Assignment 3 – Lab 3 Team 1

Work Breakdown Agreement

Members:

1. Yong Jun Ong 31861407
2. Chen Yang Seah 32311168
3. Yuki Yi Wei Wong 32599862

Task Delegation:

- Chen Yang: Creative mode - REQ 4, REQ 5
- Yong Jun: REQ 1 Teleportation part, REQ 3
- Yuki: REQ 2, DamageGround

Signatures:

1. I, Yong Jun, accept this WBA.
2. I, Yuki, accept this WBA.
3. I, Chen Yang, accept this WBA.

Requirement 1

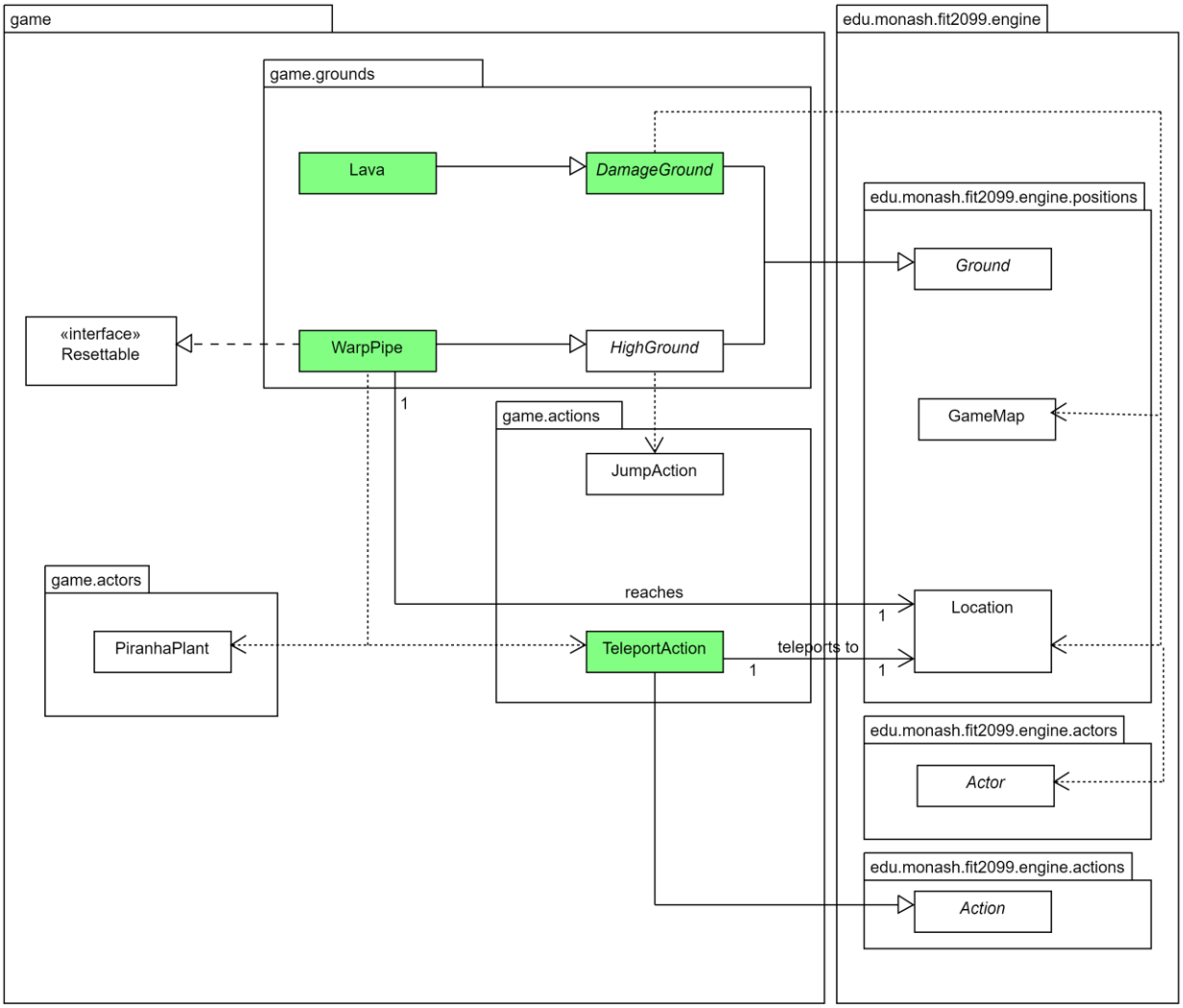
Design Rationale

The DamageGround abstract class is extended from the Ground abstract class. It implements the damage mechanisms the ground does to actors on it. This will form dependency relationships with the classes GameMap, Location and Actor, as those classes are required to hurt the Actor on the ground it there is one and drop its items as well as remove the actor if the damage kills the actor. The Lava class, a subclass of DamageGround, extends from DamageGround class to inherit the common features of all Damage Grounds like the damage mechanism. The Lava class will then specify its damage numbers, as well as add on specific features like not allowing enemies to enter. This adheres to the SRP and OCP, where each class is responsible for its own unique features and the DamageGround class is open to be extended by more damage grounds in the future, without the need to modify the DamageGround parent class. The DamageGround class is also extended by another subclass, Fire class in Requirement 2.

The WarpPipe class extends from HighGround class as it is a high ground itself. This will allow it to inherit all common features of high grounds, like being able to be jumped on when actors stand beside it. In this case, we set the success rate of all jumps onto a WarpPipe to be at 100%, as it is not specified in the brief. Each WarpPipe has one destination it allows actors to teleport to, which forms an association relationship with Location class due to the destination set for each WarpPipe. A WarpPipe also spawns Piranha Plants (REQ 2) and allows actors standing on it to teleport to its set destination. This forms dependency relationships between the WarpPipe class and PiranhaPlant class as well as TeleportAction. The WarpPipe class also implements the interface Resettable as it is required to spawn a new Piranha Plant if appropriate or increase the base attack of the existing one on it in the case of a reset occurring.

