FIT2099 ASSIGNMENT 3

Update on the Design Rationale

Lab 10 Team 2: Sok Huot Ear, Satya Jhaveri, Klarissa Jutivannadevi

Emails: sear0002@student.monash.edu, sjha0002@student.monash.edu, kjut0001@student.monash.edu

In this assignment, our group created a UML diagram that we plan to implement for our second assignment. The main purpose of creating this UML diagram is to achieve the three core design principles, which in this assignment focuses on two out of three principles. The first principle – ***Don't repeat yourself*** – is done by creating new required interfaces and classes (both abstract and not abstract) in order to prevent us from rewriting the same code for some similar methods that might be used during the game. The second principle – ***Classes should be responsible for their own properties*** – is achieved by creating the classes that are more suitable for the methods. For example, instead of having method C1() in Class A and method C2() in Class B, a new class which is Class C can be created to contain both C1() and C2().

In this assignment, we aim to uphold the SOLID principles, which are the guidelines to achieve a neater and more structured application for more efficiency. The first principle, ***Single Responsibility Principle (SRP)***, believes that '*a class should have one, and only one, reason to change'*. Any classes should only have one responsibility containing all the functionalities needed to assist that certain responsibility. This works by separating every part of code to atomic functions that cannot be broken down into smaller tasks.

The second principle, ***Open Closed Principle (OCP)***, proposed the idea that '*software entities should be open for extension but closed for modification'*. This means that the base code that has been implemented previously should never be modified, but additional functionalities can be added using abstraction to achieve the desired output. The third principle, ***Liskov***

Substitution Principle (LSP), explains that '*derived classes must be substitutable for their base class'*. Consistency is the main focus in LSP. For this principle, a child class should have the same behaviour as the parent class, where the user can expect the same type of result from the same input given. To check, LSP is obeyed when the overridden method from the child class still has the behaviour from the same method of that parent class.

The fourth principle, ***Interface Segregation Principle (ISP)***, stated that '*many client-specific interfaces are better than one general-purpose interface'*. This means that a class that does not require a certain method should not be forced to have the method declared in its class just to eliminate error. To achieve this, multiple interfaces having a minimum method is preferable than only having less interfaces with many methods and having to implement them when not all are needed in a certain class. And finally, the fifth principle, ***Dependency Inversion Principle (DIP)***, mentioned that '*high-level modules should not depend on low-level modules. Both should depend on abstraction'* and '*abstractions should not depend on details. Details should depend on abstraction'*. This can be achieved by putting an abstraction layer to limit the need of modifying the system. For example, low level modules reference an interface and that same interface will be inherited from the higher level modules instead of lower level modules references to the higher level modules.

Notes

Our group uses red, yellow, and green as the colour code as an indication to which classes are added and modified.

Green : Classes are not provided and were added in order to create a better program.

Yellow : Classes were provided in the base code, but some modifications are needed such as adding methods and/or attributes to improve the program.

Red : Classes are already provided and no modifications are needed. While designing the application, we also make sure that the engine is left untouched.

This report will cover the requirements for the third assignment (Further Design and Implementation) and will proceed to the previous assignments, which include some minor changes based on the code given.

DESIGN RATIONALE

A3 REQ 1: Lava Zone

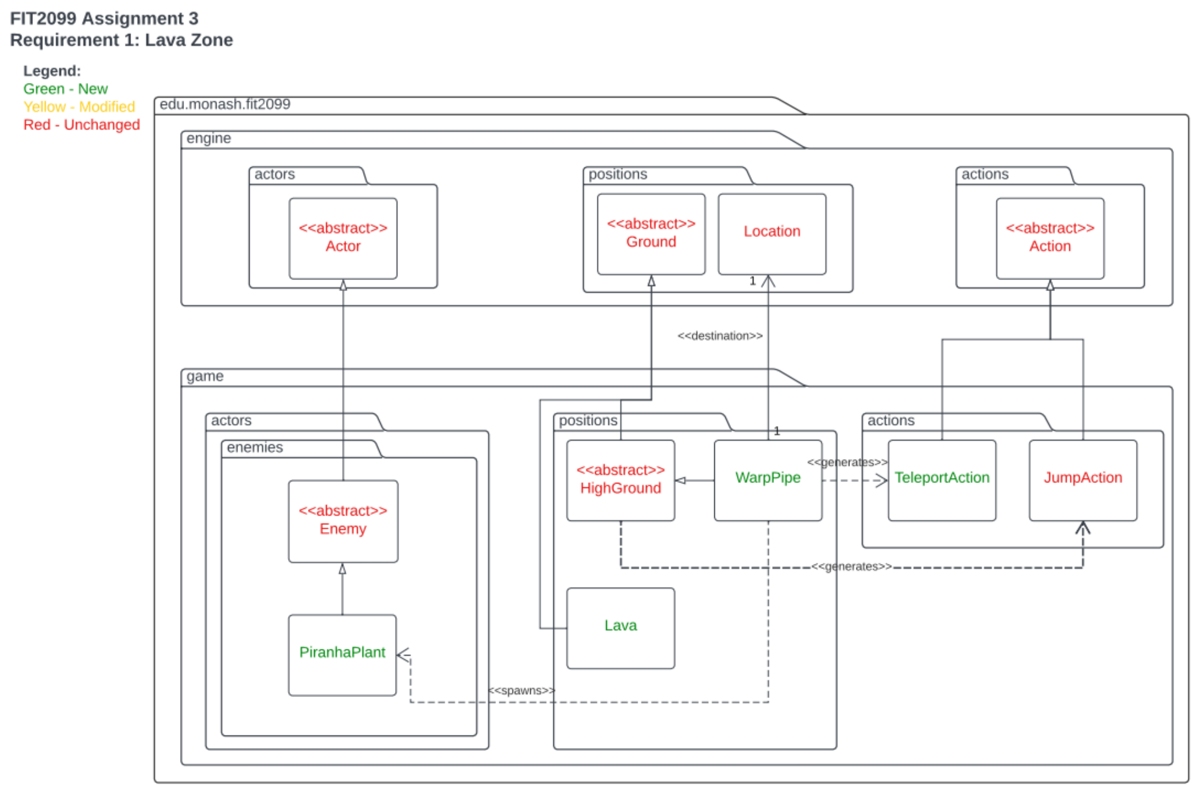

Diagram 1.1 Lava Zone UML Class Diagram

The first requirement requires a new map and warp pipe for the player to teleport. Firstly, a new map is created similar to how the original map was created, inside the main method of the Application class.

The new map has Lava, which is another object in the game that Actors can walk on, hence, the class Lava becomes a child class of Ground class. The Lava class does not allow Enemies to step on it, which is represented by the canActorEnter method of the Lava class, which returns true only if the Actor does not have the Status ‘cant_enter_lava’. This Status is given to Enemies, and allows the Lava class to prevent Enemies from entering it, without needing a direct dependency from Lava to the Enemy class. It is also stated that every turn that an Actor is on a lava, the Actor will take 15 damage, this is executed in the tick method. Since tick is executed every turn, it will always check if an Actor is standing on it, and if they are, damage will be done.

A WarpPipe class, which is a child class of HighGround is created to show that the player will need to jump in order to be able to stand in the WarpPipe and teleport. Since no jump chance was specified, it was assumed that Actors would be able to jump with a 100% success rate. Once an Actor is standing on the WarpPipe, a new action, TeleportAction, will be returned by the WarpPipes getAllowableActions method. This will trigger an action to teleport which moves the Actor to another map if executed. In the TeleportAction class, the execute method will implement the functionality of teleporting to another WarpPipe in another map, as well as linking the two WarpPipes, so that re-entering the WarpPipe that the Actor comes out of leads back to the Pipe that they went into.

There is also a dependency between WarpPipe and PiranhaPlant. This is done as PiranhaPlants are spawned on top of WarpPipes in the second round of the game, implemented via the WarpPipe’s tick method. Before the player can use the WarpPipe, it will need to attack the PiranhaPlant until it dies(which will be explained in REQ2).

Dependency Injection is used when creating the **TeleportAction** class. This is done by including an instance of Location as the parameter of the constructor of the TeleportAction class. The benefit of using this is that code becomes more reusable and testing becomes easier. For example, when a new map is to be created, we can just create a dummy location to check how the implementation will go, instead of needing to create a complete class first and then testing it.

