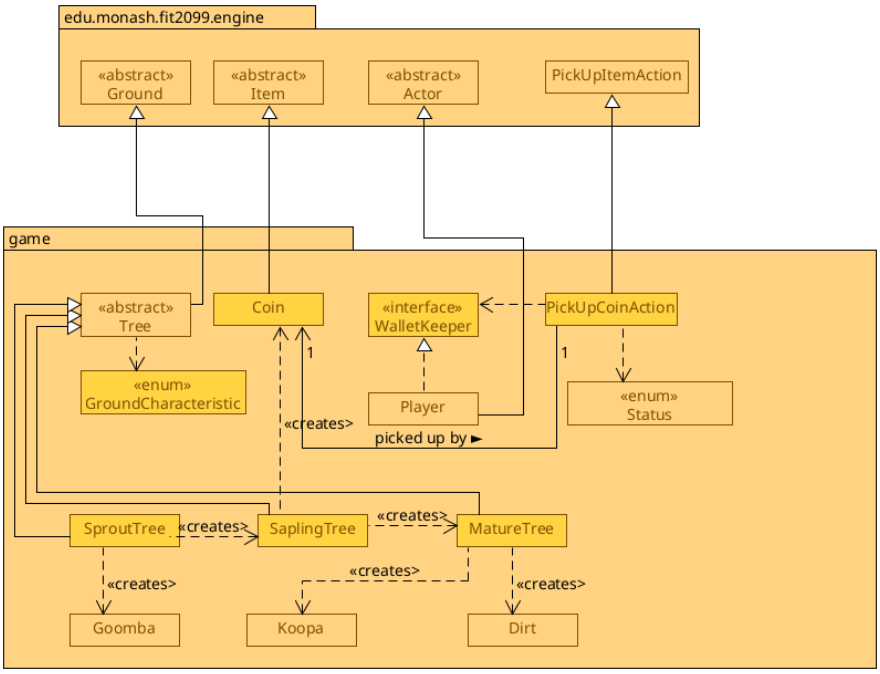
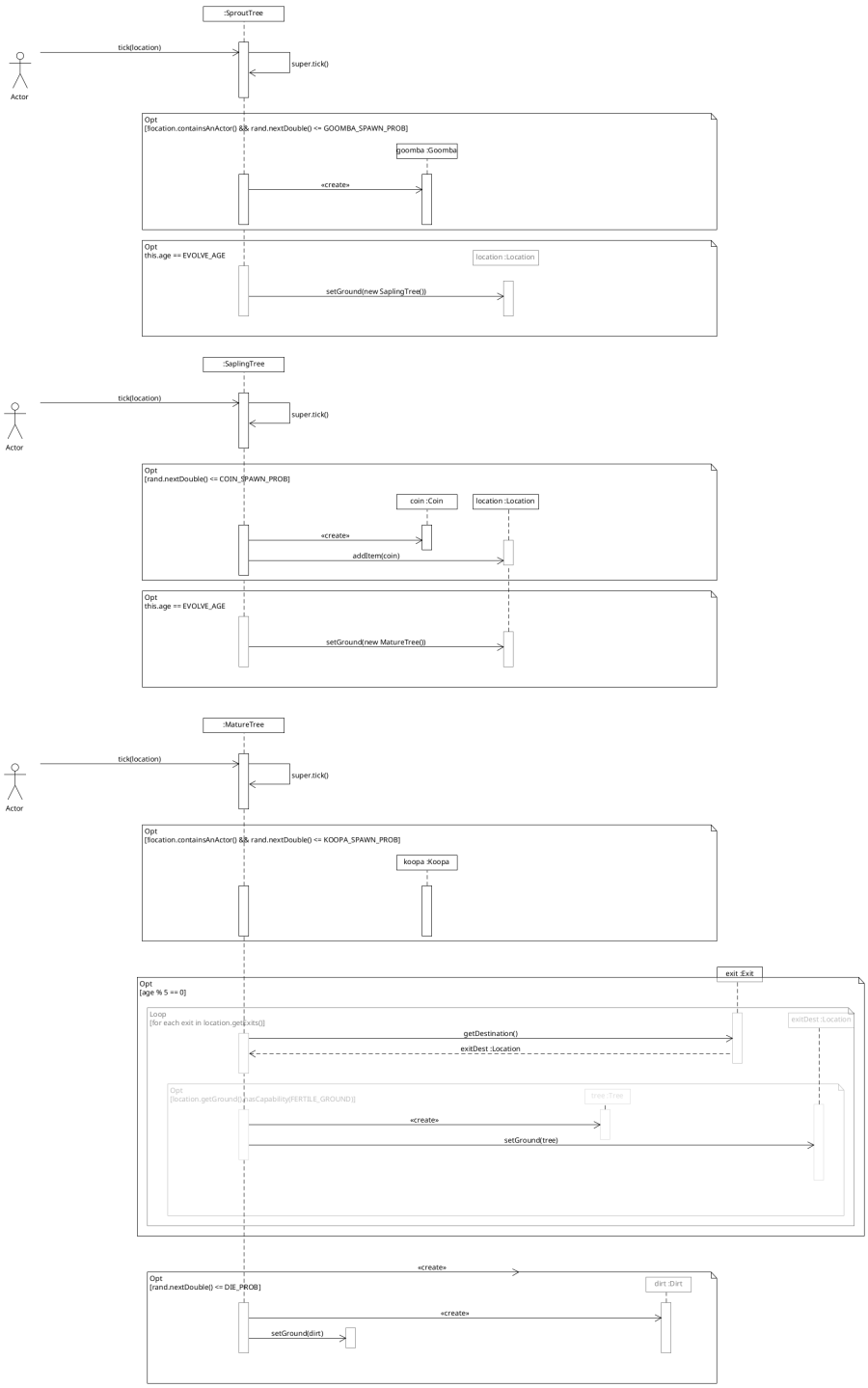
REQ1:Let it grow! Design Rationale

