# FIT2099 ASSIGNMENT 1

## *Design and Planning*

**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.

## DESIGN RATIONALE

### REQ 1: Trees

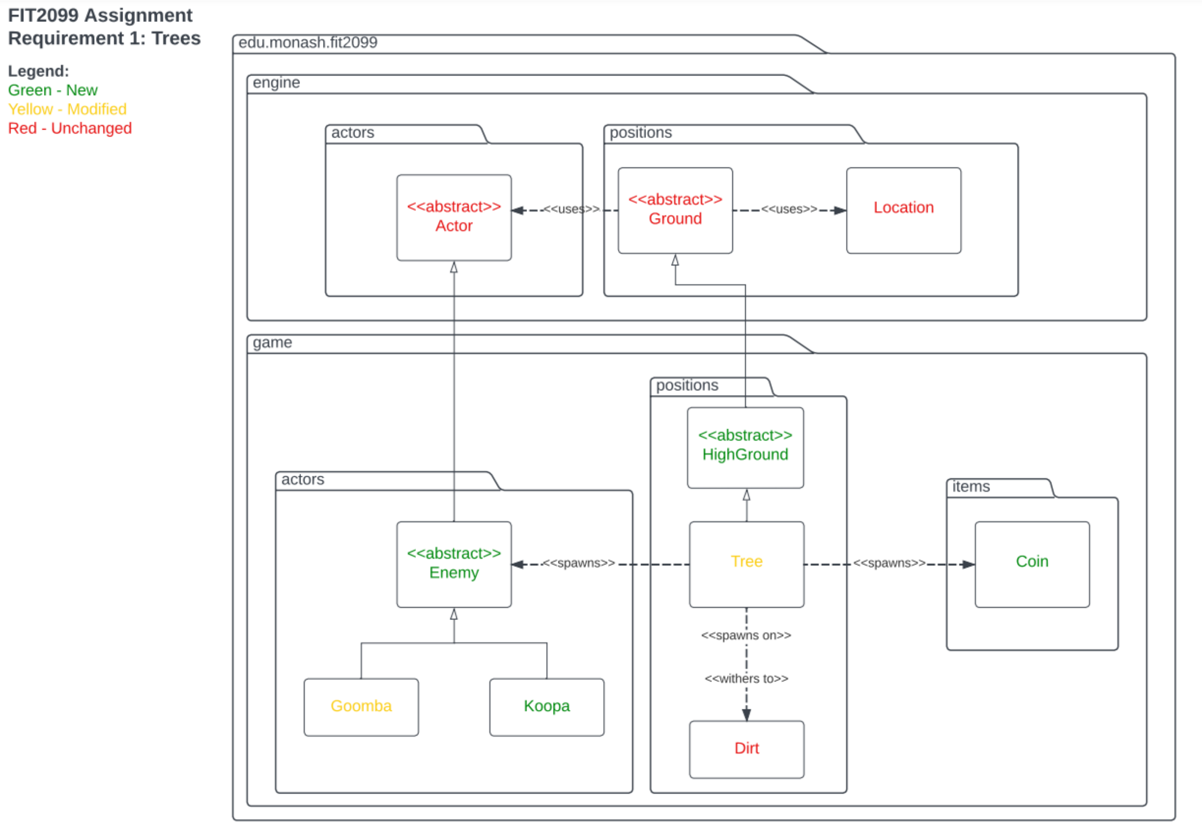


*Diagram 1.1 Tree UML Class Diagram*

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 will have an Enum attribute that tracks the current state of the Tree (Sprout, Sapling, or Mature). Subsequently, some methods in the Tree class will then have differing functionality, based on the current state of the Tree.

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. The reasoning behind this is because both enemies – Goomba and Koopa – have a dependency with Tree. By doing this, ***Dependency Inversion Principle (DIP)*** is achievable since an abstract class is used in order to prevent modules, in this case the Tree, having unnecessary dependencies on low-level modules, which are Goomba and Koopa.

