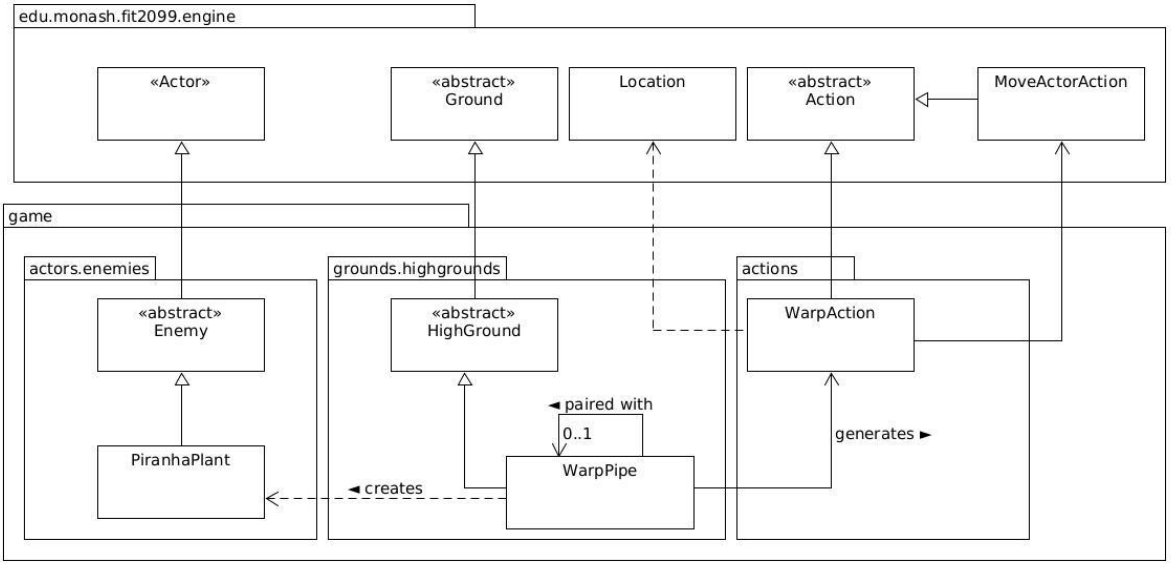
Assignment 3: Further Design and Implementation

Design Rationale

REQ1



WarpPipe

Extends HighGround. WarpPipe may be paired with another WarpPipe by calling WarpPipe.setPair(WarpPipe pairPipe). This allows an actor standing on its location to execute a WarpAction to teleport to the paired pipe.
WarpPipe will generate a Piranha plant on top.

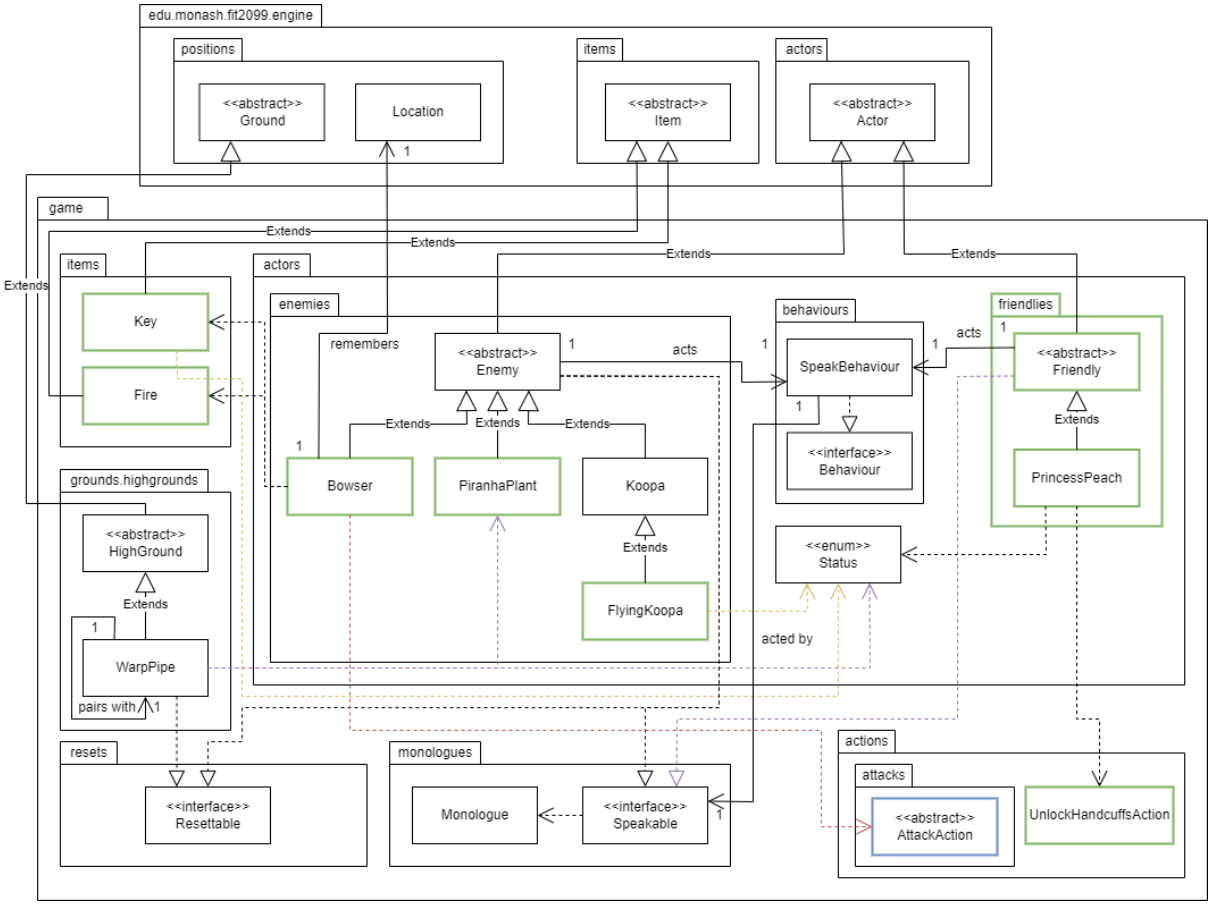
Pre-conditions
- Pairs placed onto game maps and linked using WarpPipe.setPair(WarpPipe pipePair)

Post-conditions
- Will generate a PiranhaPlant on its Location once iff.
  1. The location is vacant.
  2. This is the first PiranhaPlant generated by it OR a reset has just been called.