The TeleportAction class extends from Action abstract class to inherit the common features of all Actions, such that it can apply its own set of unique features. It has an association relationship with the Location class as it is required to move a certain actor from one location to another across game maps. An association relationship with GameMap class is not required as we can get the map of a location instance through its map accessor method.

UML Diagram


game
edu.monash.fit2099.engine
game.grounds
Lava
DamageGround
edu.monash.fit2099.engine.positions
Ground
«interface»
Resettable
WarpPipe
HighGround
GameMap
1
game.actions
JumpAction
game.actors
reaches
1
Location
PiranhaPlant
TeleportAction
teleports to
1
1
edu.monash.fit2099.engine.actors
Actor
edu.monash.fit2099.engine.actions
Action

Requirement 2

Design Rationale

Enemies

The Enemy abstract class now extends from the HostileActor abstract class, which then extends from the Actor class. This new HostileActor class is created to allow the actors' base attack (Intrinsic Weapon Damage) to be modified (Required Feature in Requirement 3).

Princess Peach

I created a Key class that extended from the Item class and inherited all of its features, allowing for customization to cater to Item's unique features. Once Bowser is defeated, this key object is dropped on the ground for the player to pick up and use to unlock the handcuffs. This is accomplished through the usage of capabilities in the Bowser class.

The UnlockAction class is extended from the Action class to inherit all its features to allow for modification to cater to UnlockAction's exclusive features. It also has an association relationship with the Actor class as the action needs to be performed to the target actor.

The Toad and Princess Peach classes are extended from the abstract ally class through polymorphism, inheriting the basic features of the ally class as they share the same behaviours whereby they are friendly actors that do not move, attack and follow the player. Princess Peach has dependency relationships with the UnlockAction class as it adds an unlock action option to its allowable actions in the case of the player having the capability of having the key. This complies with the SRP because each actor is responsible for their own specifications (Toad has TradingAction and SpeakAction with player, while PrincessPeach has UnlockAction with player), and there is no need to modify the parent class in the case of a new subclass creation.