## REQ 2: Jump Up

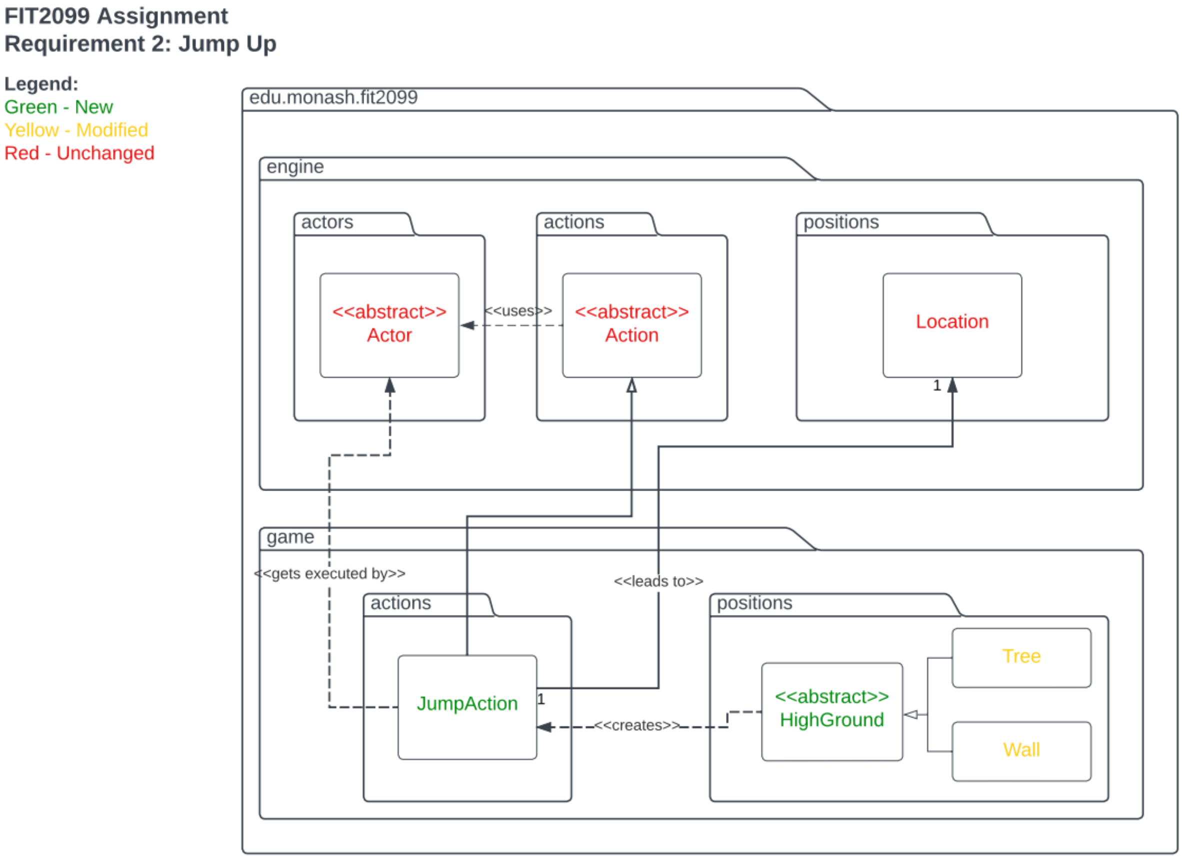


*Diagram 1.2 Jump Up UML Class Diagram*

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.