WarpAction

Creates a new MoveActorAction but executes this after removing any actor located at the paired WarpPipe location.

REQ2



Class Contracts

PrincessPeach

Pre-conditions

- Manually placed on the appropriate GameMap.

Post-conditions

- Will remain stationary in a handcuffed state.
- Cannot attack or be attacked until an actor holding a Key executes the UnlockHandcuffs action.

Bowser

Pre-conditions

- Manually placed on the appropriate GameMap.

Post-conditions

- When the player is adjacent, will follow and attack the player whenever possible.
- Drops Fire on the ground whenever it attacks.
- Drops a Key on the ground when it is defeated.

PiranhaPlant

Pre-conditions

- Generated on top of all WarpPipes at the second game turn and at a reset.

Post-conditions

- Does not move.
- Attacks the player when adjacent.

FlyingKoopa

Pre-conditions

- Generated from a Mature tree when appropriate.

Post-conditions

- Will exhibit the same behaviours as Koopa.
- Will fly over HighGrounds when wandering or following the player.

Resettable

Pre-conditions

- The resetInstance method is called by the ResetManager to signal a reset.

Post-conditions

- Must execute an implementation to reset itself on its next game turn when the resetInstance method is called to signal a reset.
- Must signal to the ResetManager when it has finished resetting itself.

Design Version 1.3.1

The Resettable instances do not need to signal to the ResetManager when they've reset themselves as the ResetManager does not need to wait for them to complete the reset (YAGNE).

Design Version 1.3

Four new actor classes need to be added. Three fit into the enemy abstraction and will extend the Enemy class. One (Princess Peach) fits into a friendly NPC abstraction with Toad. The main feature of friendlies is that they cannot be attacked and cannot attack. By extracting an abstract Friendly class, it is easy to ensure all of these friendly actors cannot be attacked by giving them the PROTECTED status, which follows DRY and OCP, since less code needs to be written when adding a new friendly. The feature of friendlies not being able to attack is difficult to enforce, since if there is code in the superclass Friendly playTurn which prevents an AttackAction then this must be called by the subclasses, which is not enforceable. Regardless, having an abstract Friendly class is the basis for following DRY and OCP when adding new friendly NPCs.

The actual reset implementation cannot always occur in resetInstance() since the dependencies for some class resets are only provided in the tick() or playTurn() method. To avoid breaking LSP, the reset implementations for all Resettable instances should occur in their tick() or playTurn() method. The reset will become the main operation of the instance on its game turn, and will take priority over other game turn actions. For example, in Enemy, when the reset is signalled the playTurn() method will do the reset operations and nothing else. A good way to ensure LSP is not broken is by making a ResetInstanceAction which is returned by Resettable actors from their playTurn() method on the reset game turn. However, this is not applicable to non-actor Resettable instances that have a tick() method as no action is returned and executed by the game engine for that method. If an action is created and executed within tick() this will break LSP since it is expected that non-actors do not execute actions. This is evidenced by the abstract Action execute method, which contains the Actor acting as a parameter. So the reset operations should not be implemented in a ResetInstanceAction, and should be implemented in the tick() or playTurn() method of the Resettable. What this means is the ResetManager is only responsible for signalling to a Resettable to reset itself, and the Resettable is responsible for implementing the reset in whatever method is appropriate for the class, and signalling to the ResetManger when done. This follows SRP, but the class contracts must be adhered to. Even though the actual reset implementations do not occur in resetInstance and occur in different methods depending on the Resettable, LSP is not broken since the implementation occurs in whatever method the class uses to play its game turn, and the reset operations are what should be executed by an instance on its game turn when a reset is active.

Design Version 1.2

The existing game reset functionality can make better use of the game engine code. The game engine iterates all the actors and locations and calls the tick or playTurn method with the dependencies required for reset operations. If the main reset implementation is moved into the tick or playTurn method, there will be no need for associations with the ResetResources class, and it can be removed. To run the reset, the ResetManager should interact with each Resettable by sending a signal instructing the instance to reset itself, and then receiving a signal indicating the instance has been reset. This way the reset of each Resettable is independent of the others. If one instance takes longer to reset the others may still continue as normal. The first signal can be implemented through the existing resetInstance method, and the second signal can be implemented with a new public method

in ResetManager. and the actual reset implementation occurs in the tick or playTurn methods of the Resettables. This does not break SRP as the ResetManager is still only responsible for activating, deactivating, and controlling the number of resets, and the Resettable is still only responsible for performing their own reset operations and signalling to the ResetManager when done. This better adheres to OCP since new Resettables only need to adhere to the Resettable interface and need not worry about the ResetResources class (now non-existent). Even though the actual reset implementation occurs in tick or playTurn, LSP is not broken since the reset operation becomes the responsibility of tick or playTurn at the game turn after the reset signal, and the main operations are blocked. DIP is also better adhered to since the concrete classes implementing Resettable no longer have a dependency with another concrete class (ResetResources).