A3 REQ 2: More Allies and Enemies

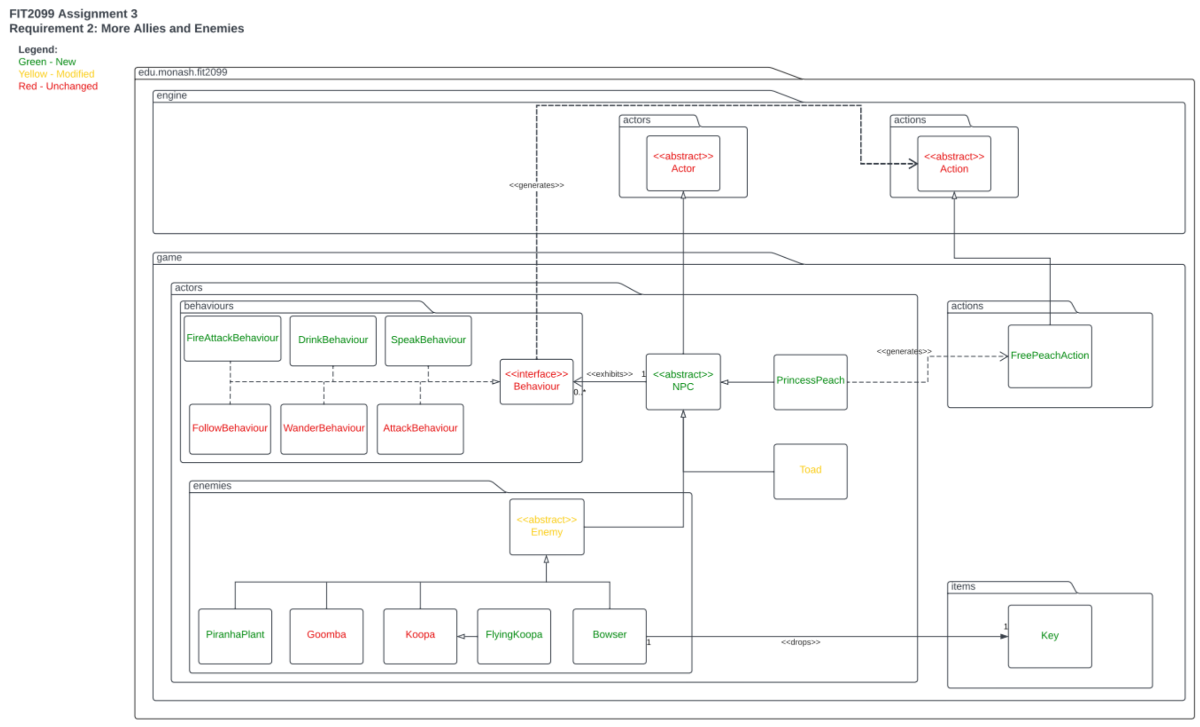

Diagram 1.2 More Allies and Enemies UML Class Diagram

For REQ2, various actors are created and an abstract NPC class is also created (will be explained in REQ5). Based on the requirements, there are 3 additional actors (Bowser, Piranha Plant and Flying Koopa) that can attack the player and 1 other actor (Peach). Hence, the 3 actors become a child class of the abstract Enemy class, where it has dependencies with many allowable behaviours.

Bowser class has an association with the Key class, as it has the key to unlock Princess Peach in its inventory. Key extends the Item class, as this way, Bowser will drop the Key when it is killed without any extra code, as Bowser's inventory will be dropped on the Ground where Bowser is when he dies.

Bowser initially has only FireAttackBehavior (which only creates actions if a 'hostile to enemy' Actor is near), so it initially does not move, but when a player is near, it gains FollowBehaviour in order to follow the Player. When it gets attacked by a player and dies, the key will be dropped for the player to pick up. The Key has a capability 'can_free_peach', which allows the holder to unlock PrincessPeach. The reason this capability is used is to remove the need for a direct dependency from PrincessPeach to Key. Furthermore, the Bowser class also overrides the resetInstance method in order to heal Bowser and return Bowser to its original position.

Bowser is given the new behaviour FireAttackBehaviour, which is similar to AttackBehaviour, only instead of generating AttackActions, it generates FireAttackActions. Hence, Bowser can attack the player using FireAttackAction.

Bowser overrides the Enemy class' reset method, as instead of being killed and removed from the map upon resetting, Bowser is healed and moved back to its original position. To achieve this, a Location attribute 'initialPosition' is given to Bowser, and this is used to move Bowser back to his original location.

A PrincessPeach class is created to create an instance of Princess Peach. Since PrincessPeach cannot move, attack, or be attacked, it is given no Behaviours. However, its allowableActions method can generate a FreePeachAction if the other Actor has the status 'can_free_peach', hence the dependency between PrincessPeach and FreePeachAction.

The PiranhaPlant is designed to be able to attack but not move anywhere. Therefore, the class is not given any wander or follow behaviour (only attack behaviour is allowed). This is done by passing false to the parentEnemy constructor parameter ‘canMove’. The rest of the methods in this class are identical to the other enemies, only with the difference of hit points, hit rate and damage. The PiranhaPlant also overrides the resetInstance method, as instead of being removed from the map, like most enemies, it is healed to full and its max HP is increased by 50.

Instead of inheriting Enemy, Flying Koopa class inherits from Koopa class. This is done as FlyingKoopa is essentially identical to Koopa, with the exception that it can walk (fly) on HighGround Ground objects. FlyingKoopa is given a ‘flying’ capability, and the Floor class’ canActorEnter method now simply checks if the Actor has the flying capability to allow Actors to walk on it (once again removing the need for a direct dependency from Floor to FlyingKoopa). Since no changes are made to methods other than the constructor, this obeys ***Liskov Substitution Principle,*** where the derived class has the same behaviour as the parent class, and never produces behaviour that the parent class would not. The mature tree has one of its functions modified; instead of only spawning Koopa, they can also spawn FlyingKoopa with a 50:50 chance of spawning either one.

A3 REQ 3: Magical Fountain

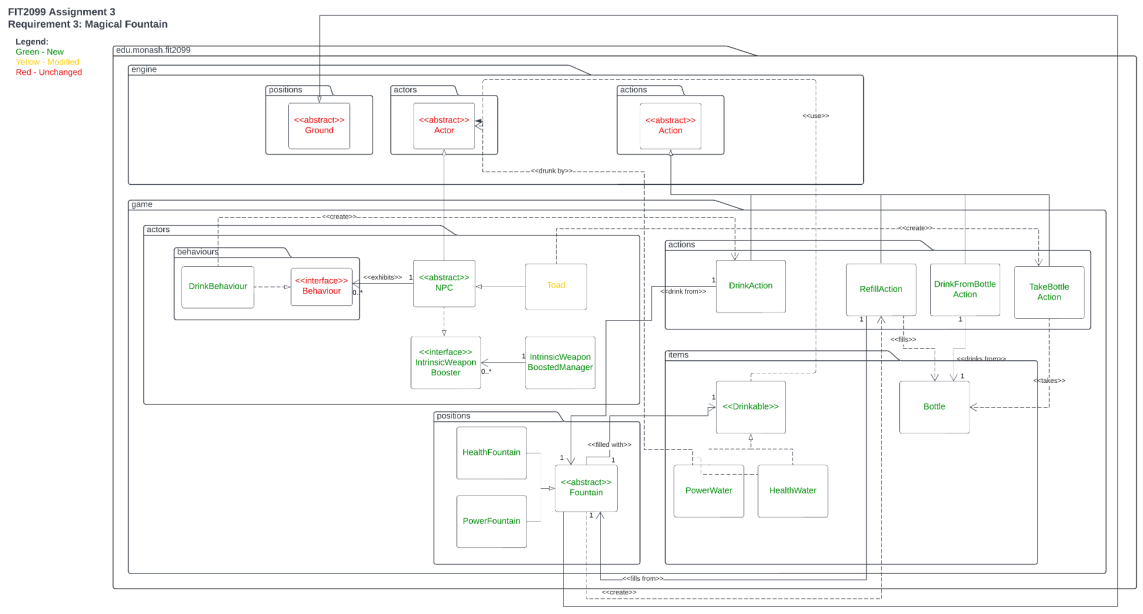

Diagram 1.3 Magical Fountain UML Class Diagram

REQ3 requires creating 2 magical fountains that allows Enemies to drink from and the player to refill a Bottle from. So, an abstract Fountain class is created. This class contains all necessary methods that child Fountain classes use, all the child Fountain classes need to do is pass a unique 'Drinkable' instance (explained below) to the Fountain class' constructor. Creating an abstract Fountain class not only eases further development (e.g. when another fountain is to be created), but also creates an 'outline' of how a fountain should be. This adheres to the ***Liskov Substitution Principle.***

A Drinkable interface is created so that the Fountains can provide different types of water. Both PowerWater and HealthWater implement Drinkable, and are created by the PowerFountain and HealthFountain classes respectively. The Drinkable interface was created separately to the Consumable interface, as the HealthWater and PowerWater should never be added to an Actor's inventory (it doesn't make sense to be able to carry water, unless the

Actor has a Bottle), which is why the Consumable interface was not used. The Drinkable interface provides the drink method, which applies necessary effects to the actor.

A Bottle class is created as a singleton since there can only be one bottle created. Using a singleton helps to prevent an instance to be created again once 1 instance has been created. Bottle can only be taken from Toad. Hence, there is a dependency between Toad class and TakeBottleAction, which is the special action created for retrieving the Bottle from Toad, since it is not purchased/traded.

Once the player is standing on top of the Fountain, there will be multiple actions being executed depending on the condition. There is a RefillAction that allows Bottle to be refilled many times. As seen from the diagram, there is also DrinkAction and DrinkFromBottleAction. This was created after a ***code smell*** is identified in the execute method of DrinkAction. Previously, the DrinkAction had too many tasks in a single function, violating **SRP**, as it was used for drinking from theBottle, as well as when Enemies drink straight from the fountain. By separating the class, the game can give options (to drink from a bottle or immediately from the fountain if the Actor has DrinkBehaviour).