## REQ 3: Enemies

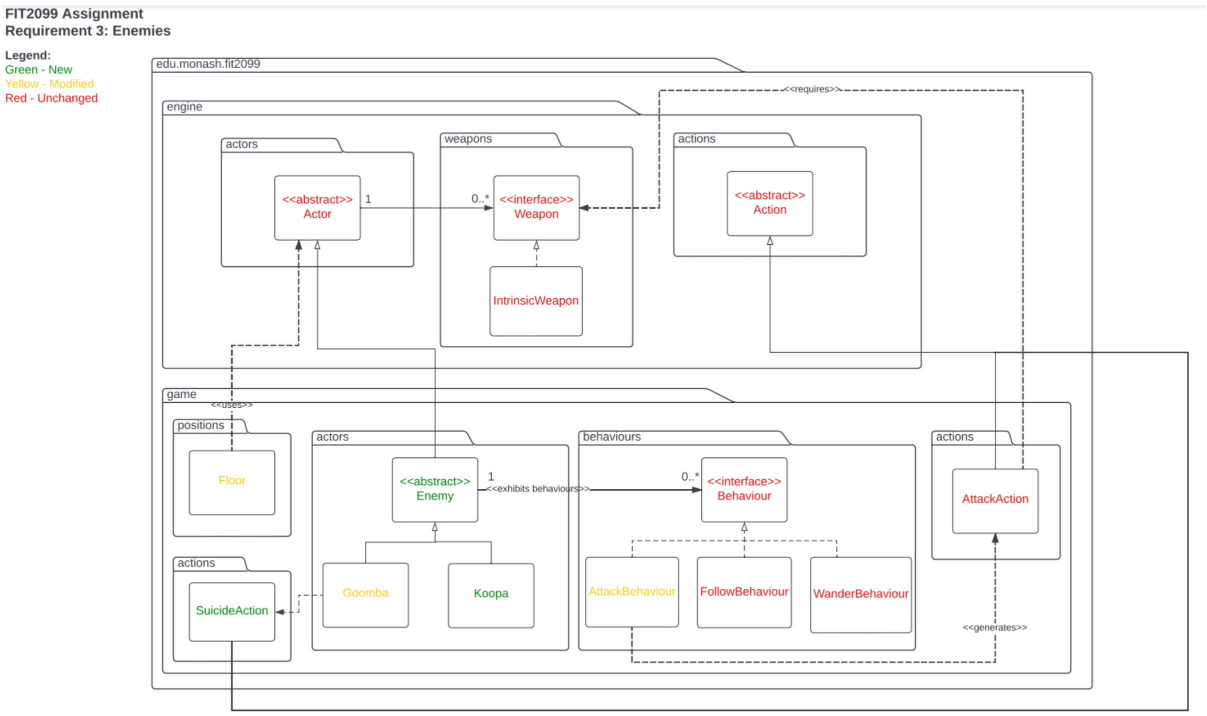


*Diagram 1.3 Enemies UML Class Diagram*

RE3 mainly talks about the *Enemy* Class. In the code given, there was no class for Koopa, therefore, a *Koopa* class was created. Instead of combining it with Goomba or inheriting from Goomba, Koopa has its own class since both Koopa and Goomba have slightly different characteristics and behaviours. For example, Goomba starts with 20 HP whereas Koopa starts with 100 HP. Since there are now 2 enemies, an abstract *Enemy* class is created. The abstract *Enemy* class is the **parent class** of *Goomba* and *Koopa*. The purpose of creating this is so that each child class of Enemy can override the functionality that is required, while still sharing some of the same inherited methods and attributes from the Enemy class.

The *Enemy* class **extends** the abstract *Actor* class. This is done as the Enemy class has all the functionality of the Actor class. Enemies can have various behaviours as attributes, such as FollowBehaviour, WanderBehaviour and AttackBehaviour, hence the Enemy class must extend actor.

There is also an association between the *Enemy* class and *Behaviour* interface. In these three Behaviour classes, a **getAction()** method will be implemented. This method determines what a certain Actor will perform, for example if the Enemy has the WanderBehaviour Behaviour, it will 'wander' randomly each turn. Similarly, if the Enemy has the FollowBehaviour, it will follow the Player each turn. The enemies have a collection of Behaviours, as indicated by the 1:0..* cardinality from Enemy to Behaviour, and the highest priority Behaviour that the Enemy has will be the Behaviour that the Enemy exhibits. The three Behaviours in the current plan for the game are WanderBehaviour, FollowBehaviour, and AttackBehaviour.

The AttackBehaviour will generate an AttackAction, which will handle the logic of the actual 'attack' move. The AttackAction requires a Weapon, and hence has a dependency on the Weapon interface.