Design Version 1.1.1

The key feature of Princess Peach is that she can't move, attack, or be attacked, until an actor with a key unlocks her handcuffs. These features can be broken into two parts:

- The actor is in a handcuffed state and cannot perform any actions.
- The actor is in a protected state and cannot be attacked.

It is assumed that once Peach's handcuffs are removed, she still cannot be attacked as she is a friendly NPC. These two features can be implemented as statuses using the existing Status enum. The second feature is also applicable to other friendly NPC characters such as Toad, but the first is not, so they should be distinct statuses.

Design Version 1.1

When Bowser attacks, a fire is set on the ground. This should be done when the AttackAction is executed by Bowser, however it is specific to Bowser so placing this code in AttackAction would break SRP and Separation of Concerns. The next best thing is for the fire to be set in Bowser.playTurn() before the AttackAction is executed. This would require the target actor of the AttackAction, which can be found if Enemy.findTarget() is refactored to return the target. The only consequential refactoring is that Enemy.playTurn() directly adds the attack and follow behaviours, which can be reused by Bowser (DRY).

Note: instanceof is needed in Bowser to conditionally set the ground on fire, since this is only done when Bowser attacks. This is not going to cause an if-else chain since only one code block is executed regardless of the condition. If the ground is set on fire under other conditions, this can be abstracted away, however this is not known at this time and is not included.

Design Version 1.0

Four new classes are required. Three are part of the enemy abstraction and will be concrete classes extending the Enemy class. These are Bowser, Piranha Plant, and Flying Koopa. The other class (PrincessPeach) is a friendly NPC, and will be a concrete class extending Actor.

The three new enemies respond to the game reset, and will implement the Resettable interface. PrincessPeach remains stationary until the game ends, so a reset is not needed for this class. The Mature tree class needs to be updated so that after the 15% spawn rate Koopas and Flying Koopas are spawned equally.

Flying Koopa has the same features as Koopa and responds to the game reset in the same way, so it will extend the Koopa class to make significant use of the existing code (DRY). The only difference is that FlyingKoopa can always travel over HighGrounds. Currently, this is only allowed if the actor has the Powerstar effect. One solution is to make the Flying Koopa always has this effect. The other way is to change the code in HighGround so that a more abstract effect TRAVEL_ON_HIGH_GROUND is used. The latter will be used as the abstraction is useful for extensions and is more appropriate for FlyingKoopa, since it does not actually have the POWERSTAR effect. Note that FlyingKoopa should not break LSP since they have the same functionality except that FlyingKoopa will always have the TRAVEL_ON_HIGH_GROUND effect.

When Bowser attacks a fire will be set on the ground. The fire needs to damage any actor standing on that ground. This can be implemented as a Fire class which extends the abstract Ground class. At every tick, the fire ground can check if an actor stands on it, and if so inflicts the appropriate damage.

When the player defeats Bowser, a key will be dropped on the ground. The key needs to be able to unlock PrincessPeach's handcuffs to end the game. The effect of being handcuffed can be implemented as a HANDCUFFED status, which will prevent the actor with this status from moving around. There can also be a Handcuffs item which will allow the actor holding them to inflict the HANDCUFFED status on other actors. This is not implemented as it is not needed for this requirement, however it is simple to add later if needed. The key can be implemented as a Key class which extends Item. The actor holding the Key will have a UNLOCK_HANDCUFFS capability, and can then execute a UnlockHandcuffsAction on the handcuffed actor, by virtue of holding the Key.

Note: it should be possible to remove the handcuffs of any actor, however this would mean returning a UnlockHandcuffsAction from the abstract Actor class, which cannot be done. At this point only PrincessPeach needs to be un-handcuffed, so the action can be returned from that class. However, if any other actor classes should be un-handcuffed, they need to return this action explicitly, instead of it being done in Actor.

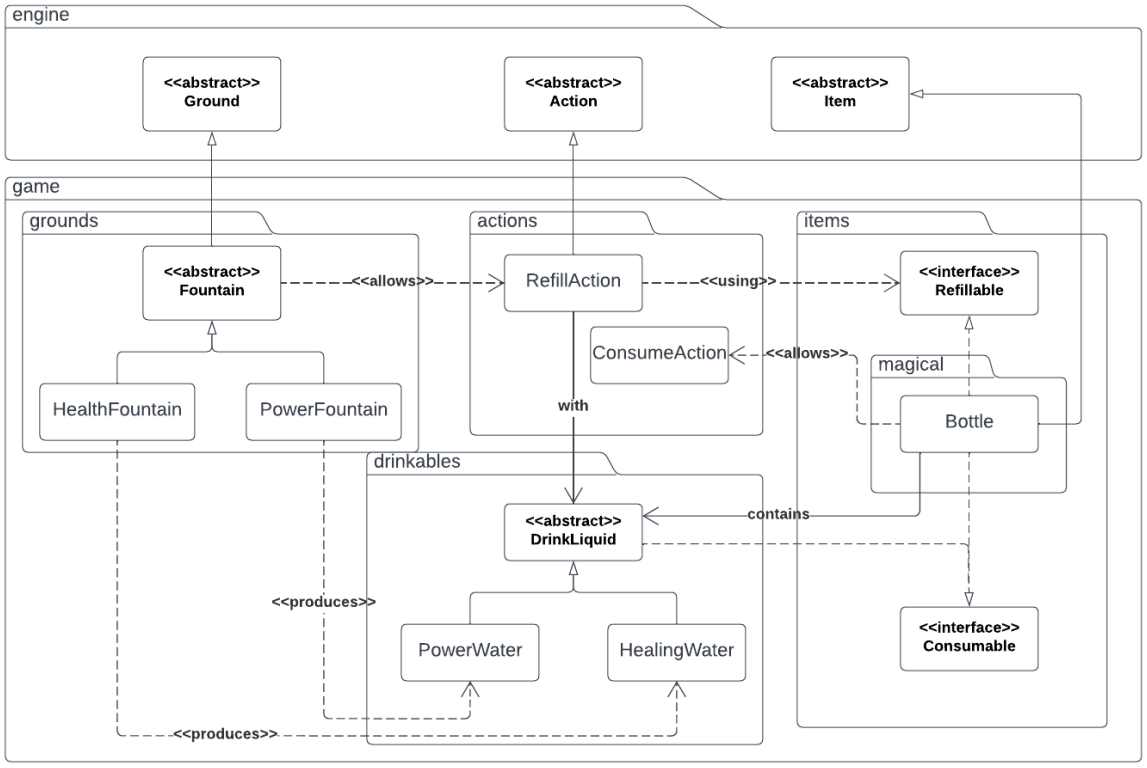
The Piranha Plant cannot move around. This could be implemented by giving it the HANDCUFFED effect to make use of the code preventing actor movement (DRY). The problem with this is it would be possible for the player holding the Key to un-handcuff the Piranha Plant so it can move, which should not be possible. The PiranhaPlant class cannot have a MoveActorAction returned by any of its behaviours, so if this is achieved the Handcuffs are not needed, and no extra code is written and DRY isn't broken.