DrinkFromBottleAction has an association with Bottle as it uses the instance of Bottle as its attribute. Meanwhile, Bottle also has a dependency with DrinkFromBottleAction, as DrinkFromBottleActions will only be generated if there are drinks in the Bottle.

The Fountain class also has a dependency with both RefillAction and DrinkAction, as these both get Drinkable instances from the Fountain. DrinkBehaviour is a class implementing the Behaviour interface that works similarly like other Behaviours classes. This is created to allow NPCs with the DrinkBehaviour to drink from the fountain if they are standing on it. The assignment specifications also mentioned that there should be a good balance between an Enemy drinking and its other behaviours such as Attack, Follow and Wander, and to ensure this balance was created, it was decided that Actors that exhibit the DrinkBehaviour will only drink once every alternating turn at most. This prevents Enemies from being killed while they are standing on fountains and lets them attack the player back, as well as ensuring they cannot become too overpowered if they are standing on a PowerFountain.

There is also a newly added interface IntrinsicWeaponBooster. This is implemented in the NPC and Player classes, and its responsibility is to allow Actors to have their intrinsic attack damage boosted. In this case, the intrinsic attack damage is boosted when the Actor drinks from the PowerFountain. By having this, we can increase the base attack by 15, as instructed in the specification. This will be useful for further implementation in case another actor can also have their intrinsic attack boosted. Furthermore, the IntrinsicWeaponBoostedManager manages all instances of this interface, and allows for the PowerWater to increase the base intrinsic attack damage. To easily see the Player's attack damage, an additional 'Attack Damage:' is displayed in the console in the Player's playTurn method, which allows for easy testing of the PowerWater.

A3 REQ 4: Flowers

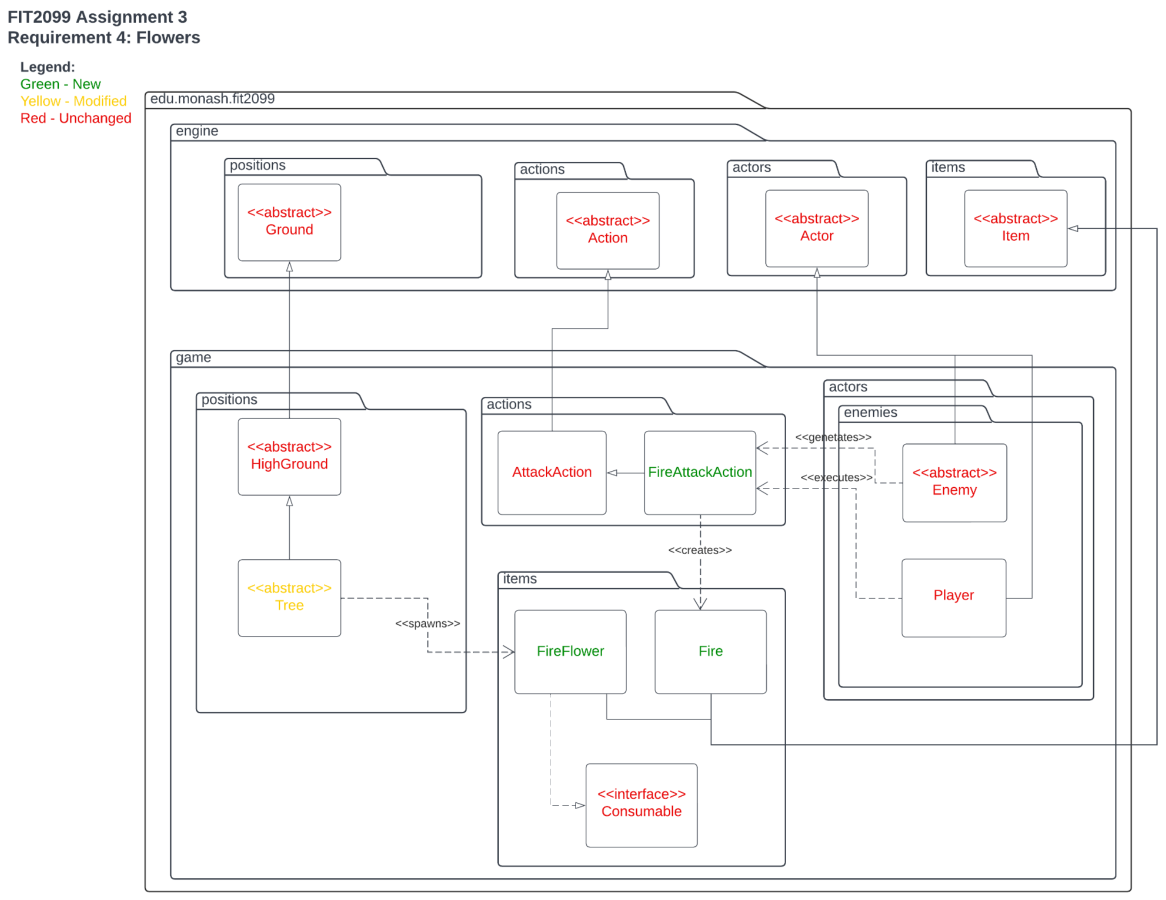

Diagram 1.4 Flowers UML Class Diagram

There are 3 new classes created for REQ4. The first class is FireFlower class. This class is made to allow creation of the instances of FireFlower. This class has a dependency with the Tree class. This is done as there is a 50% chance that FireFlower is spawned for every growing stage of the tree. Tick method will count the age of the tree and once it is in the stage to grow, it will have a half probability that it will spawn to FireFlower instead of the next growing stage of the tree.

Although it seems like FireFlower is a plant and could be a child class of Ground class, FireFlower is able to be consumed. It will be more appropriate for this class to be a child class of Item and implement Consumable interface. Once it is consumed, a FireAttackAction instance will be called. To comply with the ***Single Responsibility Principle,*** FireAttackAction is created instead of putting it in the AttackAction. Since the actual attack part is identical to AttackAction, FireAttackAction extends AttackAction; it overrides some methods and adds its functionality, still abiding by the ***Liskov Substitution Principle.***

To allow for this extension, the AttackAction class now has a boolean attribute that indicates whether the Attack was successful or if it missed, and this allows the Fire Attack Action's execute method to call the execute method of AttackAction, and then if the attack was successful, a Fire item is dropped in the location of the target.

Since FireAttack can only last for 20 turns (work in a similar way as PowerStar), the method tick is used in the FireFlower. This is done to give a 'lifetime' for the FireFlower once it is consumed. The consumption of the FireFlower gives the status 'fireattack', which allows Enemies to generate FireAttackActions if the other actor has this status (once again removing the need for a direct dependency between Enemy and FireFlower). Fire will stay on the ground for 3 turns (using the tick method to remove the fire after 3 turns), give damage to any actor standing on it and then disappear. This is why there is a dependency between FireAttackAction and Fire.

Furthermore, to ensure dormant Koopa's do not take damage from Fire on the ground (only a Wrench should be able to break their Shells), once a Koopa goes dormant, it is given the status 'immune_to_fire' and in Fire's tick method, it checks if the Actor standing on the fire has this immune to fire status, only giving damage if the Actor does not have this status.