To prevent Enemies from being able to enter Floor tiles, there is a dependency from Floor to the abstract Actor class. Then from within the Floor class, the canActorEnter(Actor actor) method will be overridden, to return true if the Actor is not an Enemy, and false if the Actor is an Enemy.

To allow the Goomba to commit suicide, there has been a new *SuicideAction* Action class, which will be randomly exectued 10% of the time. This simplifies the auto-death of the Goomba, and allows any other actor to execute it (for future modifications to the game).

**REQ 4: Magical Items**

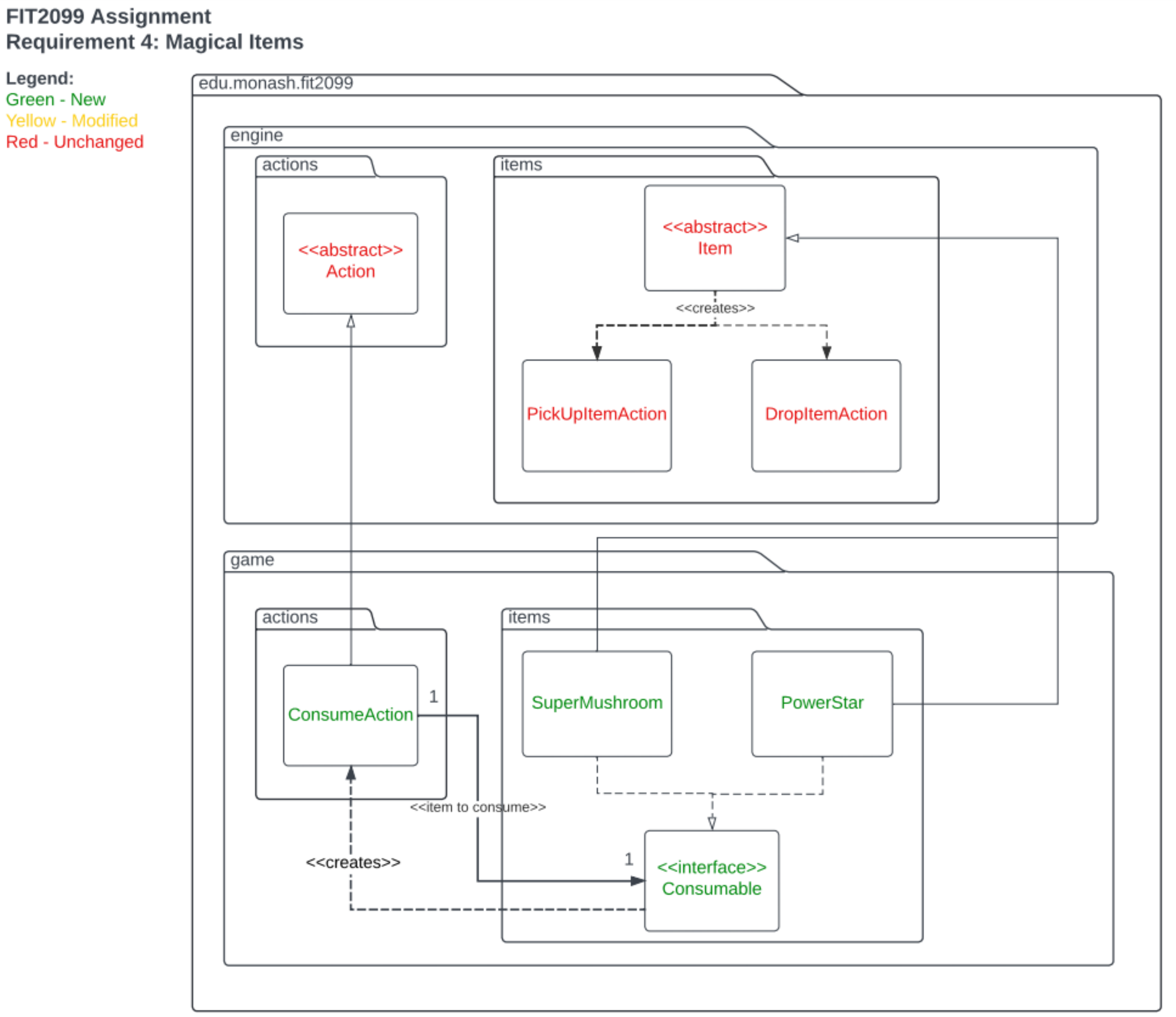


*Diagram 1.4 Magical Items UML Class Diagram*

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