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

Class Diagram



Sequence Diagram (Tree's tick method)



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.

Problems WalletKeeper interface fixes

The interesting thing here is the WalletKeeper interface. As an analogy, in the real world, you would find a $5 coin and a $10 coin on the floor and pick those up, you would have two separate coins on your person. A $5 coin and a $10 coin. The usefulness of this is that it mirrors the real world more closely, which is useful beause it is easier to think about since we deal with it ourselves. There are a few downsides though.

In this game, having that logic would get *very* messy, *very* quickly. To find how much money an actor has, in the Actor-derived class, it would need to manually loop through the entire inventory, look for anything that resembles a coin, and add it to a "total" amount of coins.

Now, consider that you accumulate hundreds of $5 coins. you would have way more than the 26 extra actions ('a' to 'z') to drop each individual $5 coin.

How about, if in the future, we would want to do something with the money, and buy or sell items? (This is actually part of REQ5). In the real world, there is some messiness with this when handling transactions, such as considering when you want to buy a $4 item, but you only have a $5 coin, and the shopkeeper doesn't have a $1 coin to give you back as change.

The WalletKeeper interface fixes all these problems. We would have an entity, for example the "Player" class implement this interface. Rather than keeping individual coins, it lets the player keep a virtual "wallet" with money being accumulated it in as a number instead of individual coins. This means it will always have an up-to-date balance for the player's wallet.

As for the actions for dropping coins, there would be no more coins to drop.

For the trading problem, the WalletKeeper interface would enforce a way to withdraw or deposit to-and-fro your wallet.

It also adds some additional functionality. We can now use it for *any* actor that implements WalletKeeper, not just the Player. Looking at PickUpCoinAction, note how it has a dependency on WalletKeeper. This abstraction is useful, because ideally we want a Single Responsibility for the class and not hard-code wallet functionality into specific classes like Player, which we only want to do a single thing. Now, we can, if needed, create a new class "Bandit", that will have a behaviour that makes them wander and pick up coins as well.

Problems with the WalletKeeperInterface

There are some downsides to it. You cannot drop coins anymore, if that would need to be done; The coins items are destroyed as you insert it into your wallet.

You can't have "special" coins that do things, for example, imagine wanting "red coins" that besides adding to your balance, gives you an invincibility status for a period of time.

PickUpCoinAction

The PickUpCoinAction was needed instead of just using the default `PickUpItemAction` that the Item class returns in `getPickUpAction` . The reason for this is that special behaviur needs to be put in place besides just picking up the item and placing it into your inventory. We need to add in the amount of that coin into the balance of the actor as well, and then remove it from the inventory. Nevertheless, it is still the action of picking up an item, with some common characteristics between it, such as the menu description of picking up the item, and the removal of the item from the map after picking it up, as such, we decide to extend this class to re-use its functionality.

If you're wondering about the dependency on `Status` for PickUpCoin action, it's to check if the actor is allowed to handle money with a `CAN_MANAGE_MONEY` status to see if it is possible to add money to wallet of the actor or not. An example would be perhaps if there was a debuff to cause Mario not to be able to pick up any coins for X amount of turns.

Design of Tree (OLD, see next section)

Looking at the sequence diagram for the tree's `tick()` method, you can see how long it is with a lot of alternating logic on class communication depending on what stage the tree is at.

An Alternative design would be to have separate classes for different stages of a Tree, and each time it needs to grow, the tree would remove itself from the map and replace itself with the next stage of the tree.

The way the sequence diagram would then look would be to have 3 separate sequence diagrams, each quite short, and all with less width than this sequence diagram, because it would not need to worry about the classes that it won't communicate with in that current stage, for example, in a Sprout sequence diagram, it won't have a `coin :Coin` or `koopa :Koopa` because those are only interacted with in the other stages. This is quite nice as we can separate the concerns of the behaviour of different stages of the tree into separate classes. If we were want to change some of the behaviour of a particular stage of the tree (e.g: Sprout) , we could do it in the Sprout class instead which is specifically single responsibility for that stage of the tree.

