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 also would like to achieve 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 support 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 already provided and no modifications are needed.

Yellow : Classes are available, but some modifications are needed such as adding methods to improve the program.

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

While creating the UML diagram, we also make sure that the engine is left untouched and all the classes added are simply used to modify the program to reach a higher level of efficiency instead of changing its functionality.

<u>DESIGN RATIONALE</u>

REQ 1: Trees

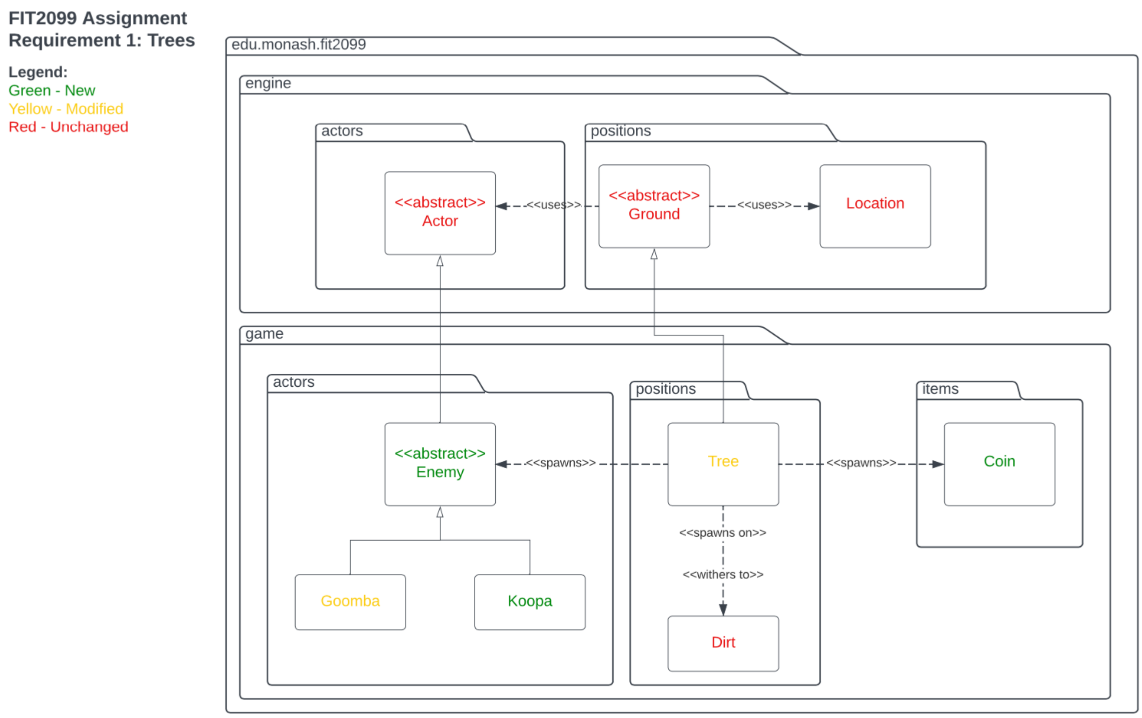

Diagram 1.1 Tree UML Class Diagram

In this UML diagram, it can be seen that *Tree* is an **extended** class of *Ground*. The *Tree* Class is modified in order to create different methods. Each method will be created depending on which stage a certain tree is on (Sprout, Sapling, or Mature). For example, a method **createSprout()** will be introduced in order to check how many turns are there and the dirt space that is still available to create sprout. A method **growSprout()** will be used to keep track of how many turns done since the sprout was there and change to sapling. There is also a dependency between *Tree* and *Dirt* as it is said that a mature tree has a probability of turning to dirt. Hence, dependency will be used by creating a *Dirt* object every time a method of **matureToDirt()** is called.

A *Tree* Class also has a **dependency** on *Coin* Class. This is because there is a probability that a coin is dropped. A *Coin* object will be created (using boolean after checking on the probability) in this method and will be called on each turn.

The *Tree* Class also has a **dependency** with the *Enemy abstract* class. The purpose of doing 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 high-level modules, in this case the Tree, having dependency on low-level modules, which are Goomba and Koopa.