Bowser

Bowser was added as a subclass of enemy because it is another type of enemy that will attack the player whenever the player is in attack range. WanderBehaviour has been removed from the list because it cannot move.

As mentioned in REQ1, an abstract class DamageGround inherited from the abstract Ground class is implemented for the grounds that damage actors on the ground, in order to avoid repeated code that violates the DRY principle. According to REQ2, whenever Bowser attacks, a fire will be dropped on the ground that lasts for three turns. As a result, I implemented fire as a subclass of this abstract class because fire is a ground that can harm any actor. Although both fire and lava share similar damage ground features, they cause different damage, and fire has a limited duration of three turns as opposed to lava, which can stay there indefinitely, making them implement as different subclasses which obey SPR.

Because resetting the game returns Bowser to his original position, it has an association relationship with Location, as I used location as an instance variable in the Bowser class to reset to his original

location. After resetting, it will remain still until the player is within Bowser's attack range, which can be accomplished by clearing all the behaviour from the list with clear() and then adding the behaviour back in when the player is in the attack range.

Piranha Plant

I implemented PiranhaPlant as a subclass of enemy as it is another form of enemy. As it is spawned by WarmPipe, it has a dependency relationship with WarmPipe. As it can't move around and follow the player who attacks it, I have removed these two behaviours (WanderBehaviour & FollowBehaviour) from the list, so it now only has a dependency relationship with AttackBehaviour.

Resetting will increase alive/existing Piranha Plants hit points by an additional 50 hit points, this can be done by resetting the max hp of this enemy with resetMaxHp().