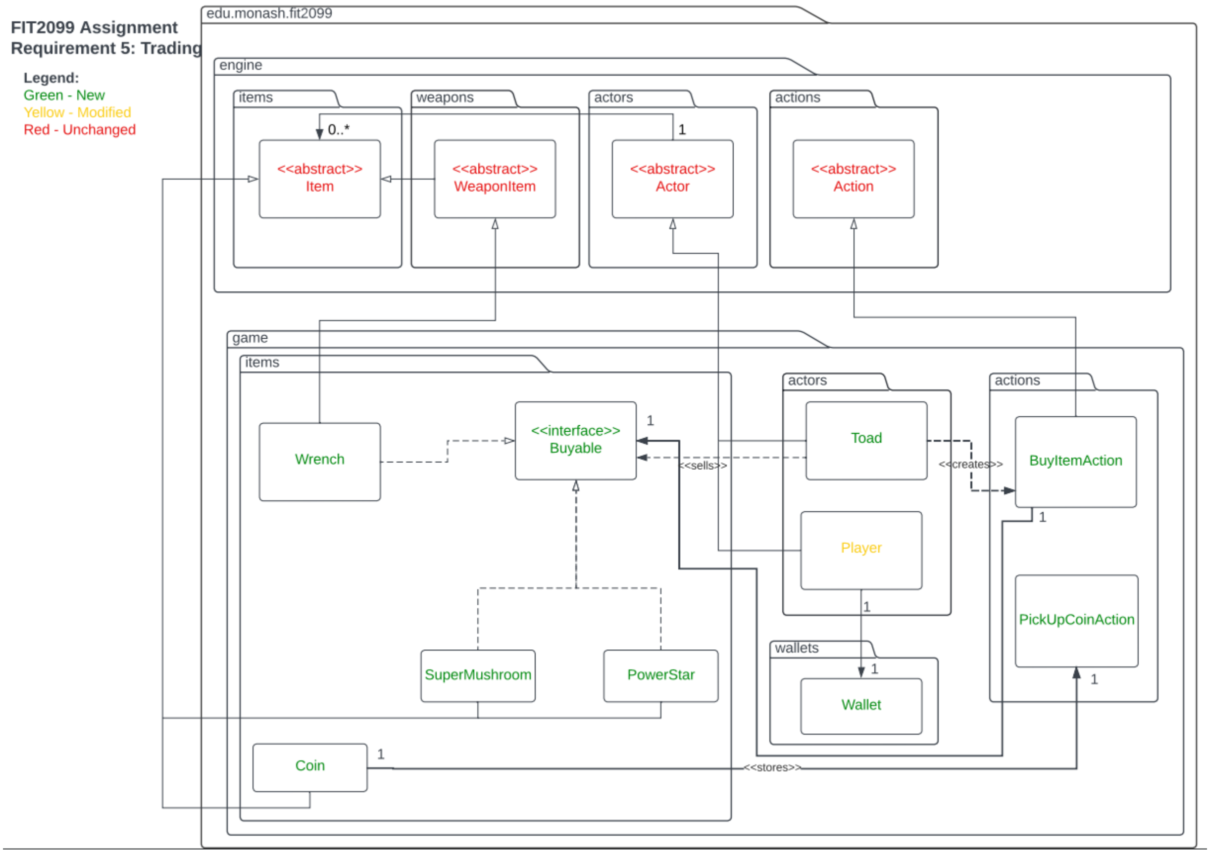


*Diagram 1.5 Trading UML Class Diagram*

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. This is done as both BuyableWeapon and BuyableItem can be used when a trade(purchase) is made, which will be executed by implementing method(s) from the Buyable interface. They are created as two different abstract classes instead of only 1 since the child classes that they have are quite different, having different functionality, Wrench must inherit from WeaponItem, whereas the PowerStar and SuperMushroom must inherit directly from Item. This slightly complicates the class relationships, and may seem like both BuyableItem and BuyableWeapon implement very similar