A3 REQ 5: Speaking

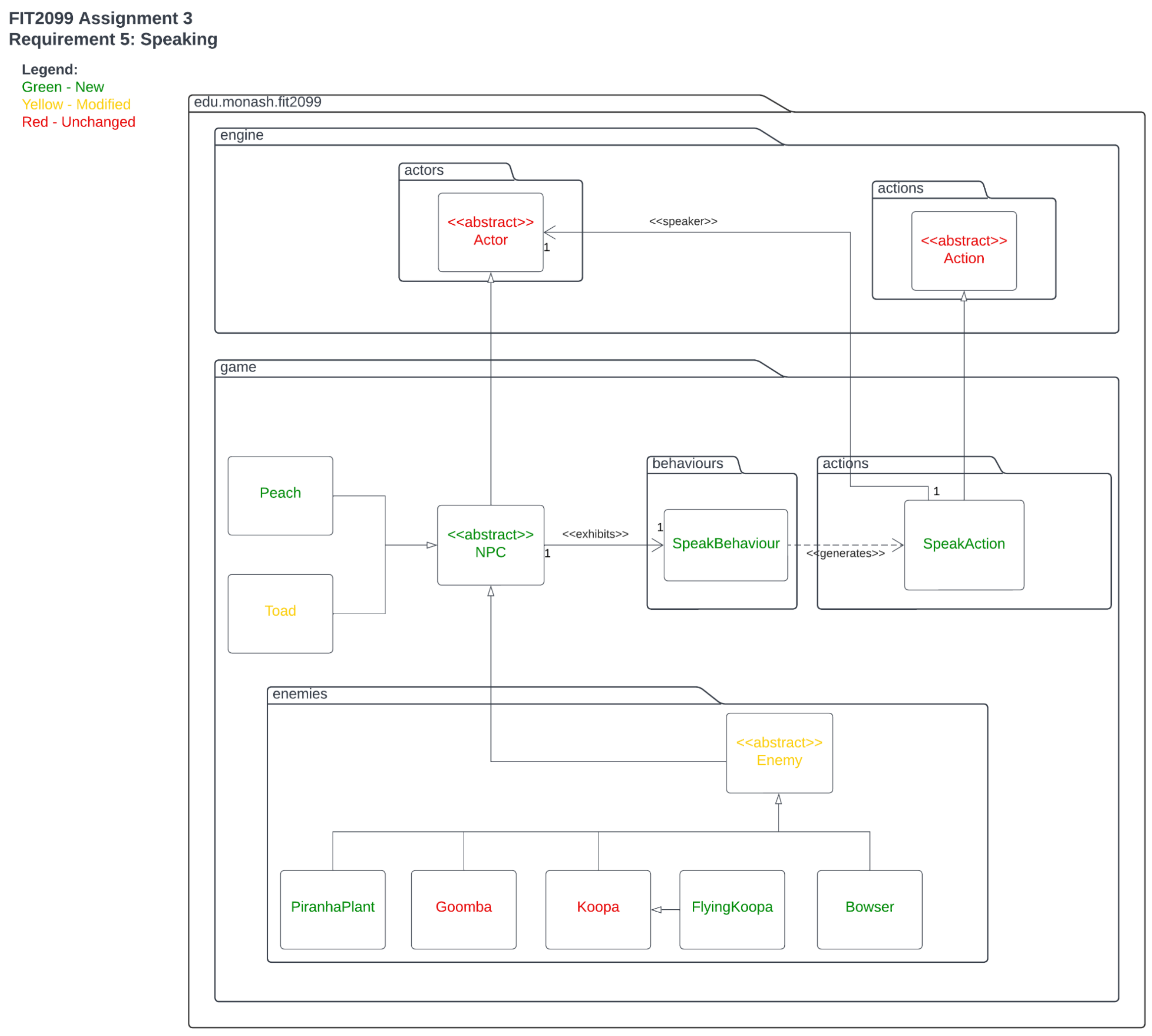

Diagram 1.5 Speaking UML Class Diagram

Previously, new actors have been created (as explained in REQ2). In this part, a new abstract NPC class is introduced. The NPC class is created as a parent class for all non-playable Actors. The NPC class now encapsulates behaviours, rather than the Enemy class. This may prove very useful in future implementations, for example if we wanted new actors to be able to exhibit behaviours (e.g. Toad can have WanderBehaviour in the future), all that needs to be done is call the NPCs addBehaviour method. But right now, the NPC class is mainly used for speaking. There are 2 classes created that are mainly related to this requirement – SpeakBehaviour and SpeakAction.

SpeakAction is modified from the previous assignment, as now it accepts two parameters in its constructor; the line that is to be spoken, as well as the Actor that is speaking the line. This is to allow for all NPCs to speak to the console and have their name displayed in the required format.

To allow NPC executing the SpeakActions, the SpeakBehaviour class is needed. The getAction method in SpeakBehaviour class allows the actor to speak in alternating turns, so as to not fill the console with too many messages. This only speaking in alternating rounds is implemented via a boolean attribute of the SpeakBehaviour class, that indicates whether the NPC spoke in the previous turn. The NPC class handles the SpeakBehaviour directly, and provides necessary methods to initialise the lines that an NPC might say, as well as methods to clear all lines (which is currently used when Koopa's go dormant) and finally a simple 'speak' method, which can be called in playTurn methods to allow the NPC to actually speak.

Changes Since Assignment 2 Submission:

Tree No Longer Uses TreeType Enum:

In the initial code, the type of tree was determined by a 'TreeType' enum attribute of Tree, which violated the OCP, as to create a new type of tree, the existing Tree class needed to be modified. In the new implementation, Tree is an abstract class, and each tree Type extends this abstract class. This now obeys the OCP, and all that needs to be done to create a new tree type is create a new child class of Tree, overriding the tick, jumpSuccess and fallDamage methods, and the existing classes do not need to be modified.

Toad no longer checks inventory when generating Monologue:

In the initial implementation, when Toad generated a monologue, it had to directly check the inventory of the Actor to see if the Actor had a wrench in its inventory, which was an unnecessary dependency. In the new implementation of the code, the Wrench is given a ‘can_destroy_koopa_shells’ status, which Toad checks the Actor for. This removed the unnecessary dependency between Toad and Wrench. Furthermore, the dormant Koopa now no longer searches for a Wrench, rather it looks for the ‘can_destroy_koopa_shells’ status to generate an AttackAction on itself in its getAllowableActions method.

Enemy constructor now accepts ‘canMove’ parameter:

With the addition of the PiranhaPlant, an Enemy that is always in the same position, it became necessary to remove the wander and follow behaviours from PiranhaPlant. Instead of simply removing the behaviours in PiranhaPlant, which would necessitate overriding the playTurn and getAllowableActions method and duplicating large portions of the code, it was decided that the Enemy class should have a ‘canMove’ attribute, and then the Enemy class; playTurn / getAllowableActions methods only add FollowBehaviour if the canMove attribute is true. This allows for future iterations of the game to create new stationary enemies by simply extending Enemy and passing canMove=false to the Enemy constructor, which makes the game much more extensible.

SpeakAction is now generalised to any Actor, not just Toad

Initially, Toad was the only Actor who spoke monologue, and hence the SpeakAction did not state which actor was speaking (as there was only one possible actor that could be speaking). However with the introduction of the Enemies and Allies speaking, SpeakAction was modified to have a ‘speaker’ Actor attribute, so that when executed, it would be clear which Actor was speaking.

Introduced the NPC class to manage Behaviours

As detailed above, the Enemy class originally handled the map Behaviours and BehaviourPriorities, however to allow the game to be more extendable, an NPC class was made that both Enemies and Friendly non-playable Actors can extend from. Additionally, this NPC class allowed for the Speaking requirement to be implemented much easier.

Assignment 1/2 Requirements:

REQ 1: Trees

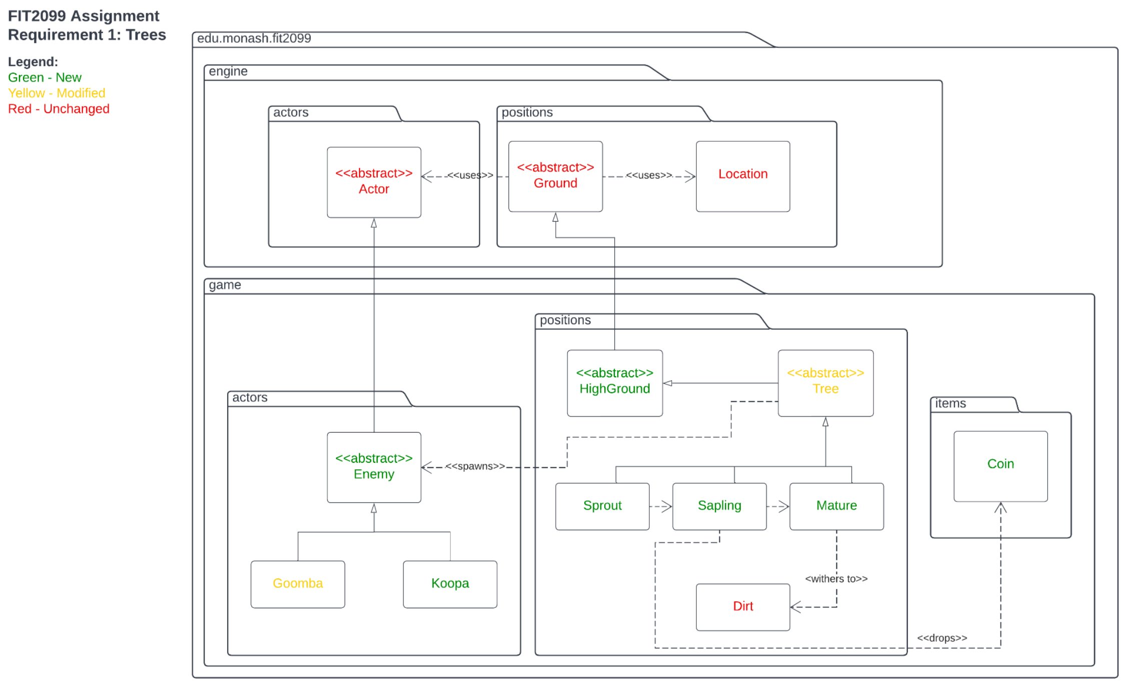

Diagram 2.1 Tree UML Class Diagram

In this UML diagram, it can be seen that *Tree* is an **extended** class of the abstract *HighGround* class, which extends from *Ground*. The *Tree* Class is modified in order to create different methods. The *HighGround* abstract class (discussed in REQ2), is implemented for the Player to be able to jump up to the trees.
The Tree class is made abstract, and its child classes are the three types of trees currently in the game; sprout, sapling and mature. The abstract Tree class provides the reset method, as well as all methods necessary for managing the age of the tree. This way, all child Tree classes need to implement is the display character, jump chance success, jump fall damage and tick methods, hence obeying the ***Open Closed Principle***.