The PiranhaPlant is spawned on top of a WarpPipe (assume WarpPipe extends HighGround). The WarpPipe can only spawn a PiranhaPlant in the second game turn, and then cannot spawn another PiranhaPlant until a reset. This can be implemented by the WarpPipe having a GENERATE_PIRANHAPLANT status. This needs to be controlled by the

tick method (deactivated after second turn), and also reactivated after a reset. This means the WarpPipe class needs to respond to the reset, and should implement the Resettable interface.

Toad and PrincessPeach are both friendly NPCs and can be packaged together in an npcs package, but there is no need for them to depend on a Friendly abstraction class, as it would not be useful. The only difference is that the enemies can return an AttackAction, while friendlies shouldn't.

REQ3:



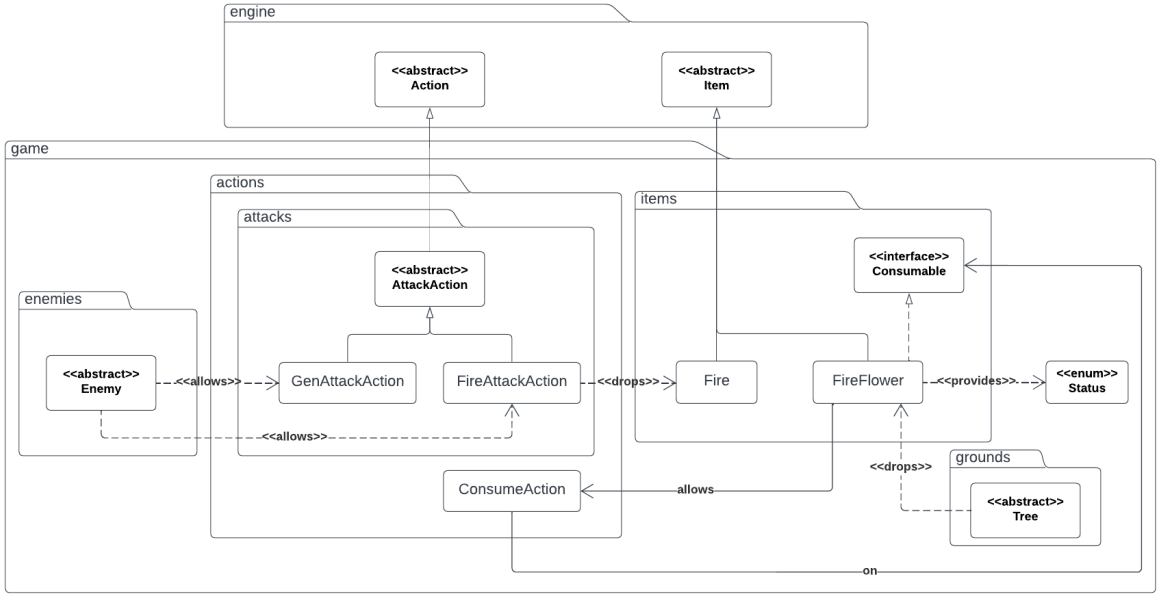
- <u>Bottle</u>
    - The bottle is implemented as a concrete class extending from Item abstract class. As it can also be consumed and refilled it implements the Consumable interface and the refill interface. This follows ISP as the bottle's different usage is separated into different interfaces.
    - The bottle has a stack of DrinkLiquids (at this stage, this is just water) that can be filled up by overriding the refill() method in the Refillable interface and be consumed by overriding consume() in the Consumable interface. This follows Open-Closed Principle as no modifications were needed in the interfaces to implement these features for Bottle. As the bottle is a magical item, it was given the ItemEffect enum STORE_LIQUID. In the future the bottle could potentially store more liquids so this capability could come in handy.
    - Preconditions: an Item object