behaviour, but since they could have multiple child classes in future additions to the game, this approach simplifies future modifications and additions, as we can simply create new purchasable power-ups or weapons in the future by extending these abstract classes, rather than needing to dig deeper into the game engine and game class structures to understand how the addition would be made.

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

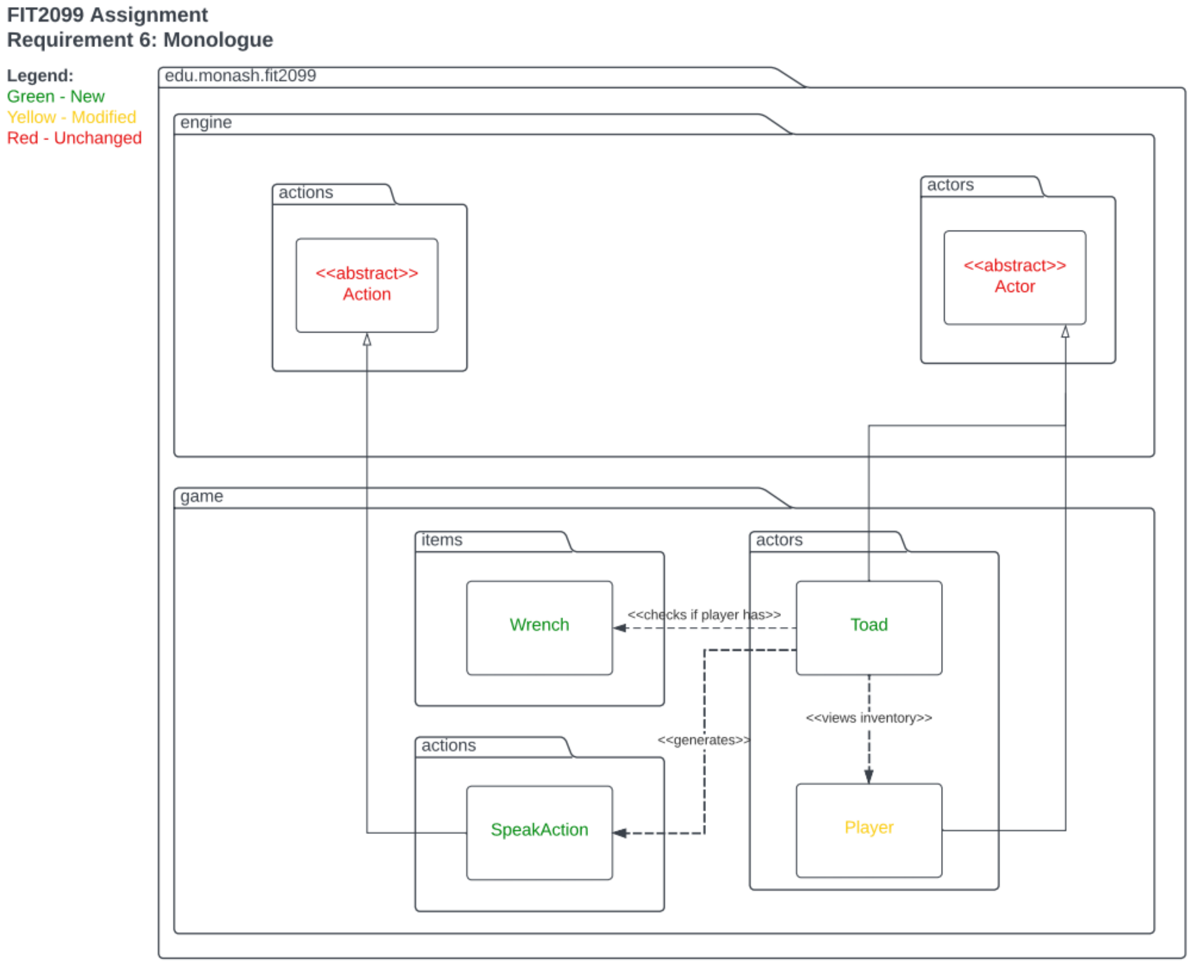


*Diagram 1.6 Monologue UML Class Diagram*

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. Hence, Toad has a dependency on Player, as it must view the inventory (for Wrench) and capabilities (for PowerStar invincible status).