The problem with that approach is that it would be harder to have the common attributes instances shared across all the tree stage classes. For example, what if we want to give trees a HP bar and ability to be punched down in any stage of the tree? Then, all three classes would need to be modified to account for this. For example, the instance of the age of the tree needs to be kept across all stages of the tree, if we were to destroy the tree and create a new more matured tree to replace it with, we would need to put an "age" in the constructor to keep this age value. As we add more variables that stay the same across all the stages of the tree (e.g: Colour of the tree, Ground Characteristics, etc.) they would have to all be in the constructor as well. Additionally, the logic for tree growth would be scattered instead of being in a single place, e.g: The logic for a sprout growing to a sapling would be in the Sprout class, but the logic for the Sapling growing to a Mature tree would be in the Sapling class. Hence, we use a TreeState enum to keep track of the tree's stage of growth instead.

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

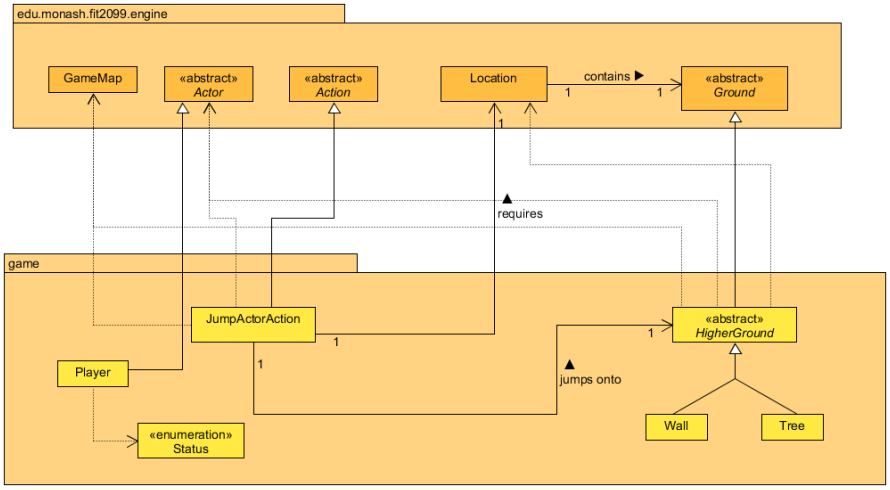
It was decided to go with the alternative approach with having separate classes, each extending the "Tree" class, for the different stages of the tree. The drawbacks are outweighted by the simplicity this new design has by leveraging a Single Responsibility Principle, with each class being able to concentrate 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.

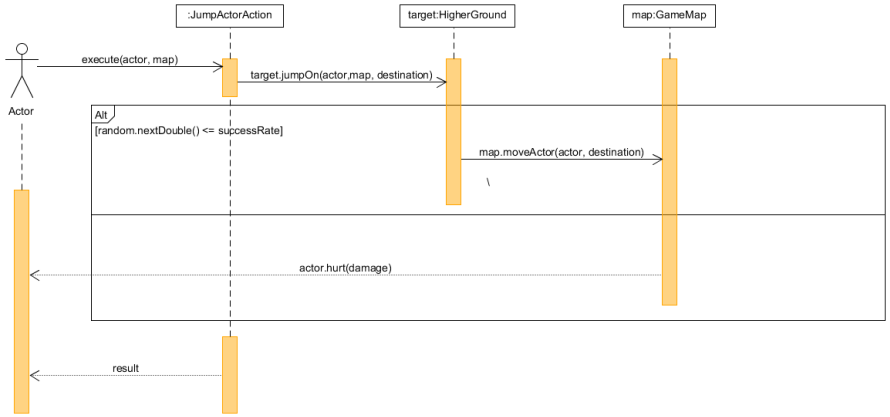
REQ 2: Jump Up, Super Star! Design Rationale

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

Class Diagram



Sequence Diagram



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.

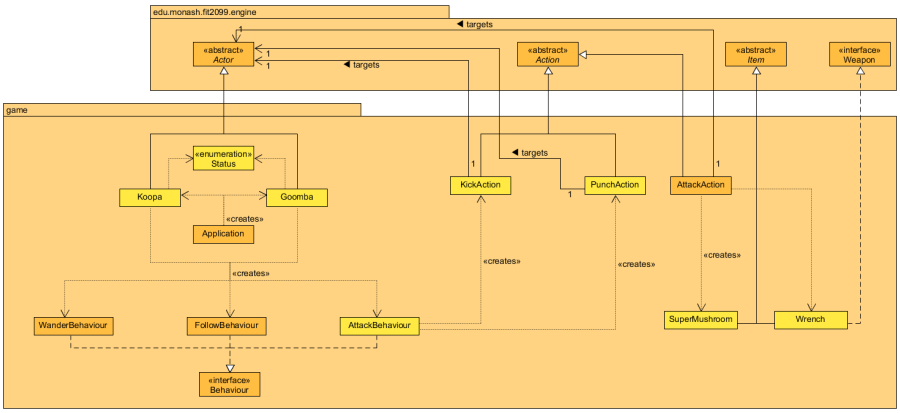
The HigherGround class is a base class extended by the Tree and Wall classes to allow the program to identify both classes as one single higher ground entity. It contains the jumpOn method which moves the Player to the specified high ground if the generated probability is within the range of the success 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.