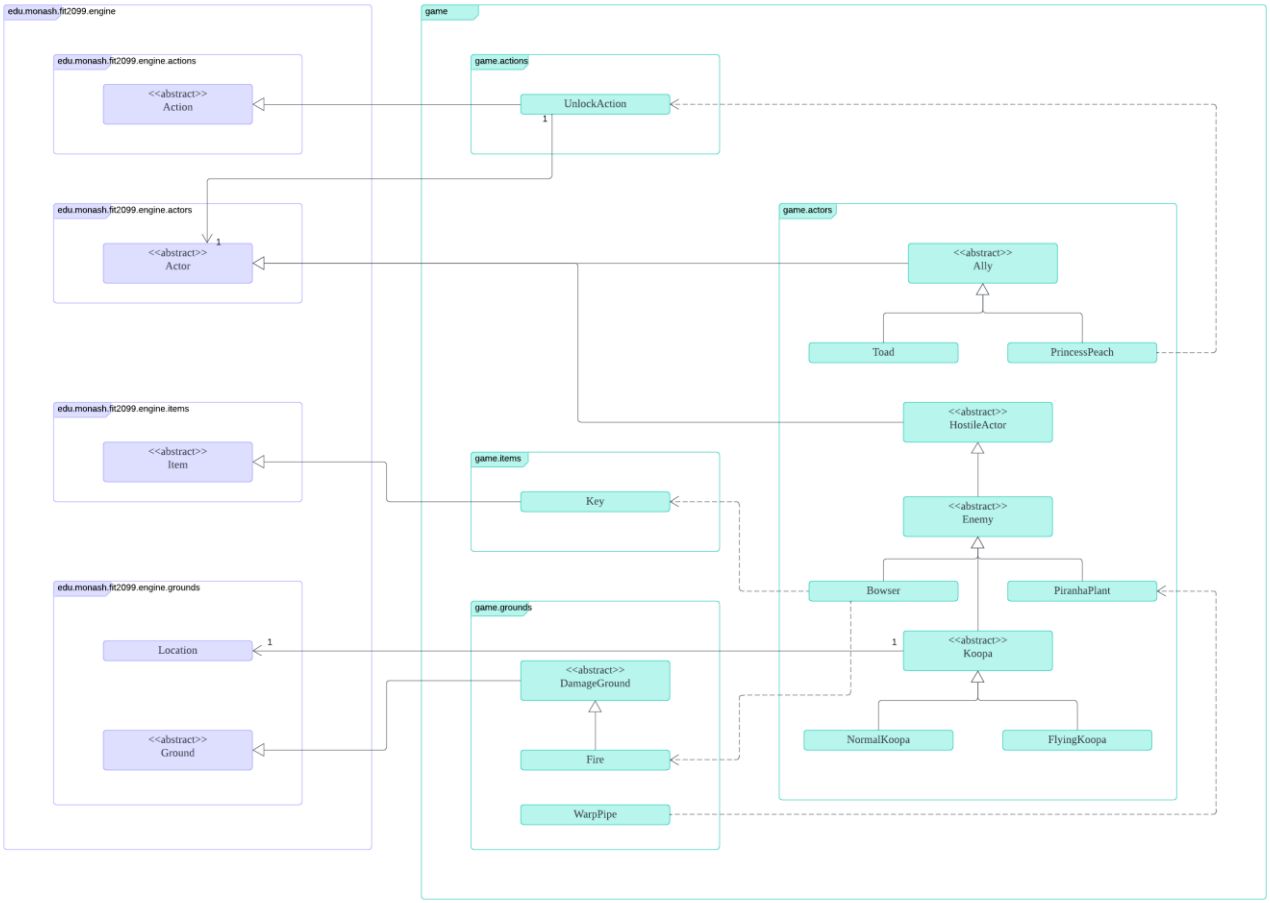
Flying Koopa

I first implemented an abstract koopa class, as both Normal Koopa and Flying Koopa share the same behaviours whereby they are enemies that will be in a dormant state whenever it is unconscious until the player brings a wrench to crack the shell. After cracking, a Super Mushroom is dropped. The mature tree has a 50:50 chance of spawning either normal Koopa or Flying Koopa (after a successful 15% spawn rate), making it to have a dependency relationship with mature tree.

These two classes are extended from the abstract koopa class through polymorphism, inheriting the basic features of a Koopa class. This obeys the SRP and OCP as the Koopa class is only responsible for the basic features of a koopa and any subclasses extended from it will not require modifications towards the Koopa class.

As Flying Koopa has a distinguishing feature of flying over the trees and walls from Normal Koopa, they are formed as two subclasses. Flying Koopa can fly over these grounds as now I have enabled it to enter these grounds through the use of capabilities in the form of Status enum in canActorEnter().

UML Diagram


edu.monash.fit2099.engine
edu.monash.fit2099.engine.actions
<<abstract>>
Action
edu.monash.fit2099.engine.actors
<<abstract>>
Actor
edu.monash.fit2099.engine.items
<<abstract>>
Item
edu.monash.fit2099.engine.grounds
Location
<<abstract>>
Ground
game
game.actions
UnlockAction
1
game.actors
<<abstract>>
Ally
Toad
PrincessPeach
<<abstract>>
HostileActor
<<abstract>>
Enemy
Bowser
PiranhaPlant
1
<<abstract>>
Koopa
NormalKoopa
FlyingKoopa
game.items
Key
game.grounds
<<abstract>>
DamageGround
Fire
WarpPipe
1
1

Requirement 3

Design Rationale

A new abstract Fountain class is created, which extends from the Ground class. This class implements the common feature of all fountains, which is to allow for refills, and creates an abstract method to provide water, which must be implemented by the class' subclasses to define what specific water the fountain provides, forming dependency relationships with the Water class and RefillAction class. Two subclasses of fountains, which are PowerFountain and HealthFountain, will both inherit the common features of fountains, and define the water they provide through implementing the abstract method defined in their parent class. This adheres to the SRP, where each subclass is responsible to define its own specific set of features. This also adheres to LSP, as any subclass of the Fountain abstract class is accepted in inputs that expect a fountain instance as an input, like RefillAction's constructor. OCP is also adhered as more fountains can be created without the need to modify the fountain abstract class.