The *SpeakAction* class is created so that whatever line Toad must speak can be printed to the console.

## REQ 7: Reset Game

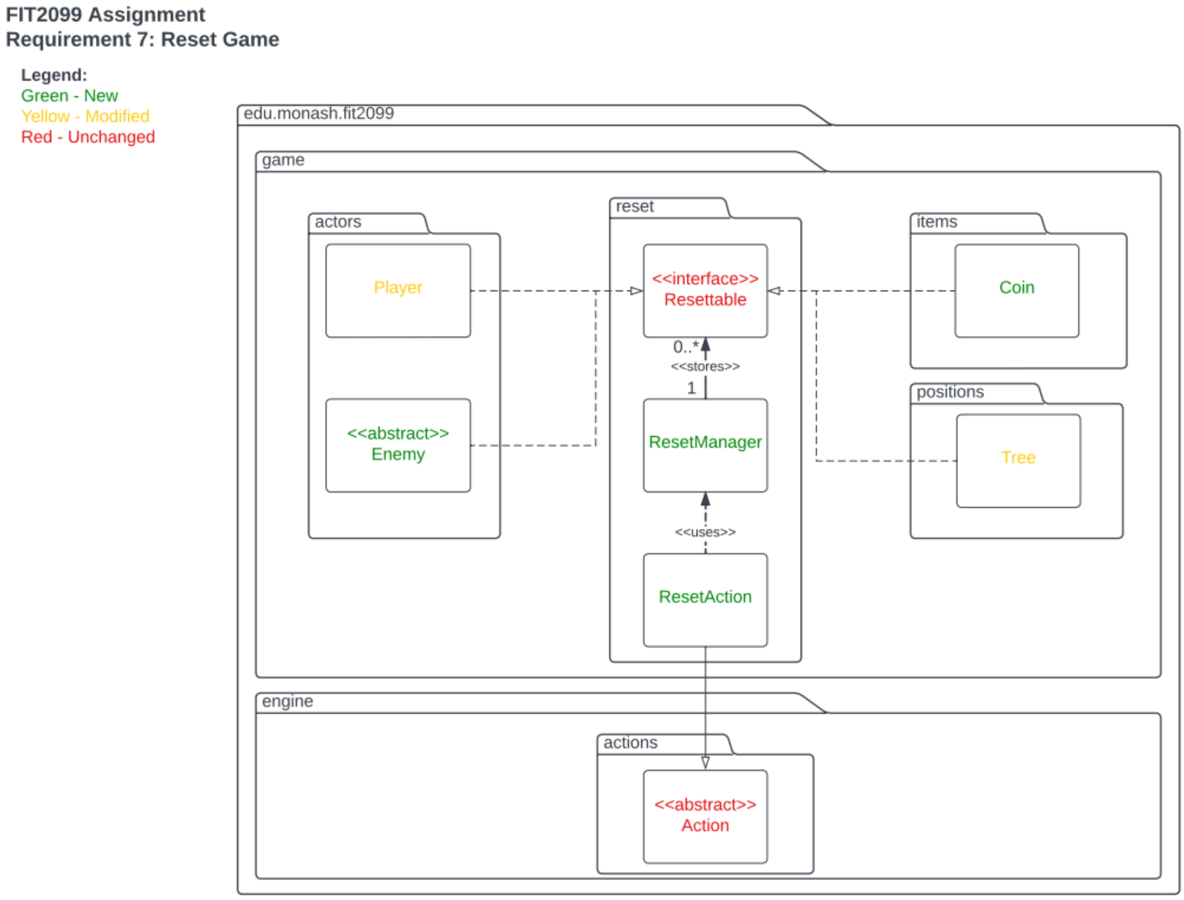


*Diagram 1.7 Reset Game UML Class Diagram*

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.