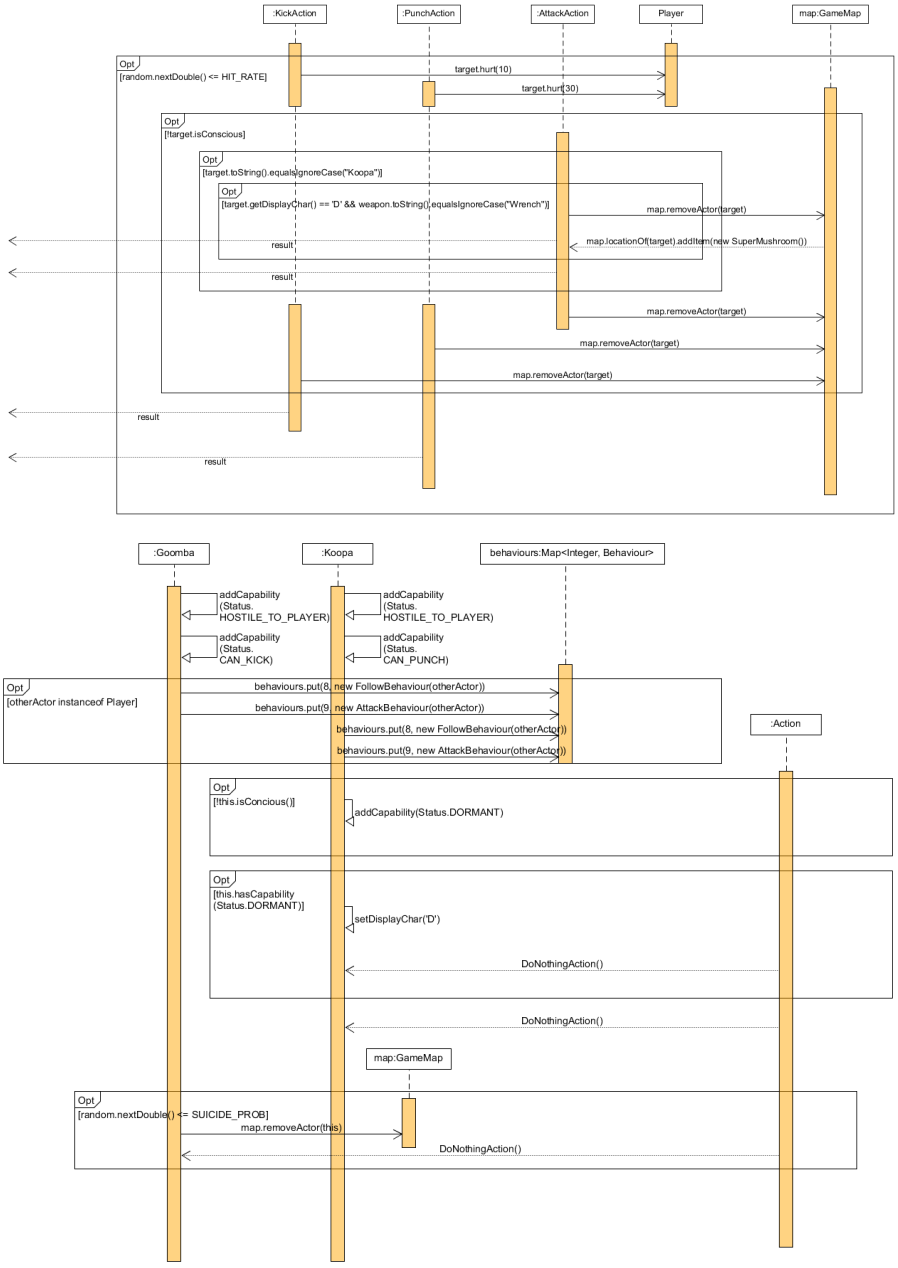
REQ 3: Enemies Design Rationale

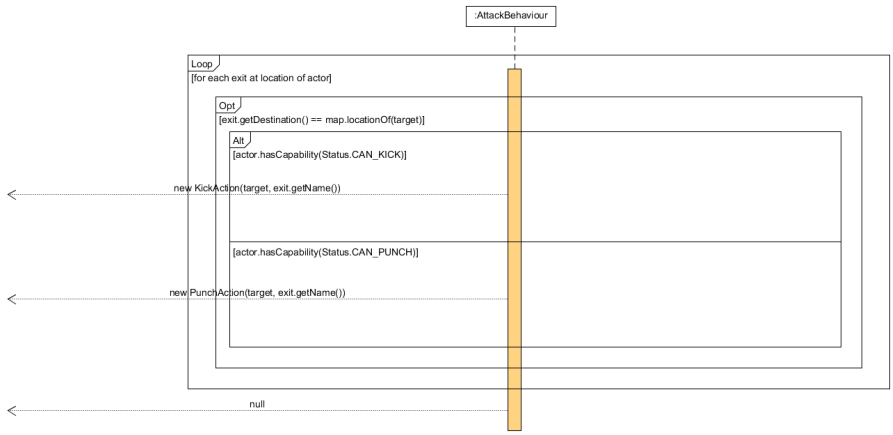
For reference, here are the class diagrams and sequence diagrams.

Class Diagram



Sequence Diagram





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. The Koopa and Goomba will then utilise the PunchAction and KickAction classes respectively which are instantiated in the AttackBehaviour class to attack the Player. The PunchAction and KickAction classes will generate a random probability and compare it to the hit rate of the attack, if it is within the range of the hit rate, the