REQ 2: Jump Up

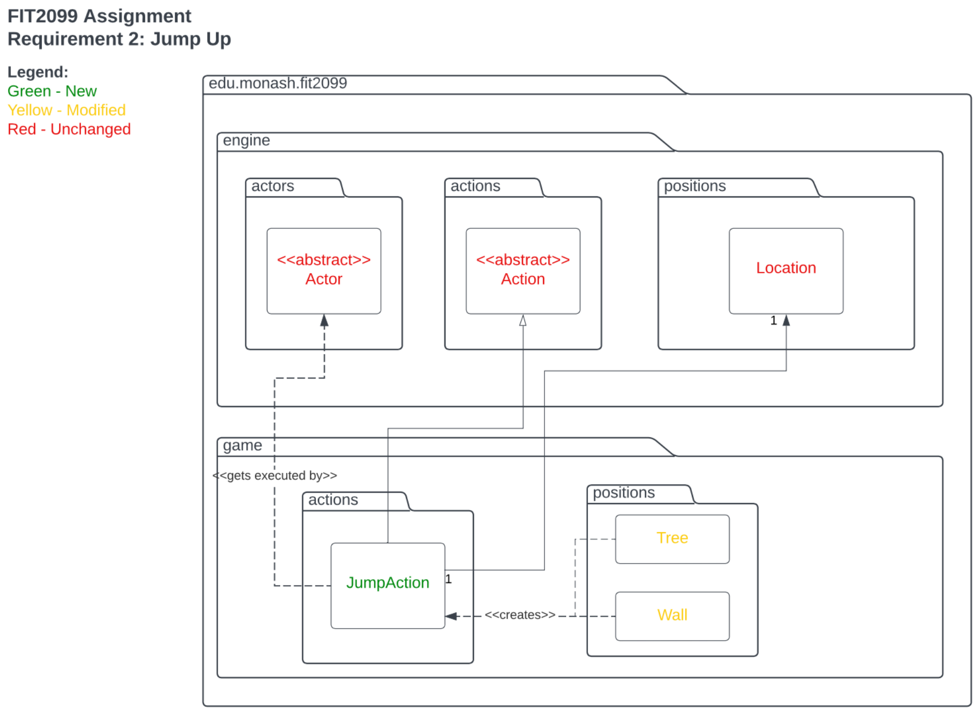

Diagram 1.2 Jump Up UML Class Diagram

In REQ2, the main focus will be the *JumpAction* class. *JumpAction* class will be a child class of *Action* class, since it inherits the behaviour of the parent class. JumpAction class is added in order to obey the ***Single Responsibility Principle (SRP)***. Instead of adding methods in the available class, a new class is created specially to support this functionality. In this UML, it shows that there is a **dependency** between the *JumpAction* class and *Actor abstract* class. The main purpose of doing this is because one of the features mentioned something about the actor and its relation with jumping. Having a **dependency** is required since the program needs to know the personal information of the Actor object to be able to add fall damage for it.

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 any methods of JumpAction to be executed, a Location object should always exist as JumpAction can only be decided once it gets the location 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.

JumpAction also has **dependency** with *Wall* Class and *Tree* Class. In this case, several methods 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 can only be calculated in JumpAction and the exact value can only be known based on which Wall or Tree.