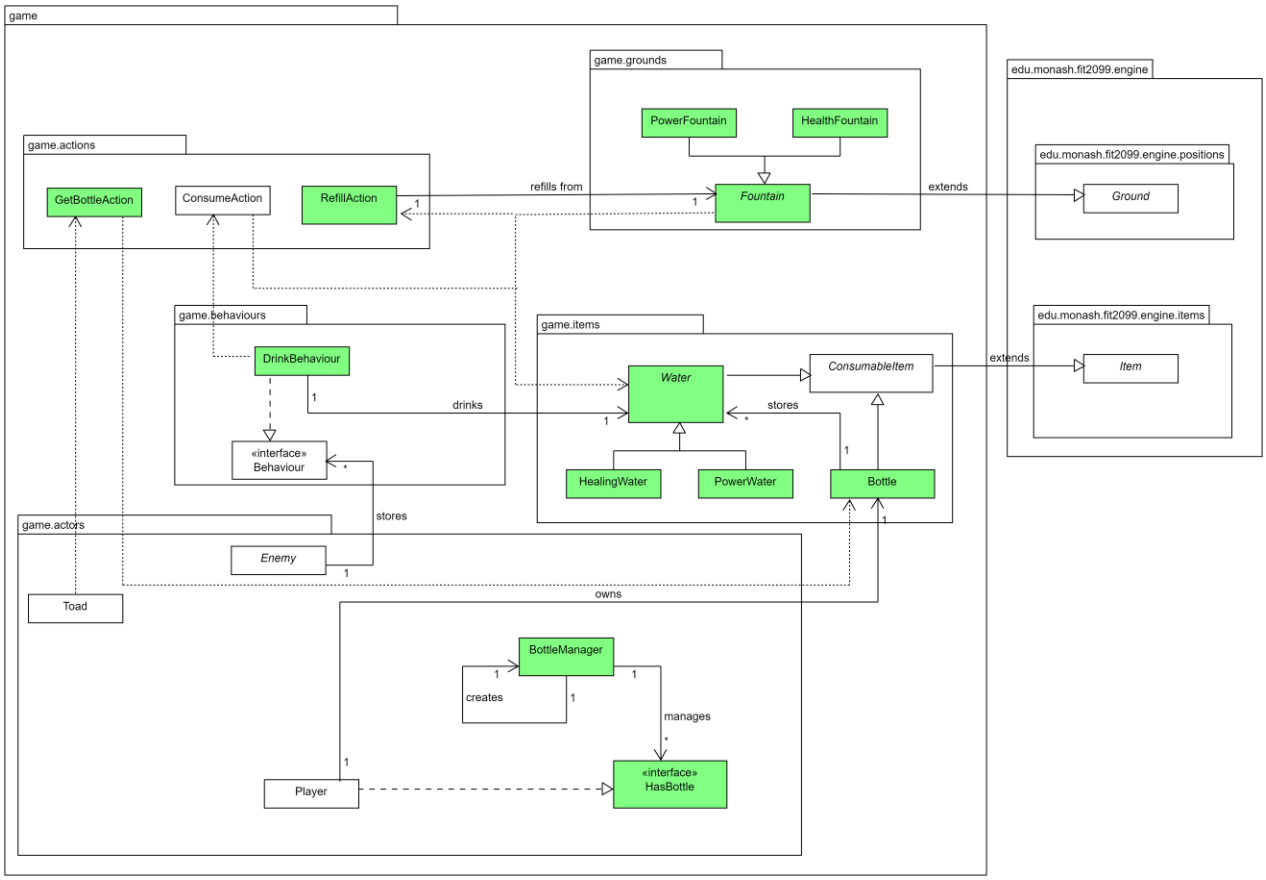
The RefillAction class has an association relationship with the Fountain class, as it needs to know which specific fountain it is refilling from, such that it can deplete that fountain's water storage. There also exists a GetBottleAction, to allow players to retrieve a bottle from Toad. This forms dependency relationships between Toad class and GetBottleAction class, as well as GetBottleAction class and Bottle class. ConsumeAction class now has a dependency relationship with Water class as it requires the knowledge of what water is next to be consumed by the player in an event of a player consuming his/her bottle's water storage. For enemies to drink from fountains whenever appropriate, a new DrinkBehaviour class is created, implementing the Behaviour interface, as the Enemy class has a hash map of behaviours. The DrinkBehaviour class has an association relationship with the Water class as it requires knowledge of what water it is drinking. To allow enemies to consume water, the DrinkBehaviour class will need to return a ConsumeAction instance, which forms a dependency between the DrinkBehaviour class and ConsumeAction class.