    - Postconditions: a Bottle object
- Refillable Interface
    - This interface has an abstract method refill() that allows any class that implements it to be a refillable item that can be refilled in their own ways. This interface allows future extensibility as there maybe more refillable items or items that can be recharged for example a saber that needs to be recharged with electricity.
    - Preconditions: None
    - Postconditions: a Refillable object
- Abstract DrinkLiquid
    - This abstract class implements the consumable interface as an actor can drink water, following ISP. It is abstract as water itself is not instantiated and it encapsulates the similarity of its subclasses like water implementing consumable and toString(), following OOP and abstraction.
    - This abstraction also allows Open-Closed for potential future liquids like poison and DIP as its subclasses are inheriting from an abstract class.
    - Preconditions: None
    - Postconditions: a DrinkLiquid object
- Healing Water
    - This class extends DrinkLiquid, following DIP and has an instance variable HP_HEAL as a constant. This reduces the connascence to only name connascence when using it in the overridden consume() to heal the actor by 50 points.
    - Preconditions: a DrinkLiquid object
    - Postconditions: a Healing Water object
- Power Water
    - This class extends DrinkLiquid, following DIP and overrides the consume() from the Consumable interface to give an actor a POWER_UP Status enum. In the actor's overridden getIntrinsicWeapon() would simply increase the base attack damage. This allows good encapsulation as Power Water is not accessing the actor's intrinsic weapon but simply providing a buff.
    - Preconditions: a DrinkLiquid object
    - Postconditions: a Power Water object
- Abstract Fountain
    - This abstract class extends Ground rather than the item the actor is consuming the water it provides, not the fountain itself. To allow the player to get water from the fountain when it stands on it, the allowableActions() method would check if the player has a means to store the liquid and if the actor is on the fountain before adding a RefillAction at the fountain.
    - To get the special liquids from each fountain, an abstract getWater method is provided in Fountain so that its subclasses can override and return whatever drink liquids they like, therefore following DIP and OCP.
    - Preconditions: a Ground object
    - Postconditions: a Fountain object
- Power Fountain
    - This class extends abstract Fountain, following DIP and overrides the getWater method to return new PowerWater every time it's called.
    - Preconditions: a Fountain object

- Postconditions: a Power Fountain object

- Health Fountain
  - This class extends abstract Fountain, following DIP and overrides the getWater method to return new HealingWater every time it's called.
  - Preconditions: a Fountain object
  - Postconditions: a Health Fountain object
- RefillAction
  - This class extends Action as the player needs to be able to execute it using the menu. It has an instance variable called liquid of type DrinkLiquid that is initialised through the getWater method of Fountain which is passed in the constructor.
  - To refill the bottle, the execute() method would obtain the refillable item from the actor's storage and call its refill() method. While dependency injection could have been implemented by adding the getStorage method in fountain and passing that into RefillAction constructor, however, it would violate SRP in Fountain as it shouldn't know about where or not an actor has a storage item or not, it is simply a ground object.
  - Preconditions: actor has a liquid storage, actor is on a fountain
  - Postconditions: actor's liquid storage is refilled

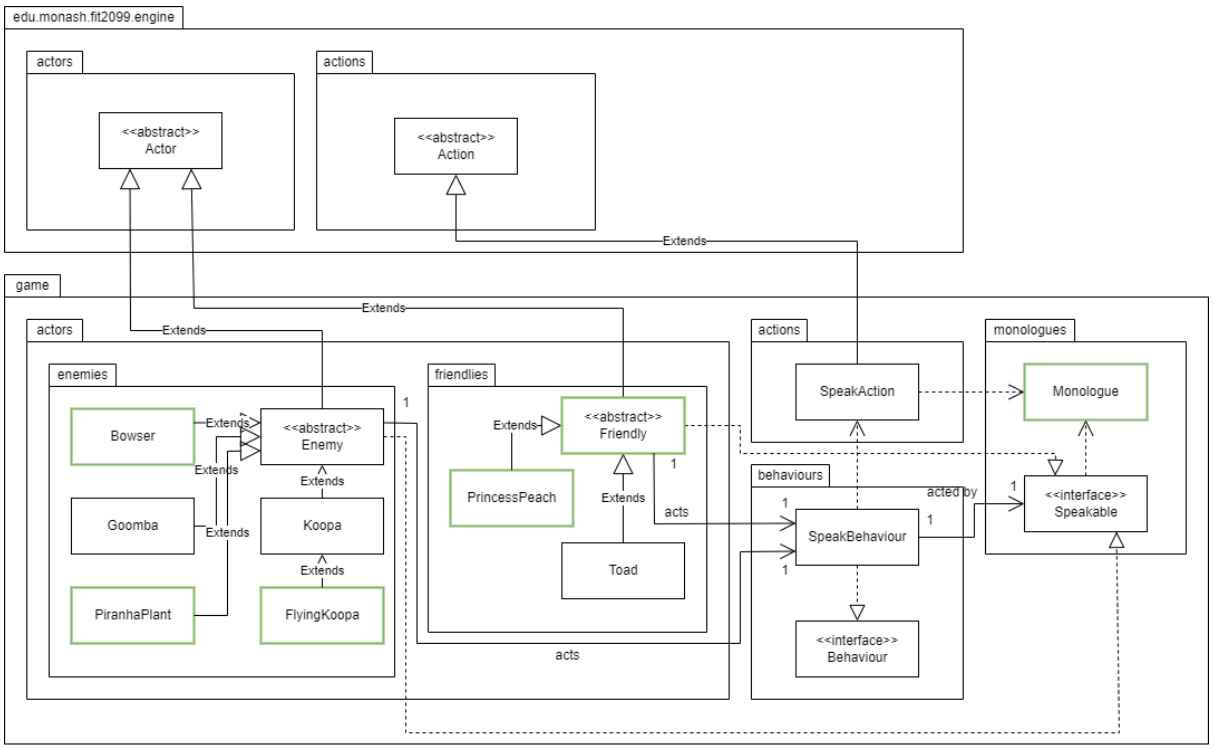
REQ4:



- FireFlower
  - This class extends Item as it makes more sense for it to be an item as it allows multiple fire flowers to be at one location and avoids resetting the ground to its previous state when fire flower is removed.
  - The fire flower 50% chance of being created in every tree growth stage is implemented by adding this to a grow() method in the tree class that can be overridden in sprout and sapling to grow into other tree phases and call the super.grow() to grow fire flowers. This not only improves the previous design by increasing SRP as growing trees has its own method, but also allows

abstraction as repetition and dependency on the fire flower class is reduced to only in the tree class.
    - To consume the fire flower, it implements the consumable interface and has a consumeAction instance variable that gets removed from the allowableActions list once it is used which follows ISP.
    - To allow the player to use fireAttack, fire flower has a FireAttack ItemEffect enum, giving the player this ability as the fireFlower is added to the inventory. And to restrict its effects to 20 turns, a tick method is overridden to reduce the age of the fire flower every turn, once the age is 0, it would be removed from the player's inventory.
    - Reduction of connascence was also used by adding MAX_AGE (20) as a constant, which is only a naming connascence.
    - Preconditions: an Item object
    - Postconditions: a Fire Flower object
- Abstract AttackAction
    - This is changed from the previous design where it is a concrete class to reduce DRY in attack type actions. Methods like dropItems(), variables like target and direction can be added into this abstract class instead of repeating, following DRY and DIP.
    - The usage of AttackAction before is now put into the subclass GenAttackAction that deals with the basic attack actions that simply uses a weapon to attack the target.
    - SRP is also followed where the Power Star status which gives the actor essentially an invincible status, could be checked in the invincibleActor() method in this super class rather than in execute().
    - Preconditions: an Action object
    - Postconditions: an AttackAction object
- FireAttackAction
    - This extends AttackAction and has a dependency to Fire as it drops a fire on the target's location as the attack. It also would still drop a fire on the target's location if the attacker has Power Star but the target would simply be killed right away.
    - To allow the player to attack with fire, the enemy class would simply check for the player's FIRE_ATTACK capability and return this action.
    - Preconditions: an AttackAction object
    - Postconditions: a FireAttackAction object
- Fire
    - This class extends abstract Item ,a non-portable item, as it doesn't require the need to reset the ground to its previous ground once extinguished. The extinguishment of fire after 3 turns is implemented in an overridden tick() where an life counter is increased every turn and once it reaches the MAX_LIFETIME (3), it would be removed from the ground.
    - To hurt the actor on its ground, it would use location.getActor().hurt(Fire.FIRE_DAMAGE).
    - The implementation of MAX_LIFETIME and FIRE_DAMAGE as Fire class constants reduces connascence to only naming.
    - Preconditions: an Item object
    - Postconditions: a Fire object

REQ5:


edu.monash.fit2099.engine
actors
<<abstract>>
Actor
actions
<<abstract>>
Action
Extends
game
Extends
Extends
actors
enemies
Bowser
Extends
<<abstract>>
Enemy
1
Extends
Extends
Goomba
Extends
Koopa
Extends
PiranhaPlant
FlyingKoopa
friendlies
Extends
<<abstract>>
Friendly
1
PrincessPeach
Extends
acts
Toad
acts
actions
SpeakAction
behaviours
1
SpeakBehaviour
1
1
acted by
<<interface>>
Behaviour
monologues
Monologue
1
<<interface>>
Speakable

Class Contracts

Monologue

Pre-conditions

- A sentence is passed to the constructor.

Post-conditions

- Represents a specific monologue an actor can speak.

Speakable

Pre-conditions

- Any speaking actors implement this interface.

Post-conditions

- Will return a collection of monologues the actor may speak.

SpeakAction

Pre-conditions

- Is given a collection of monologues.

Post-conditions

- Will speak a random monologue to the display.

Design Version 1.1

The actors who speak randomly will have a dependency with SpeakBehaviour. It is assumed that the actors should speak every second turn (stated in the requirement), which will prevent them from performing any other actions in that turn.