REQ 3: Enemies

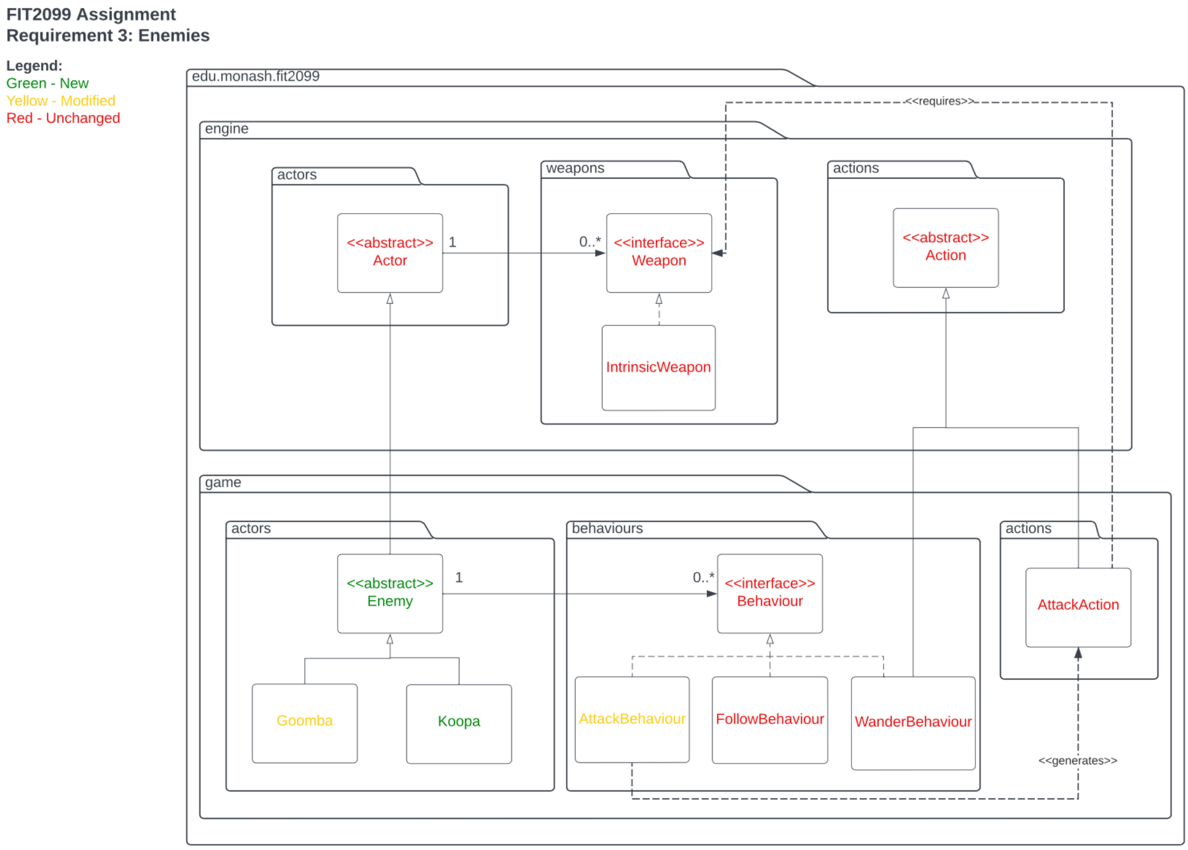

Diagram 1.3 Enemies UML Class Diagram

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, Koopa has its own class since both Koopa and Goomba have different characteristics and behaviours. For example, Goomba starts with 20 HP whereas Koopa starts with 100 HP. Since there are now 2 enemies, an *Enemy abstract* class is created. *Enemy abstract* 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.

The *Enemy* class **extends** from the *Actor abstract* class. This is done as the Enemy class has all the functionality of the Actor class. It is mentioned that when the player stands in the enemy surrounding, it will follow the player. This means that both Goomba and Koopa are alive, and therefore have the functionality of the Actor Class.

There is also a relationship between the *Enemy* class and *Behaviour* interface. In these three classes, a **getAction()** behaviour will be implemented. This function determines what a certain object should perform. The enemies can implement Wander, Follow, and Attack behaviour. —--Lack explanation—------