Both Bottle class and Water abstract class are extended from the ConsumableItem class, as those items are to be allowed for consumption, which eliminates repeated code to allow items to be consumed. The Bottle class has an association relationship with the water class as it stores a stack of Water instances as its instance variable. The Water abstract class is formed to implement common features of all water types, like being able to be stored in a bottle and unable to be picked up. Its subclasses, HealingWater class and PowerFountain class will then implement their own respective properties and features. This adheres to SRP and LSP as each class is responsible for their own unique features and all subclasses of Water can be appropriately stored in a bottle, where Water instances are expected. This also adheres to OCP as more water types can be easily created without the need to modify the Water abstract class.

For players to each own a bottle, the Player class will have an instance variable of Bottle type as each of them have their own bottle. This forms an association relationship between Player class and Bottle class. The Player class will also implement the HasBottle interface, such that they can have access to methods that make their bottles function. There will also be a singleton class called BottleManager to manage all instances that implement the HasBottle interface, such that specific methods can be used on the instances' bottles.

Regarding the drinking behaviour, an enemy will wander around until it lands on a fountain, then it stays on the fountain to drink until there are other actors that are hostile to it are in contact. Then, they will try to attack and follow its target, leaving the fountain.

UML Diagram



Requirement 4

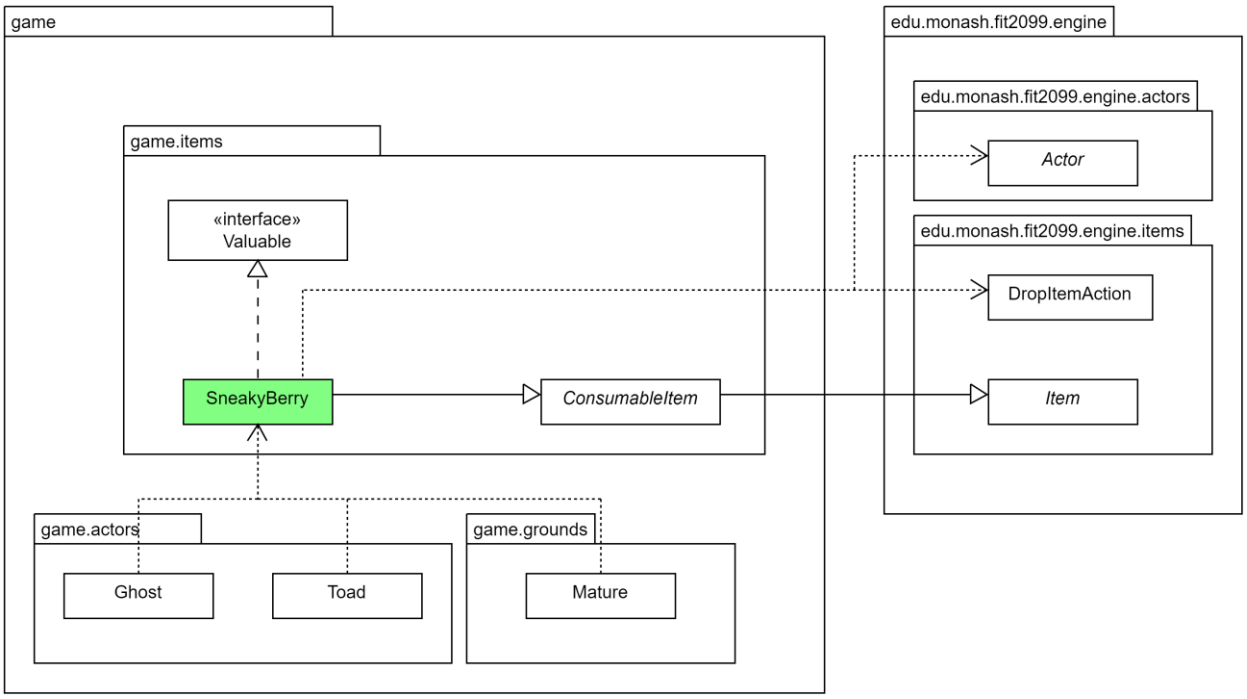
Design Rationale

The SneakyBerry class extends from the ConsumableItem abstract class, inheriting all common features of consumable items, like allowing the action of consumption by an actor. As it is a magical item that holds value and can be acquired from the Toad actor, it also implements the Valuable interface. The SneakyBerry's distinct buff and duration of the buff would be implemented within the SneakyBerry class in compliance with the single responsibility principle (SRP). The Open-closed principle (OCP) is adhered here, where the SneakyBerry class can be added without modifying the parent class it extends from, which is the ConsumableItem class. The SneakyBerry class forms dependency relationships with the Actor and DropItemAction classes as it has a distinct feature, where it can be dropped by Ghosts when they are defeated, instead of not being able to be dropped at all, thus those classes are required to implement that feature.

The Ghost, Toad and Mature classes all have dependency relationships with the SneakyBerry class due to many reasons. All Ghost actors will have a SneakyBerry in their inventory such that it will be dropped when they are dead. Besides, the Toad actor needs a new SneakyBerry instance to allow it to be traded with the Player actors. Moreover, Mature trees have a super rare chance to spawn a new SneakyBerry instance when they wither.

Since the SneakyBerry's lifespan will be similar to that of PowerStar, therefore it will use the same implementation as before.