There is also a **dependency** between *Tree* and *Dirt* as it is said that a mature tree has a probability of turning to dirt. Additionally, the Mature Tree grows a new sprout in an adjacent fertile square every 5 turns.

The *Tree* Class also has a **dependency** on *Coin* Class. This is because each turn, there is a chance for a sapling Tree to drop a coin.

The *Tree* Class also has a **dependency** with the abstract *Enemy* class, because Goomba, Koopa and FlyingKoopa can be spawned by Trees.

REQ 2: Jump Up

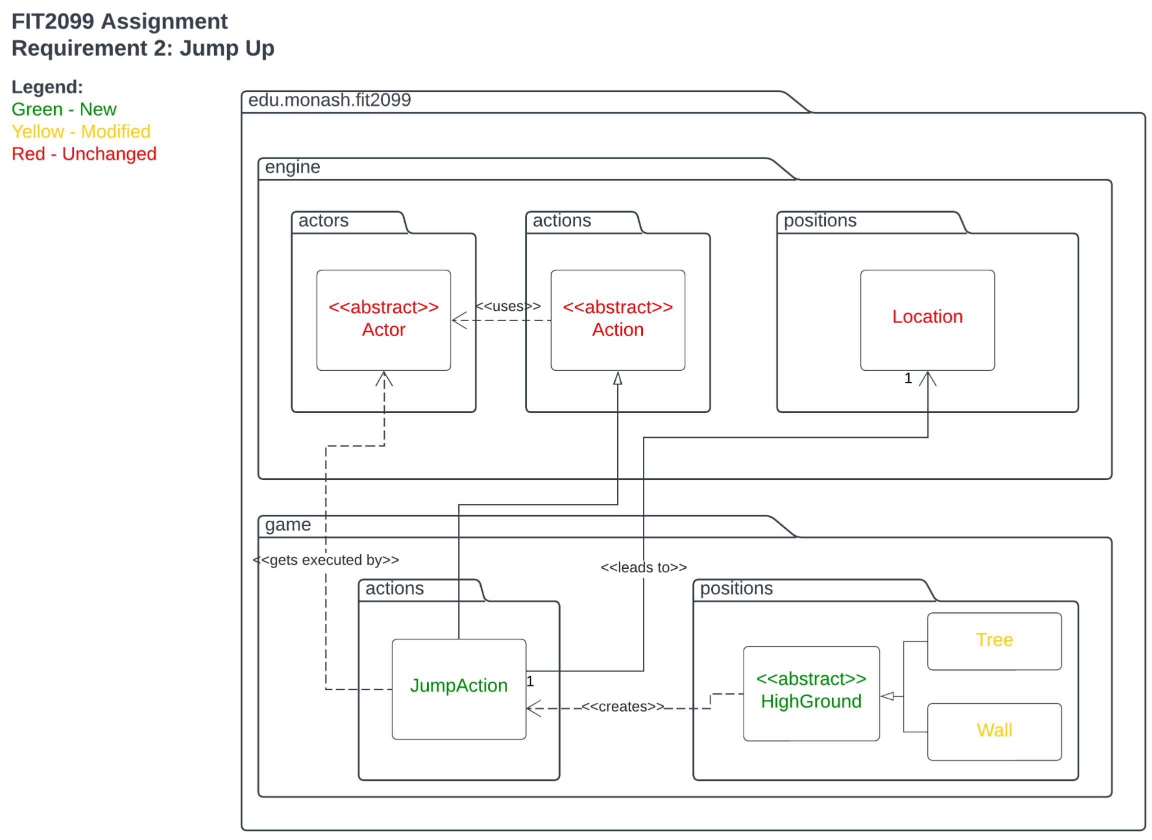

Diagram 2.2 Jump Up UML Class Diagram

In REQ2, the main focus will be the *JumpAction* class. *JumpAction* class is a child class of the *Action* class, since it overrides methods of the parent Behaviour class. The JumpAction class was added in order to obey the ***Single Responsibility Principle (SRP)***. Instead of adding methods in the available class, a new class is created to support this functionality. There is a **dependency** between the *JumpAction* class and *Actor abstract* class. This is because each JumpAction instance will have an Actor object that it is executed by. Having a **dependency** is required since the program needs to know the personal information of the Actor object (such as whether or not the Actor has a power-up/capability to take fall damage or not), as well as to be able to actually subtract fall damage from the Actor's health.

There is also an **association** between *JumpAction* class and *Location* class with a cardinality of 1:1. The reason why association exists between these two classes is because for a JumpAction to be executed, a ‘destination’ Location object is required, in order to know where the actor is going. The Location will be the deciding factor for JumpAction to know where it reaches (e.g. Tree or Wall) before other behaviour is being executed progressively.

The *HighGround* class has a dependency with *JumpAction*, as the *HighGround* class will generate *JumpAction*s for the Actors to possibly execute. A method will use the JumpAction object to calculate the success rate based on the wall and trees that are jumped from. This needs to be done since the fall damage of the player should not be calculated in JumpAction and the exact value can only be known based on whether the Player is jumping to a Wall or Tree.
The *HighGround* class overrides the canActorEnter method, and now only allows Actors with a ‘flying’ status, or ‘invincible’ status to enter, as those two statuses are the only way an Actor can move on top of a HighGround object without jumping.

REQ 3: Enemies

See REQ 2: More Enemies and Allies for assignment 3 above, as Enemies have been overhauled and are explained in detail there.

REQ 4: Magical Items

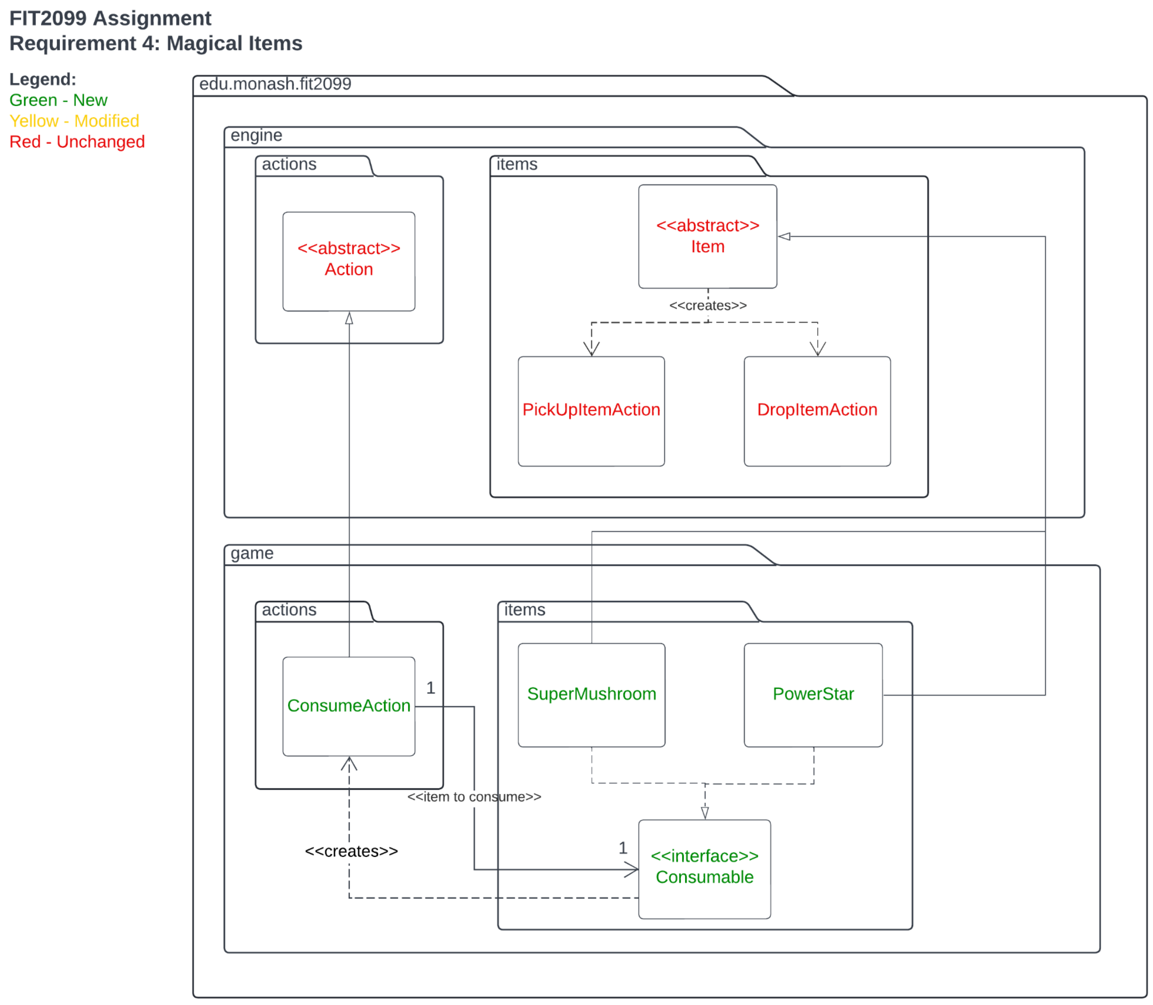

Diagram 2.4 Magical Items UML Class Diagram

In REQ4, most of the classes are newly added. In the game scenario, there are two magical items which are Super Mushroom and Power Star. Hence, the *SuperMushroom* and *PowerStar* classes are created. Instead of only creating 1 class for both magical items, we created one for each in order to obey ***Single Responsibility Principle (SRP)*** which only allows one responsibility (managing the magical items object).