Player will take damage. My previous approach for this was to have the Koopa and Goomba have the capability HOSTILE_TO_ENEMY but this would have resulted in the Koopa and Goomba attacking themselves instead of the Player which was why I went with this approach.

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.

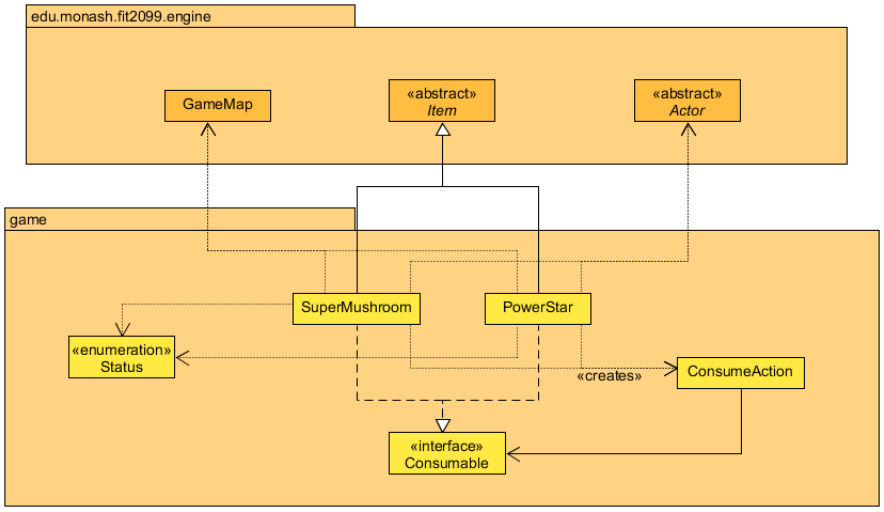
The AttackBehaviour class creates the KickAction class which is used by a Goomba to kick(attack) the Player and the PunchAction class which is used by a Koopa to punch(attack) the Player.

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.