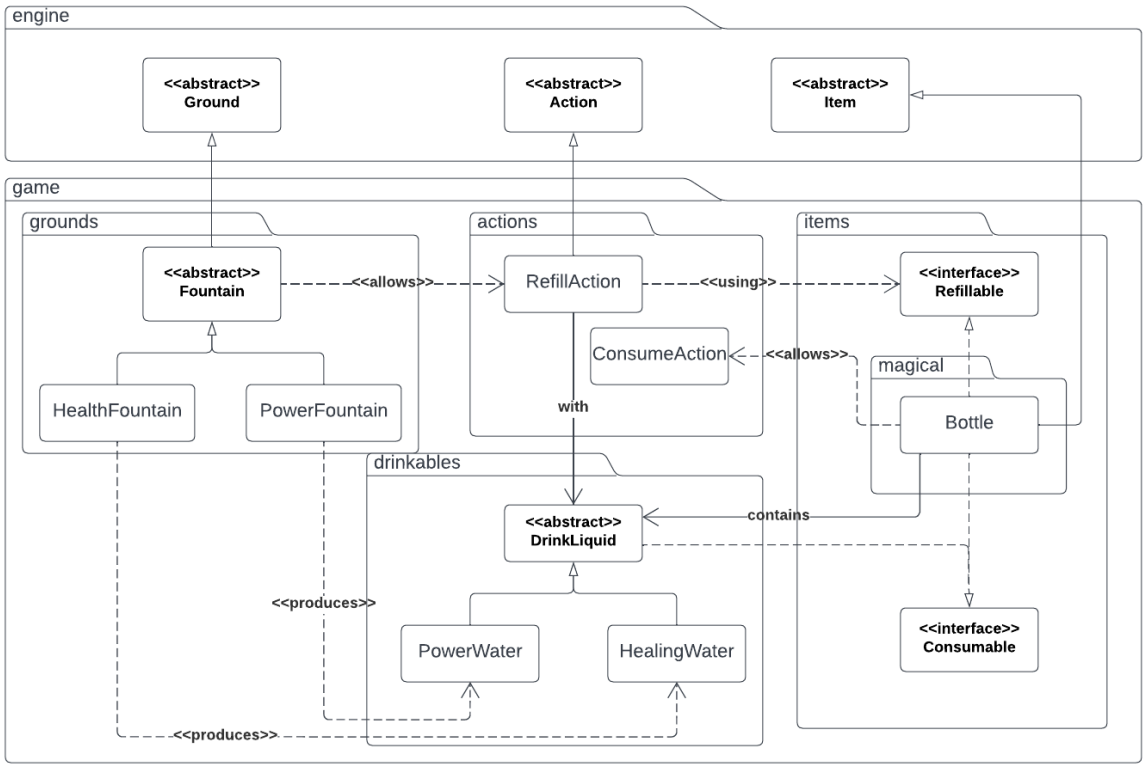
Design Version 1.0

Several actors are required to speak randomly to the console. The existing functionality for speaking is that every actor who speaks implements the Speakable interface, which has a method which returns a SpeakAction. It is common among all speaking actors that one of a collection of sentences is randomly chosen. To account for the primitive obsession code smell, each monologue sentence will be refactored into a Monologue class. The Speakable interface will then have a method to return the collection of monologues the actor can speak. The SpeakAction class will be refactored to change the primitive parameter to a Monologue object parameter (part of code smell solution), and randomly choose a Monologue to speak. Again, since sentences are randomly spoken by all actors, factoring this random selection code into the SpeakAction class follows the DRY principle. This better follows SRP and OCP

as well since any new speaking actors have less work to do; they only need to return a Monologue collection.

Rationale

REQ3:



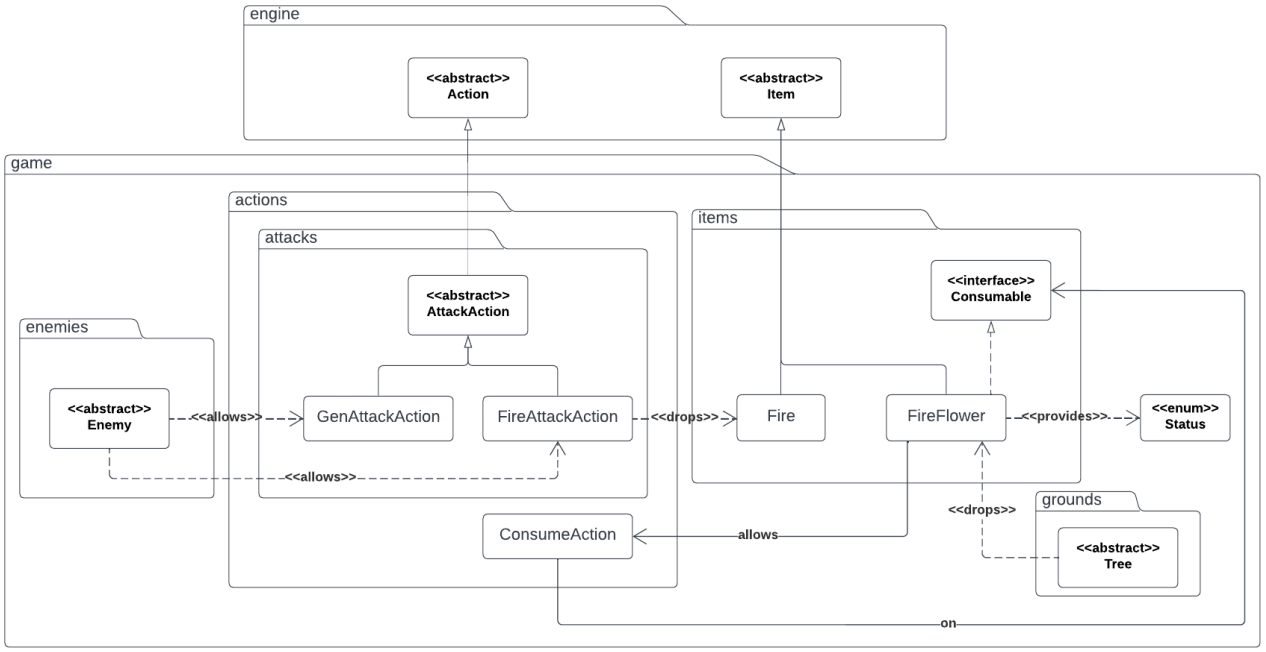
- <u>Bottle</u>
    - The bottle is implemented as a concrete class extending from Item abstract class. As it can also be consumed and refilled it implements the Consumable interface and the refill interface. This follows ISP as the bottle's different usage is separated into different interfaces.
    - The bottle has a stack of DrinkLiquids (at this stage, this is just water) that can be filled up by overriding the refill() method in the Refillable interface and be consumed by overriding consume() in the Consumable interface. This follows Open-Closed Principle as no modifications were needed in the interfaces to implement these features for Bottle. As the bottle is a magical item, it was given the ItemEffect enum STORE_LIQUID. In the future the bottle could potentially store more liquids so this capability could come in handy.
    - Preconditions: an Item object
    - Postconditions: a Bottle object
- <u>Refillable Interface</u>
    - This interface has an abstract method refill() that allows any class that implements it to be a refillable item that can be refilled in their own ways. This interface allows future extensibility as there maybe more refillable items or items that can be recharged for example a saber that needs to be recharged with electricity.
    - Preconditions: None
    - Postconditions: a Refillable object