UML Diagram



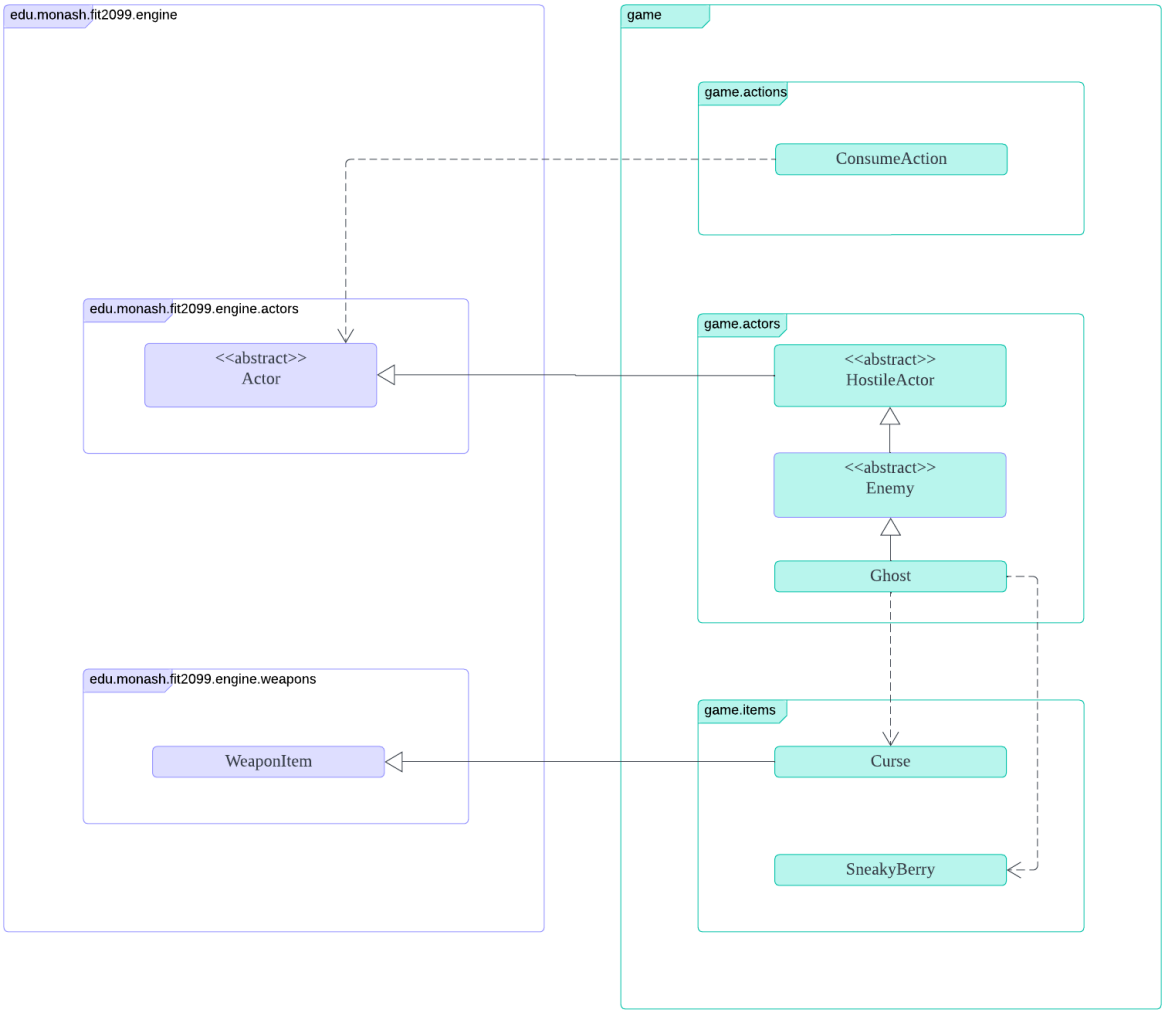
Requirement 5

Design Rationale

The Ghost class is created as a subclass of the Enemy abstract class because it is another type of enemy that will attack Player actors whenever they are in range. This allows the Ghost class to inherit all common features of all Enemy actors, like their behaviours and more. To implement the unique attack mechanism of Ghost actors that curse Player actors for 3 rounds, the Curse class is created as a subclass from WeaponItem, which will the weapon Ghost actors use to apply an effect on the target actor which will cause damage to the target over the next few rounds. As a Curse instance is added to the inventory of every newly created Ghost actor in its constructor, this forms a dependency relationship between the Ghost class and the Curse class. A SneakyBerry is added to every newly created Ghost actor through its constructor, such that it will drop the said item after it has been defeated. This forms a dependency relationship between the Ghost class and the SneakyBerry class. The Ghost class extending from the Enemy class shows how OCP is adhered here, where a new enemy can be added without the need to modify the Enemy parent class. The Ghost and Curse classes also adhere to SRP, where they are solely responsible for their distinct and unique features and specifications.

To remove the cursed status on a Player actor, the ConsumeAction will remove the status from the actor consuming whenever the consume action involves healing water, which forms a dependency relationship between ConsumeAction class and Actor class.

UML Diagram


edu.monash.fit2099.engine
game
game.actions
ConsumeAction
edu.monash.fit2099.engine.actors
<<abstract>>
Actor
game.actors
<<abstract>>
HostileActor
<<abstract>>
Enemy
Ghost
edu.monash.fit2099.engine.weapons
WeaponItem
game.items
Curse
SneakyBerry

Sequence Diagram for RefillAction.execute()