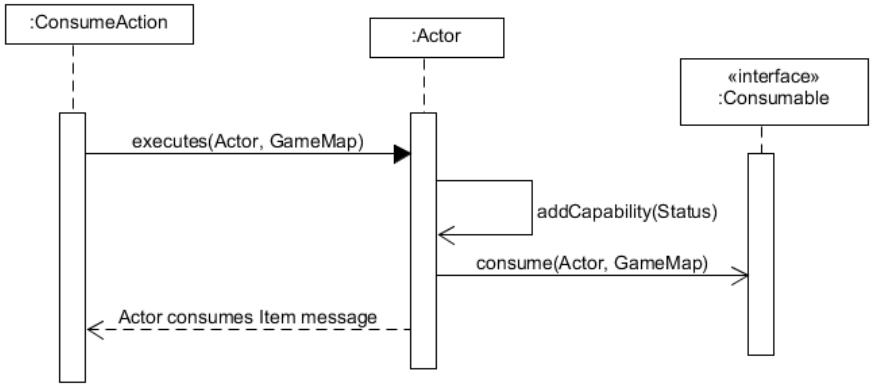
REQ4: Magical Items 🍄 Design Rationale

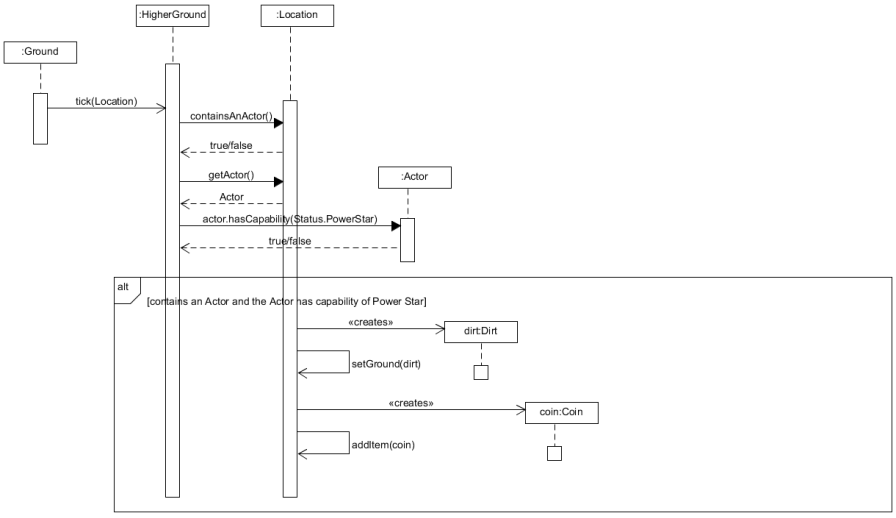
For reference, here are the class diagrams and the sequence diagrams.

Class Diagram



Sequence Diagram





Rationale

The Consumable interface will have two methods that need to be implemented.

1. effect() - This will return the `Status` that consuming it will give. For example, consuming a SuperMushroom would give it a TALL status, so this would return `Status.TALL`
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.

Note that we could just handle adding the effect of consuming the item in consume(), but we want to make sure that all Consumables have an effect, hence the effect() method.

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. Allow for pseudo "multiple inheritance". Imagine if we decide to create a another class called Drinkable, which is similar to Consumable. So we go ahead and create a DrinkableItem class. Now, we decide we want to have an item that is *both* drinkable and consumable, We aren't able to extend both classes! Perhaps the best we can do is create a class called ConsumableDrinkableItem and inherit from that instead, and you can see how this would get messy very quickly as we add other types of item classes, and not having a base type we can use, for example in the ConsumeAction class. This would adhere to the Interface Segregation Principle, where the large interface of "DrinkableConsumableItem" is split into smaller ones called "ConsumableItem" and "DrinkableItem".

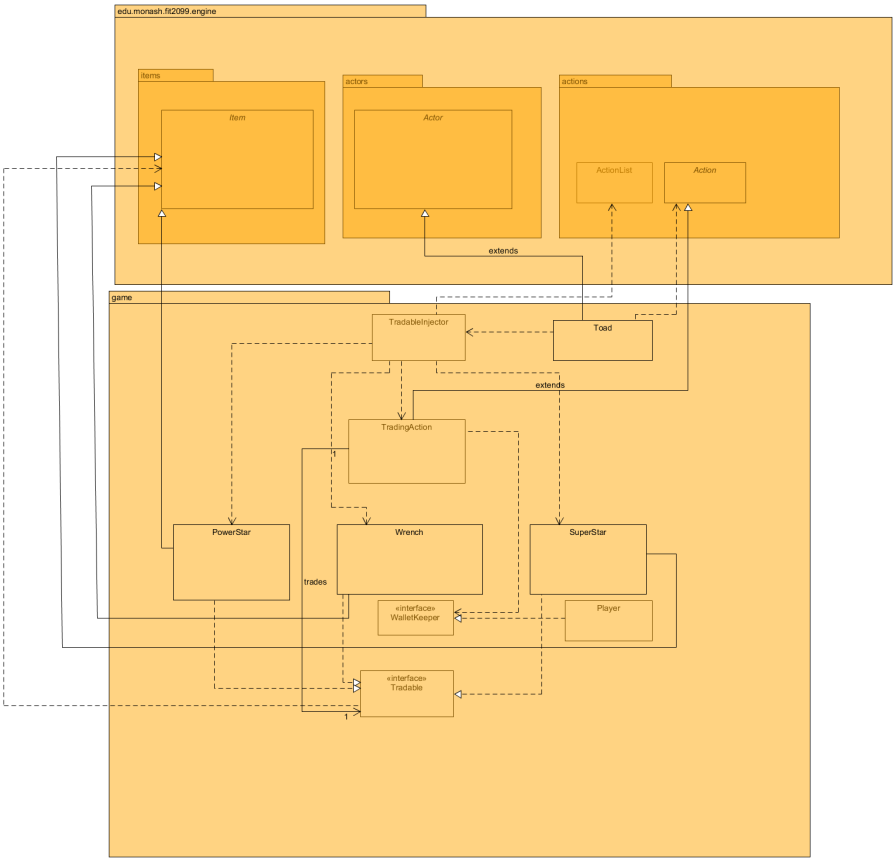
Changes from Assignment 1 to Assignment 2

None.