- Abstract DrinkLiquid
    - This abstract class implements the consumable interface as an actor can drink water, following ISP. It is abstract as water itself is not instantiated and it encapsulates the similarity of its subclasses like water implementing consumable and toString(), following OOP and abstraction.
    - This abstraction also allows Open-Closed for potential future liquids like poison and DIP as its subclasses are inheriting from an abstract class.
    - Preconditions: None
    - Postconditions: a DrinkLiquid object
- Healing Water
    - This class extends DrinkLiquid, following DIP and has an instance variable HP_HEAL as a constant. This reduces the connascence to only name connascence when using it in the overridden consume() to heal the actor by 50 points.
    - Preconditions: a DrinkLiquid object
    - Postconditions: a Healing Water object
- Power Water
    - This class extends DrinkLiquid, following DIP and overrides the consume() from the Consumable interface to give an actor a POWER_UP Status enum. In the actor's overridden getIntrinsicWeapon() would simply increase the base attack damage. This allows good encapsulation as Power Water is not accessing the actor's intrinsic weapon but simply providing a buff.
    - Preconditions: a DrinkLiquid object
    - Postconditions: a Power Water object
- Abstract Fountain
    - This abstract class extends Ground rather than the item the actor is consuming the water it provides, not the fountain itself. To allow the player to get water from the fountain when it stands on it, the allowableActions() method would check if the player has a means to store the liquid and if the actor is on the fountain before adding a RefillAction at the fountain.
    - To get the special liquids from each fountain, an abstract getWater method is provided in Fountain so that its subclasses can override and return whatever drink liquids they like, therefore following DIP and OCP.
    - Preconditions: a Ground object
    - Postconditions: a Fountain object
- Power Fountain
    - This class extends abstract Fountain, following DIP and overrides the getWater method to return new PowerWater every time it's called.
    - Preconditions: a Fountain object
    - Postconditions: a Power Fountain object
- Health Fountain
    - This class extends abstract Fountain, following DIP and overrides the getWater method to return new HealingWater every time it's called.
    - Preconditions: a Fountain object
    - Postconditions: a Health Fountain object
- RefillAction
    - This class extends Action as the player needs to be able to execute it using the menu. It has an instance variable called liquid of type DrinkLiquid that is

initialised through the getWater method of Fountain which is passed in the constructor.

- To refill the bottle, the execute() method would obtain the refillable item from the actor's storage and call its refill() method. While dependency injection could have been implemented by adding the getStorage method in fountain and passing that into RefillAction constructor, however, it would violate SRP in Fountain as it shouldn't know about where or not an actor has a storage item or not, it is simply a ground object.
- Preconditions: actor has a liquid storage, actor is on a fountain
- Postconditions: actor's liquid storage is refilled

REQ4:



- <u>FireFlower</u>
  - This class extends Item as it makes more sense for it to be an item as it allows multiple fire flowers to be at one location and avoids resetting the ground to its previous state when fire flower is removed.
  - The fire flower 50% chance of being created in every tree growth stage is implemented by adding this to a grow() method in the tree class that can be overridden in sprout and sapling to grow into other tree phases and call the super.grow() to grow fire flowers. This not only improves the previous design by increasing SRP as growing trees has its own method, but also allows abstraction as repetition and dependency on the fire flower class is reduced to only in the tree class.
  - To consume the fire flower, it implements the consumable interface and has a consumeAction instance variable that gets removed from the allowableActions list once it is used which follows ISP.

    - To allow the player to use fireAttack, fire flower has a FireAttack ItemEffect enum, giving the player this ability as the fireFlower is added to the inventory. And to restrict its effects to 20 turns, a tick method is overridden to reduce the age of the fire flower every turn, once the age is 0, it would be removed from the player's inventory.
    - Reduction of connascence was also used by adding MAX_AGE (20) as a constant, which is only a naming connascence.
    - Preconditions: an Item object
    - Postconditions: a Fire Flower object
- Abstract AttackAction
    - This is changed from the previous design where it is a concrete class to reduce DRY in attack type actions. Methods like dropItems(), variables like target and direction can be added into this abstract class instead of repeating, following DRY and DIP.
    - The usage of AttackAction before is now put into the subclass GenAttackAction that deals with the basic attack actions that simply uses a weapon to attack the target.
    - SRP is also followed where the Power Star status which gives the actor essentially an invincible status, could be checked in the invincibleActor() method in this super class rather than in execute().
    - Preconditions: an Action object
    - Postconditions: an AttackAction object
- FireAttackAction
    - This extends AttackAction and has a dependency to Fire as it drops a fire on the target's location as the attack. It also would still drop a fire on the target's location if the attacker has Power Star but the target would simply be killed right away.
    - To allow the player to attack with fire, the enemy class would simply check for the player's FIRE_ATTACK capability and return this action.
    - Preconditions: an AttackAction object
    - Postconditions: a FireAttackAction object
- Fire
    - This class extends abstract Item ,a non-portable item, as it doesn't require the need to reset the ground to its previous ground once extinguished. The extinguishment of fire after 3 turns is implemented in an overridden tick() where an life counter is increased every turn and once it reaches the MAX_LIFETIME (3), it would be removed from the ground.
    - To hurt the actor on its ground, it would use location.getActor().hurt(Fire.FIRE_DAMAGE).
    - The implementation of MAX_LIFETIME and FIRE_DAMAGE as Fire class constants reduces connascence to only naming.
    - Preconditions: an Item object
    - Postconditions: a Fire object