Both *SuperMushroom* and *PowerStar* also extend the *Item* abstract class as they have the functionality of any other items. Both *SuperMushroom* and *PowerStar* implement the *Consumable* interface. Consumable interfaces generate ConsumeItemActions, which will be executed when certain capabilities are obtained by the player when the Player consumes the Magical Items. Therefore, an interface is used since it will be implemented by both classes as they need this functionality.

The ConsumeAction class is also set as the child class of the parent *Action abstract* class. Similar to the other inherited classes, *ConsumeAction* will inherit all of the methods that were previously declared in Action class, with slight modification on the code by overriding methods such as the execute method. By doing this, the ***Liskov Substitution Principle (LSP)*** is achieved, as the Action can still be executed by the World.processActorTurn method.

The ConsumeAction will handle the task of applying the buff, debuffs or capabilities to the Actor that executes that Action (not shown in the UML class diagram for this requirement, as we have established previously that Actor uses Action).

Each ConsumeAction instance will have a Consumable object that it acts on, hence the 1:1 cardinality between ConsumeAction and the Consumable interface.

Also included in this UML diagram is the PickUpItemAction and the DropItemAction, as they relate quite closely to this requirement. They are both quite self-explanatory based on their names; they implement the functionality for an Actor to pick up an item (add it to the Actor's inventory), and for an Actor to drop an item (remove it from the Actor's inventory).

REQ 5: Trading

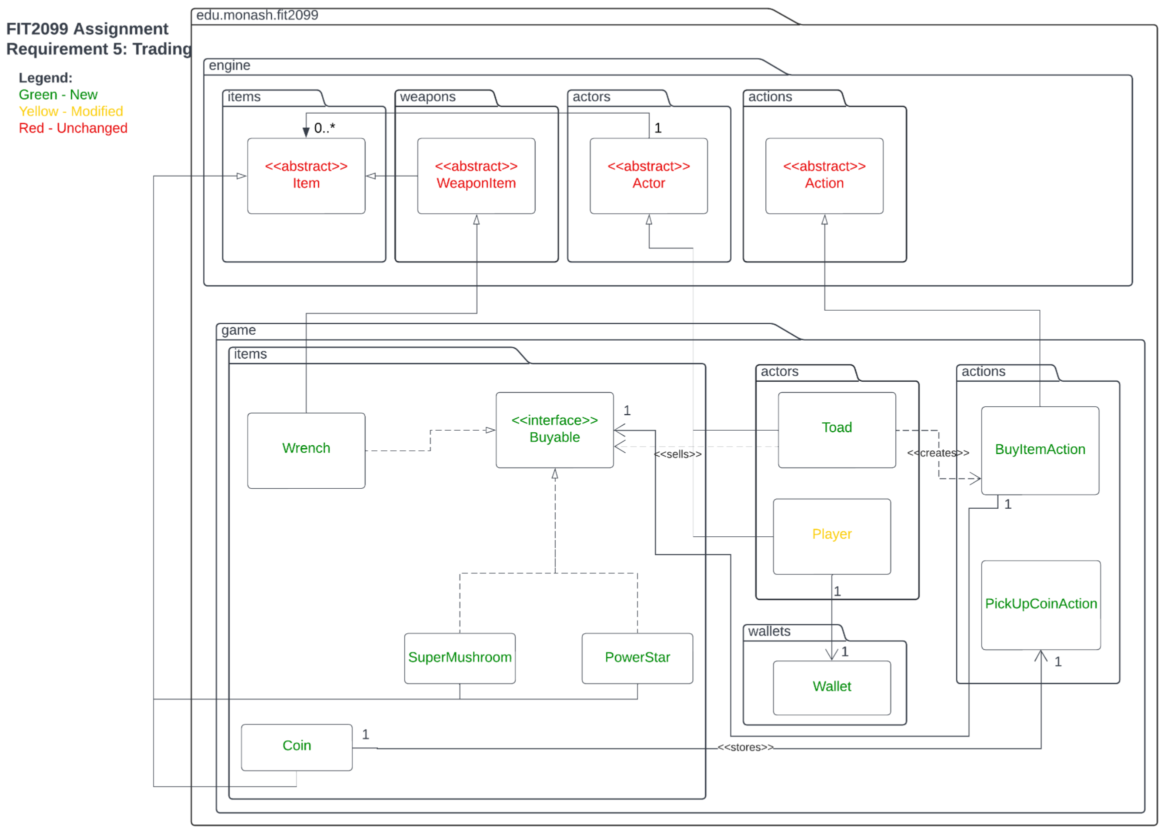

Diagram 2.5 Trading UML Class Diagram

The UML diagram above shows how REQ5 is going to be implemented in the program. Both *BuyableWeapon abstract* class and *BuyableItem abstract* class **implement** the *Buyable* interface.

The *BuyItemAction* class which has an **association** with objects that implement the *Buyable* interface. This is done as a Buyable object should exist in order for BuyItemAction to be executed. Without it, BuyItemAction would not be executed in the first place. Before BuyItemAction can be executed, *the Toad* object will need to be called first. Toad act as the bridge for purchasing to happen. It has a **dependency** with *BuyItemAction* as it creates the *BuyItemAction*s.

The Coin class extends the abstract Item class, as it shares the majority of its functionality with the Item class. A Wallet class is created for the purpose of keeping the coins that the player has collected. This is shown by the 1:1 cardinality of the **association** from *Player* to *Wallet*. Since Wallet collects coins, a **dependency** between *Wallet* and *Coin* is created where an instance of coin will be used in one of Wallet methods. The value of the coin that the player obtained will be created as an instance and the total will be counted in the Wallet.
A new action *PickUpCoinAction* has been created, as since the regular *PickUpItemAction* is part of the game engine, we cannot modify it to add coins to an Actor's wallet. Furthermore, adding an action to specifically pick up coins obeys the **Single Responsibility Principle (SRP)** as *PickUpItemAction* was specifically to add an item to an actor's inventory, and since coin is not added to the inventory when it is picked up, it is better design to implement a new Action.

Both *Toad* and *Player* are one of the **extended classes** of *Actor*, since both Toad and Player will make use of methods that the abstract Actor class provides. This goes the same for the *BuyItemAction* class which **inherits** from the superclass *Action*, *Coin*.

There is also an **association** from the *Actor* class to the *Item* class, as each Actor has a collection of 0 or more Item instances, in its inventory attribute.

REQ 6: Monologue

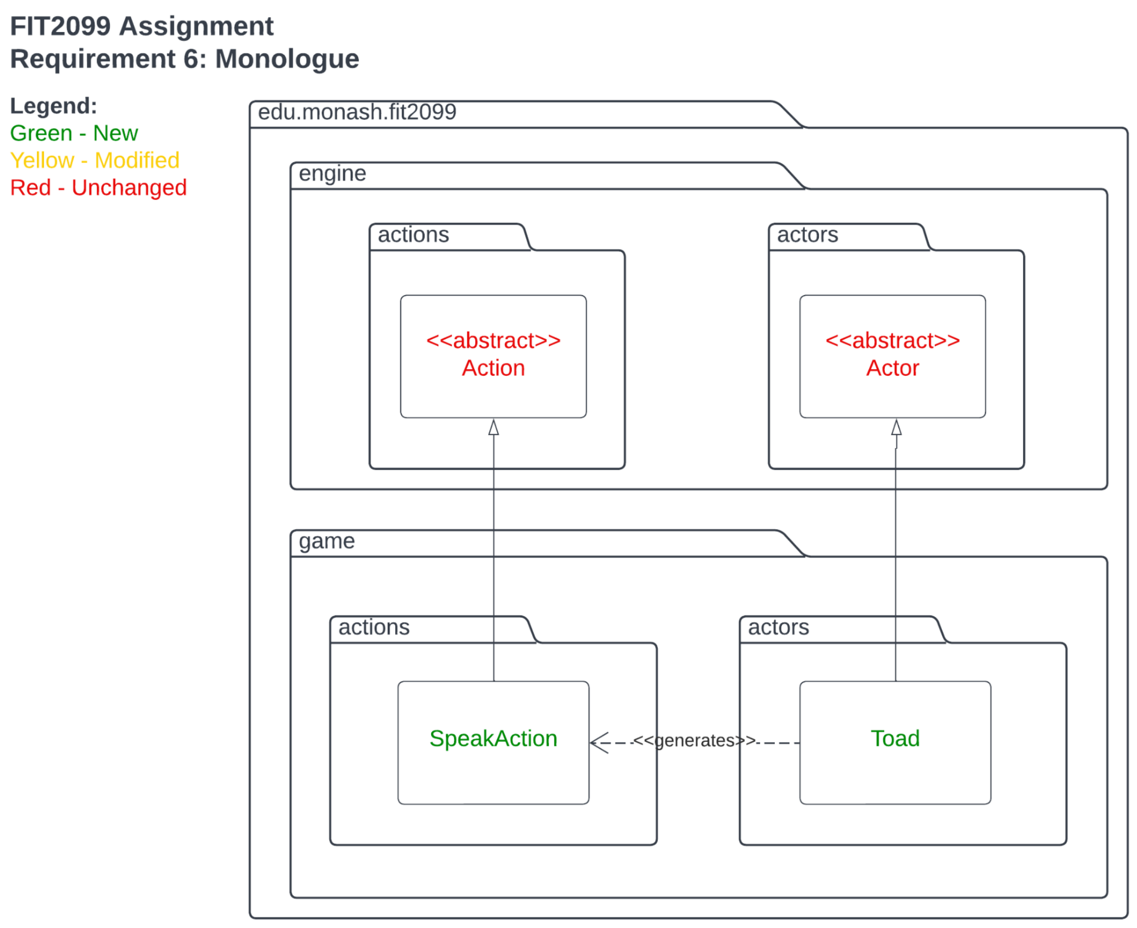

Diagram 2.6 Monologue UML Class Diagram

REQ6 deals with having Toad give a dynamic monologue to the user when the Player chooses to speak to Toad.