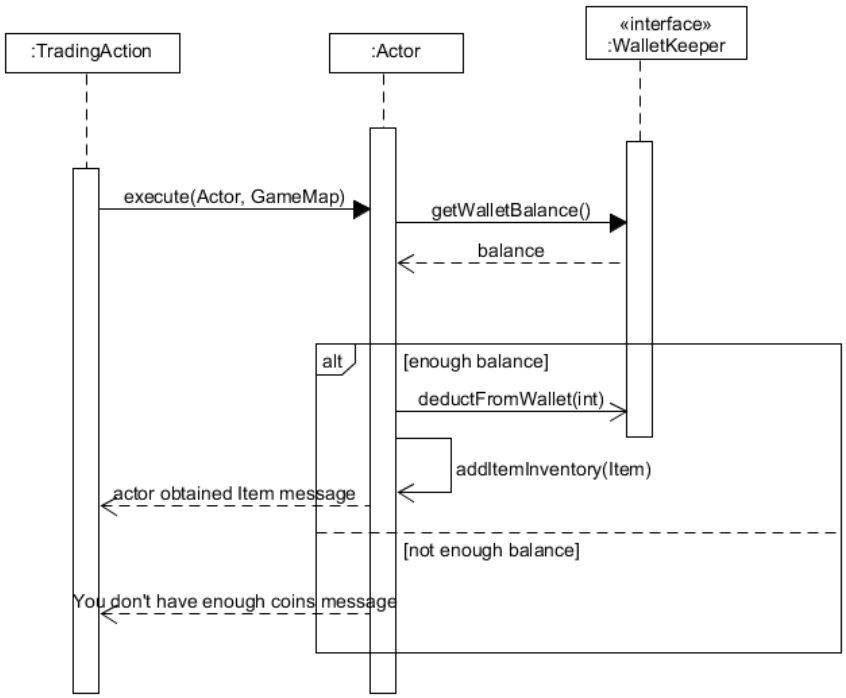
REQ 5: Trading Design Rationale

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

Class Diagram



Sequence Diagram



Rationale

In summary, Toad is an interactive actor who the player can interact with. 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: This is followed as well, as we extended from the Action class
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: The reason we extended Toad as an Actor is because in the World engine, the player actually detects its surroundings and check for what allowableActions the surrounding actor has. By making Toad an Actor, we are not required to re-code existing checkings of surrounding and just implement the allowableActions which can be performed with Toad. Thus, we decided that it will be better to make Toad an Actor.
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

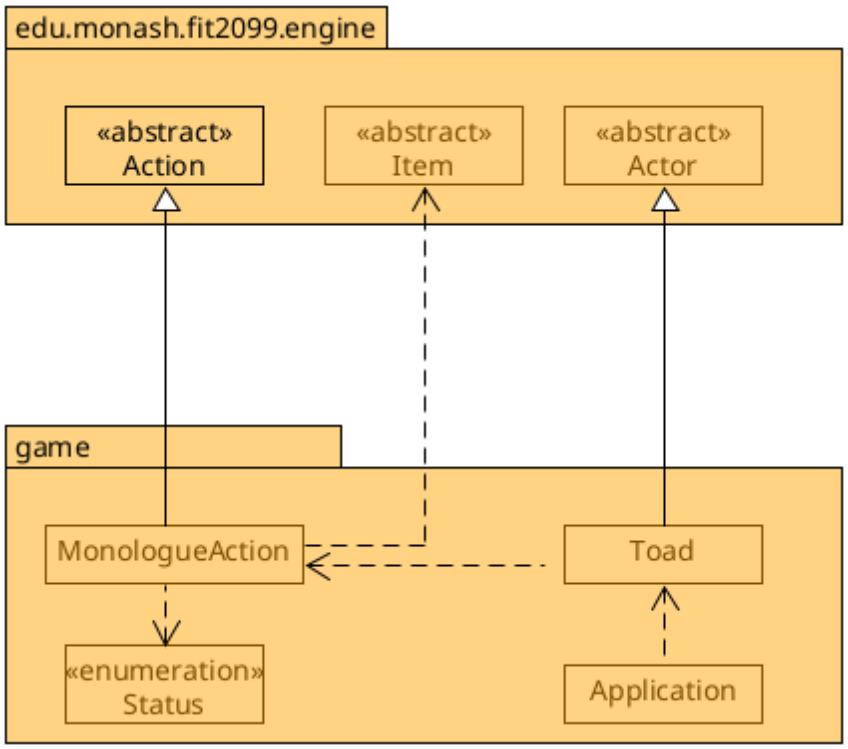
I decided that a Tradable interface and TradingInjector class was required after Week 8. The Tradable interface follows DIP which allows the TradingAction class to only use those methods in the Tradable. This is good design since it will prevent TradingAction class to access from unnecessary methods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