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

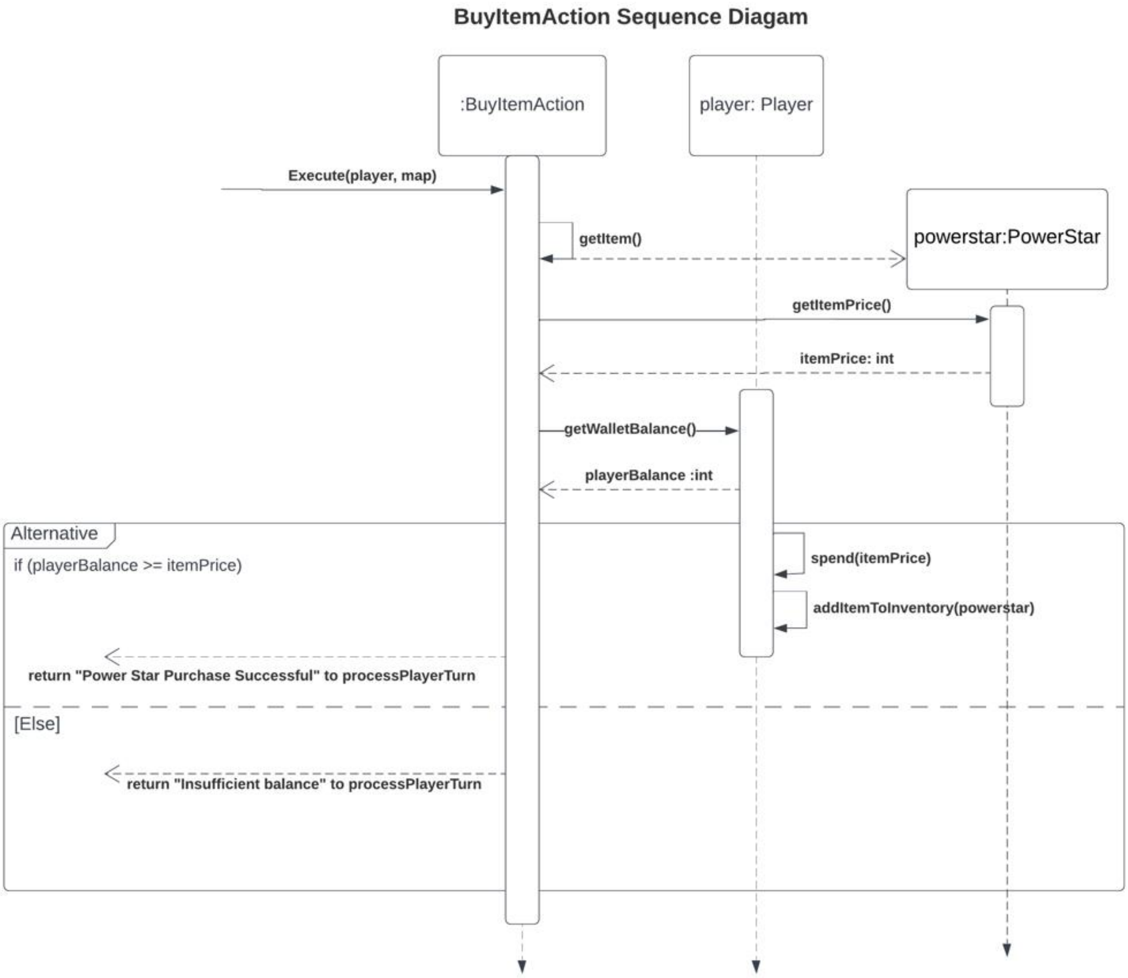


*Diagram 2.1 Trading Sequence Diagram (PowerStar Item)*

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.