REQ 4: Magical Items

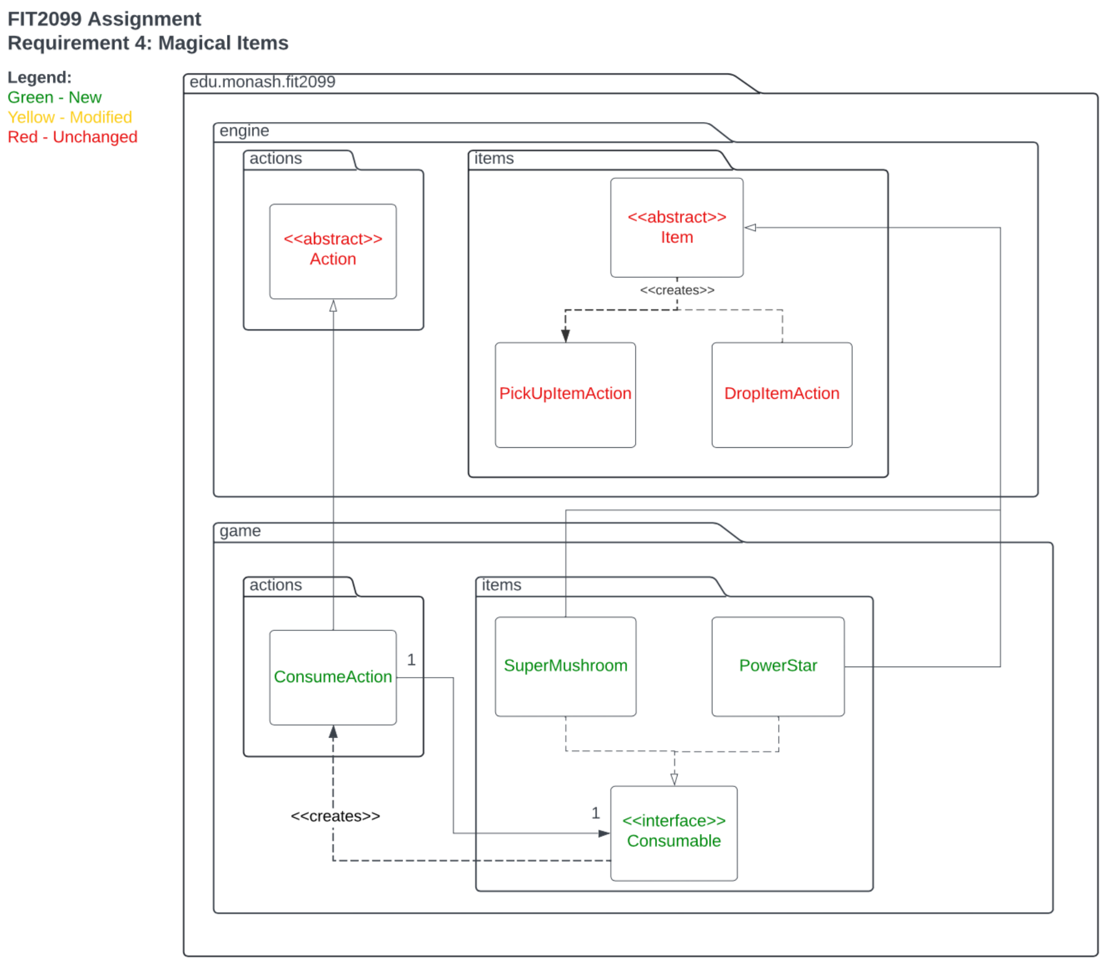

Diagram 1.4 Magical Items UML Class Diagram

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, classes of *SuperMushroom* and *PowerStar* 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 functionality from the *Consumable* interface. Consumable interface is used because both classes need this functionality. —--Explain consumable and consume action—------