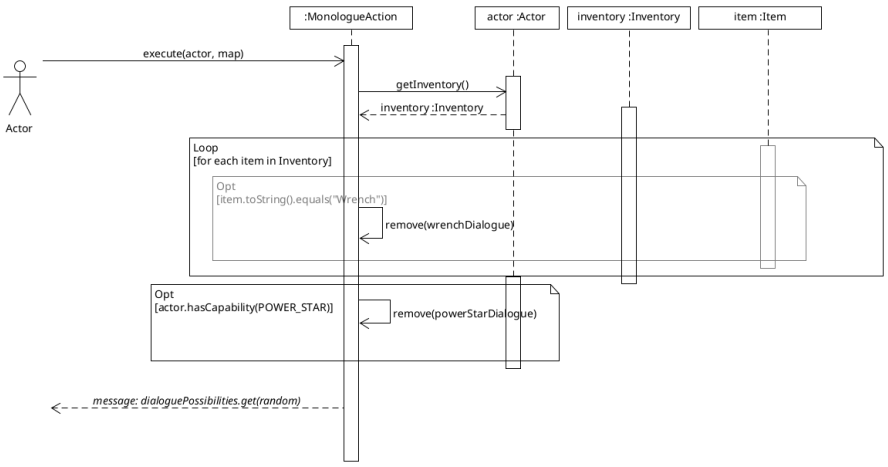
REQ6: Monologue 💬 Design Rationale

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

Class Diagram



Sequence Diagram



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.

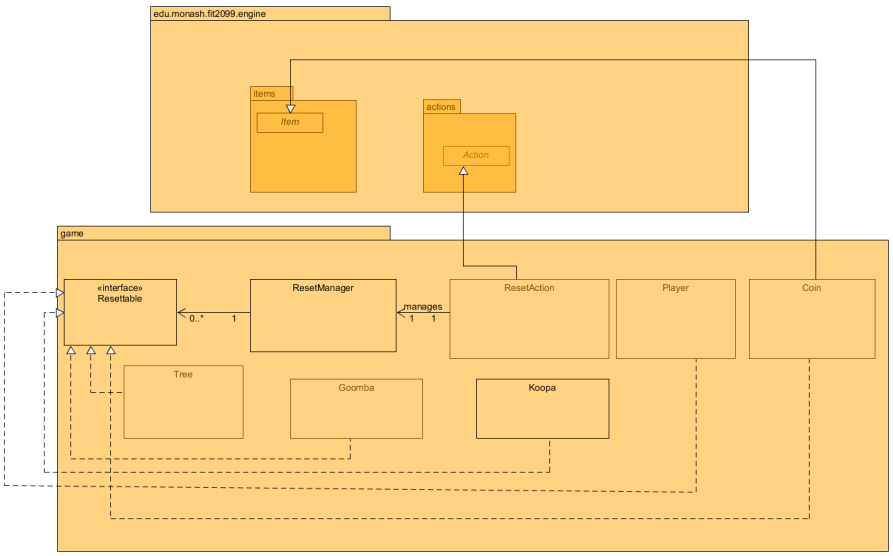
Changes from Assignment 1 to Assignment 2

None.

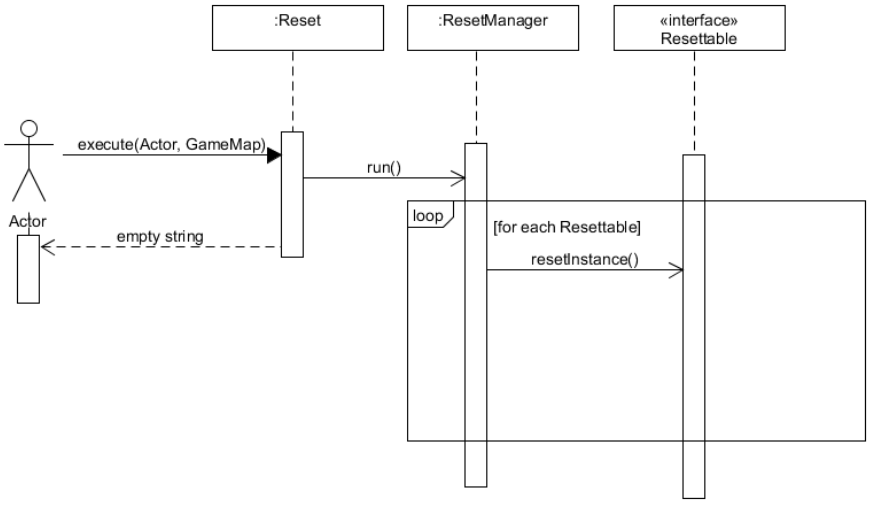
REQ 7: Reset Game Design Rationale

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: Since it is an Action which can be performed by the player, I decided that it will be great to extend from Action class since it requires the menu, the hotkey and even the execute method! The other reason

also being that the player can only call the Reset option if it is an Action, so undoubtedly it is not possible to not extend it from an Action.
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

WBA Acceptance I accept this WBA

Yi Zhen Nicholas Wong

- `REQ5: Trading` by 2022-04-08
- `REQ7: Reset Game` by 2022-04-08

WBA Acceptance I accept this WBA

Yu Zhang Ooi

- `REQ2: Jump Up, Super Star!` by 2022-04-08
- `REQ3: Enemies` by 2022-04-08

WBA Acceptance I accept this WBA

Team tasks

- `REQ4: Magical Items` by 2022-04-08

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