This can be implemented within the overriding allowableActions method in the Toad class (inherited from the abstract Actor class). In this method, a single randomly chosen *SpeakAction* will be added to the output ActionList. However, if the player already has a wrench , Toad will not speak the sentence relating to the Wrench, and similarly if the Player has used a PowerStar and it is still active, Toad will not speak the sentence relating to the PowerStar. To allow Toad to check if the player has a wrench without introducing a dependency from Toad to Wrench, a status 'can_destroy_koopa_shells' is given to Wrench (and in effect, to the Player holding the Wrench), and similarly with the 'invincible' status for PowerStars. The *SpeakAction* class is used so that whatever line Toad must speak can be printed to the console.

REQ 7: Reset Game

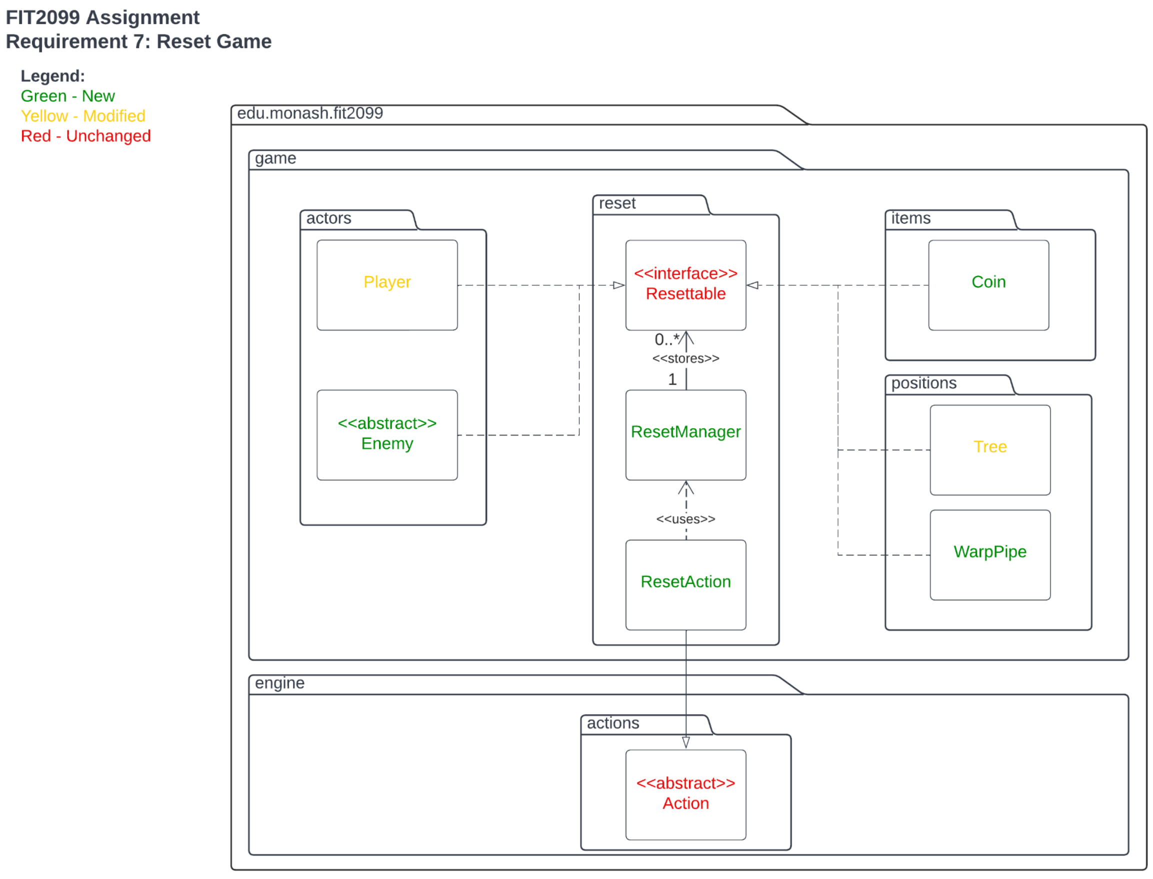

Diagram 2.7 Reset Game UML Class Diagram

REQ7 deals with allowing the player to ‘reset’ the game once. When the user chooses to reset, the following will happen:

- “Trees have a 50% chance to be converted back to Dirt”, hence the Tree class implements the Resettable interface
- “All enemies are killed”, hence the abstract Enemy class implements the Resettable interface. This allows all child classes of Enemy (for the current state of the game, this means Goomba and Koopa) to inherit the implementation of the Resettable interface. This is permitted as all enemies are destroyed, regardless of their specific enemy type
- “Reset player status” and “Heal player to maximum”, hence the Player class implements the Resettable interface
- “Remove all coins on the ground (Super Mushrooms and Power Stars may stay).”, hence the Coin class implements the Resettable interface. Since only coins are being removed, and not other types of items, only the coin class implements the Resettable interface.
- WarpPipes that have a PiranhaPlant on them will increase the max health of the PiranhaPlant by 50, and warpPipes that don’t have a PiranhaPlant on them will spawn a new PiranhaPlant on them. Hence, WarpPipe implements the resettable interface.

The Resettable interface will consist of a method that is called when the user opts to reset the game. Each implementing class will override this method and carry out the above functionality.

In order to keep track of all of the Resettable objects, rather than iterating over each Item in each Location in each GameMap and calling a ‘reset’ method that is either empty or carries out a certain action (similar to how the tick methods are executed each turn in the game engine code), which is a rather over-complicated and will have many redundant empty methods, we instead opt to use a singleton class ResetManager, which will track every instantiated object that implements the Resettable interface, and when the user chooses to reset, the collection of Resettable objects in ResetManager will be iterated over, for each object, the reset method being called.

This implementation has the one disadvantage in that for every class that implements the Resettable interface, all constructors for that class must add the newly created object to the ResetManager. If the alternate implementation was used (the one inspired by the tick methods), the ResetManager would not be needed.

The ResetAction handles the actual logic of resetting each resettable interface object. It inherits from the abstract Action class, and uses the ResetManager singleton to reset each Resettable object when the ResetAction is executed.

Interaction Diagrams

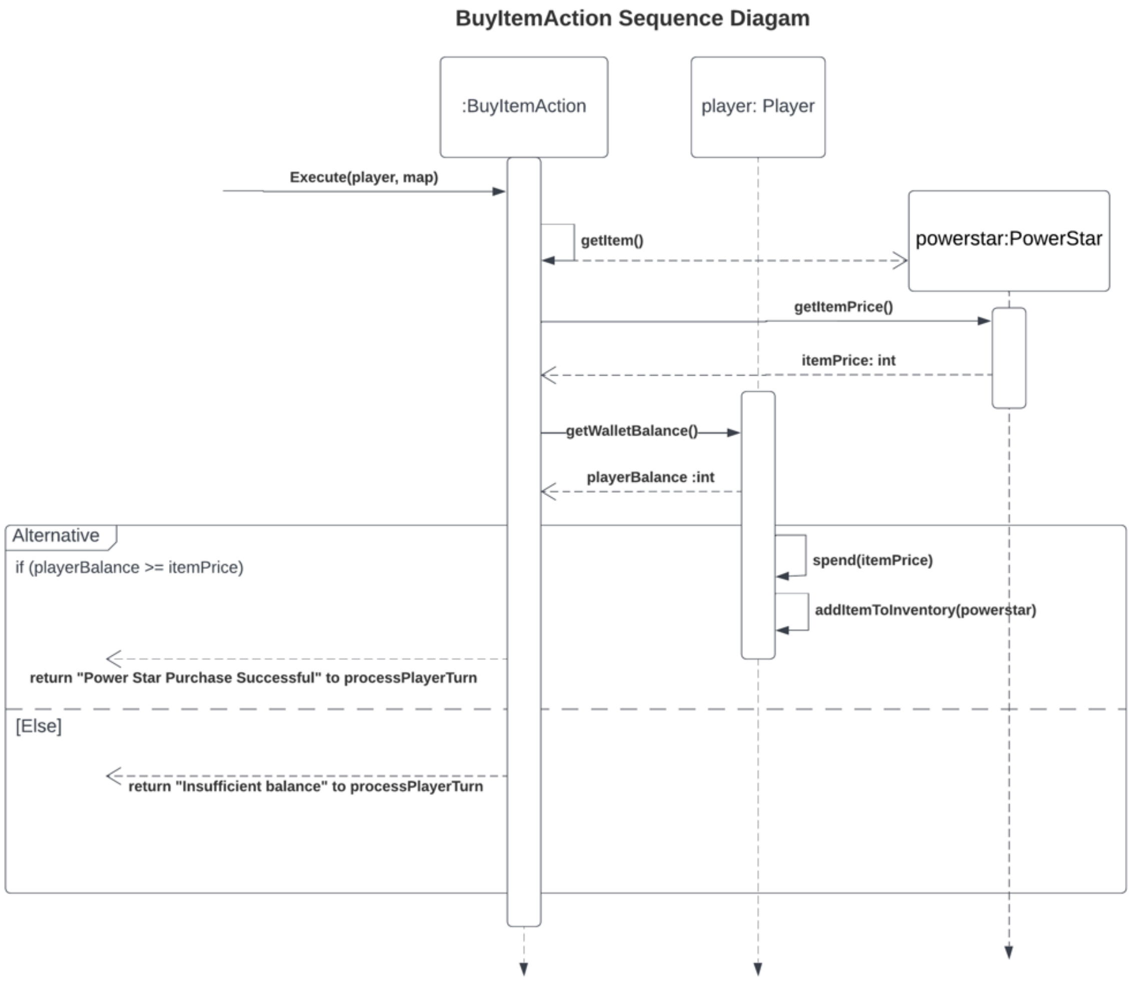
BuyItemAction Sequence Diagam
:BuyItemAction
player: Player
powerstar:PowerStar
Execute(player, map)
getItem()
getItemPrice()
itemPrice: int
getWalletBalance()
playerBalance :int
Alternative
if (playerBalance >= itemPrice)
spend(itemPrice)
addItemToInventory(powerstar)
return "Power Star Purchase Successful" to processPlayerTurn
[Else]
return "Insufficient balance" to processPlayerTurn

Diagram 3.1 Trading Sequence Diagram (PowerStar Item)

In this assignment, not all sequence diagrams were required. Hence, we decided to create a sequence diagram of one of the REQs that we have made many modifications and additions to. From the UML diagram, REQ5 can be seen to have many new classes added.

In this event, the BuyItemAction class is executed first with the starting method of execute(player, map). The BuyItemAction class will first call the getItem method on itself to know which item is chosen by the player. From getItem. In this case, a powerStar instance is created as a local variable in the execute method. The BuyItemAction class will then call getItemPrice in order to get the price of the magical star.

Next, the Player class is used to call the getWalletBalance method in order to retrieve the wallet balance. Now that the 2 attributes required have been retrieved, BuyItemAction class creates an if condition, which is represented by the alternative fragment. In this diagram, the fragment shows 2 conditions to check whether the player is able to buy the powerStar. If playerWallet is larger or equal to itemPrice, the first inner frame will be executed. In this case, the Player class will call methods on themselves to deduct the balance of the player's wallet by calling the function spend(itemPrice), which removes itemPrice from the Player's wallet. Then, it will add the newly bought powerStar instance to the inventory of the player using addItemInventory.

It will then display the message showing that purchase is successful. This is done by returning the string "Power Star purchase successful" to processPlayerTurn. The second inner frame will be executed instead if playerWallet is not larger or equal to itemPrice. Similar to the previous part, the BuyItemAction class will return an "Insufficient balance" string to processPlayerTurn. After either one of these are executed, this marks the end of the Trading sequence, and the end of the lifeline.

New Assignment 3 Sequence Diagram:

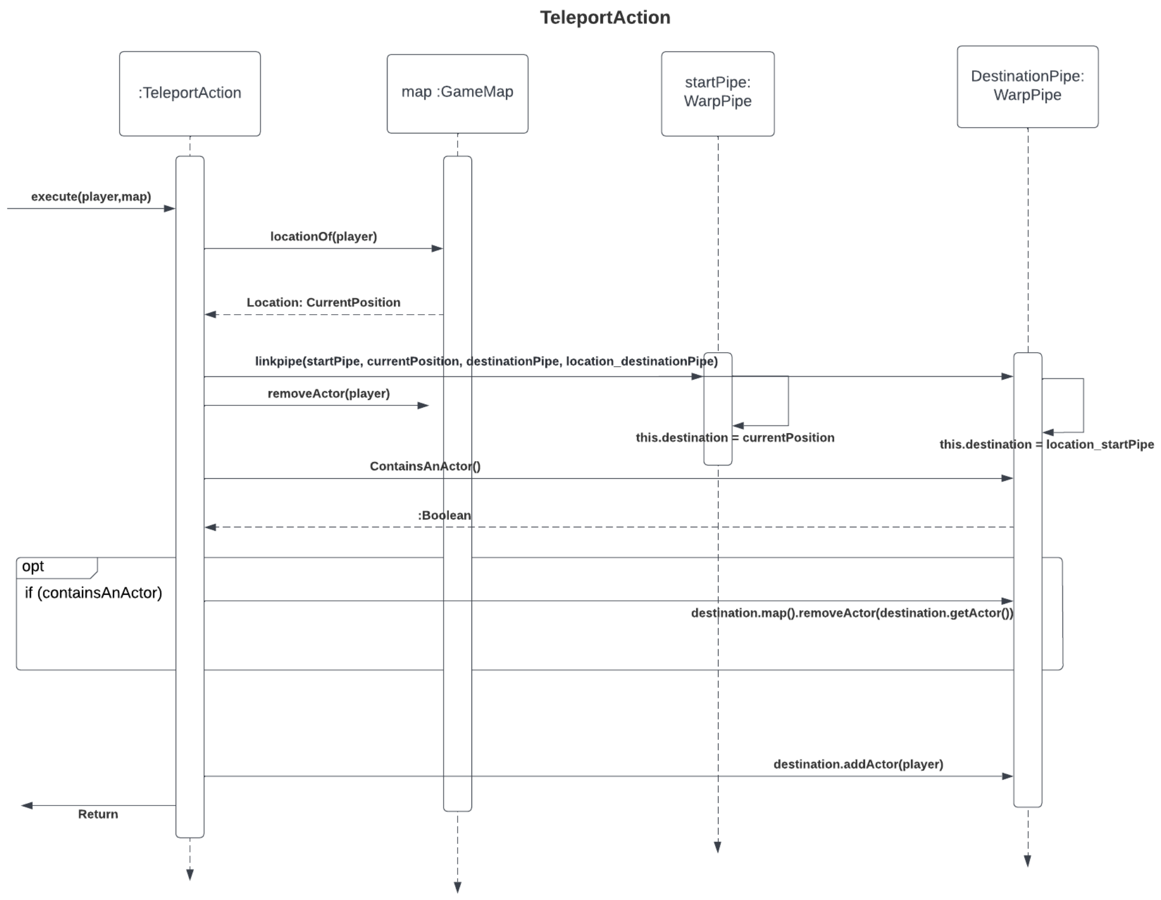
TeleportAction
:TeleportAction
map :GameMap
startPipe: WarpPipe
DestinationPipe: WarpPipe
execute(player,map)
locationOf(player)
Location: CurrentPosition
linkpipe(startPipe, currentPosition, destinationPipe, location_destinationPipe)
removeActor(player)
this.destination = currentPosition
this.destination = location_startPipe
ContainsAnActor()
:Boolean
opt
if (containsAnActor)
destination.map().removeActor(destination.getActor())
destination.addActor(player)
Return

Diagram 3.2 TeleportAction Sequence Diagram

In this diagram, the sequence starts from TeleportAction being executed. It takes the parameters of the player and the map. It will then find the location of the player in that map, using the method locationOf(player). In this part, the GameMap class will return the location of the player. The returned value is sent back to the TeleportAction class.

Once it gets the position of the player, it will use the method linkPipe(startPipe, currentPosition, destinationPipe, location_destinationPipe) to make the two WarpPipes linked to one another. This method goes to the first (start) WarpPipe class, and will return the currentPosition. In the TeleportAction, the actor will be removed from its current map and the player is no longer on its original map.

Since the start WarpPipe is no longer used, the activation bar ends. It will also check whether the end WarpPipe contains an actor and return boolean. Then, a fragment is used in order for execution to happen when the condition is true. In this example, the guard is (ContainsAnActor). If the guard returns true, the actor that is located in that position will be removed from the map, the most common reason for this is to remove the PiranhaPlant that is on the destination WarpPipe. It will then add the actor to the other destination on the end WarpPipe. Once this is executed, execute(player, map) will return the string to give a message. This marks the end of all the activation bars of the 4 classes.