REQ 5: Trading

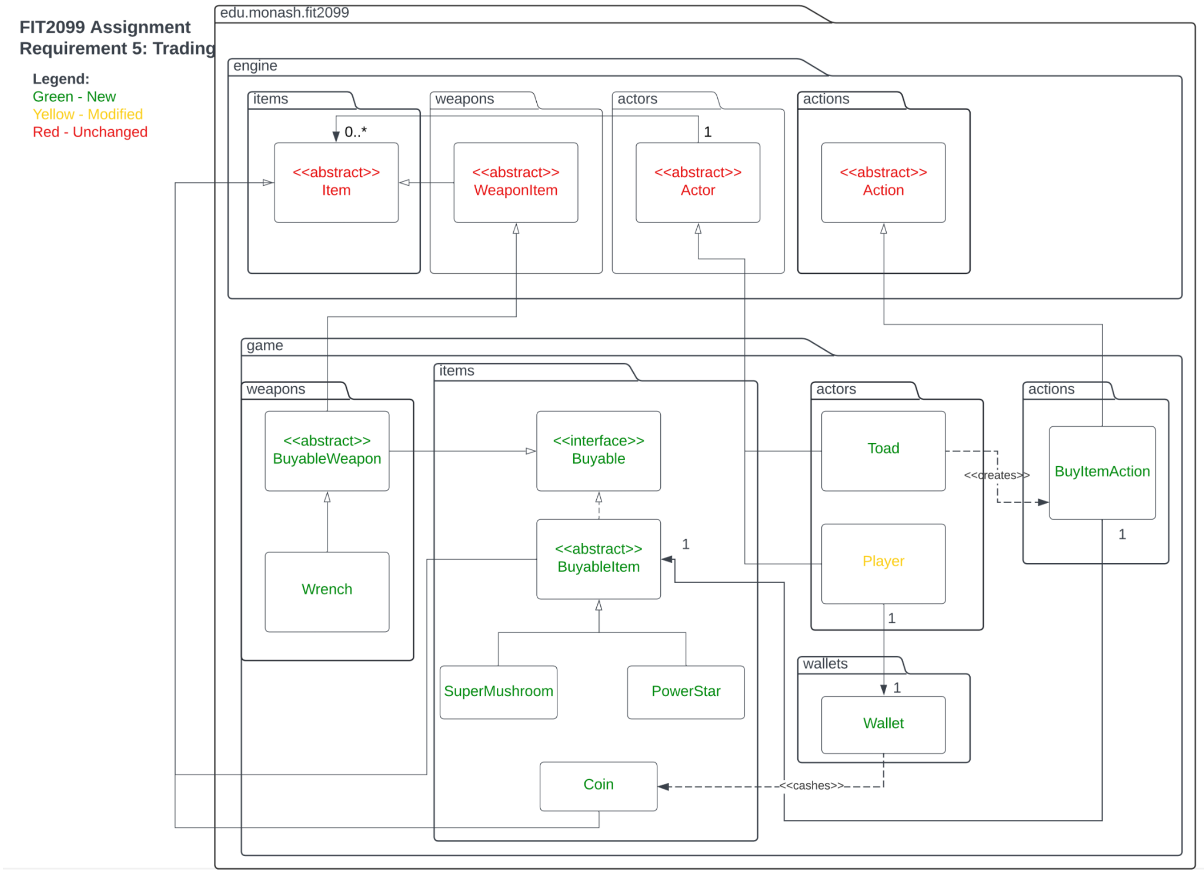

Diagram 1.5 Trading UML Class Diagram

The UML diagram above shows how REQ5 is going to be implemented in the program. Both *BuyableWeapon abstract* class and *BuyableItem abstract* class **implements** the *Buyable* interface. This is done as both BuyableWeapon and BuyableItem can be used when a purchase is done, which will be executed using the method from the Buyable interface. They are created as two different abstract classes instead of only 1 since the child classes that they have are completely different, having different behaviours.

There is a *BuyItemAction* class which has an **association** with *BuyableItems*. This is done as a BuyableItem 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 object in order to buy the items.

A Wallet class is created for the purpose of keeping the coins that the player has collected. This is shown by 1:1 cardinality of an **association** between *Player* to *Wallet*. Since Wallet collects coins, a **dependency** between *Wallet* and *Coin* is created where an instance of coin will be created 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.

Both *Toad* and *Player* are one of the **extended classes** of *Actor*, since both Toad and Player are alive and have the same basic behaviour. This goes the same for *BuyItemAction* which **inherits** from the superclass *Action*, *Coin* and *BuyableItem* also **extends** from *Item* class and also *WeaponItem* **extends** *BuyableWeapon*.

There is also an **association** from *Actor* class to *Item* class. The reason why association is a better option than dependency since the Item might be used on several methods (e.g. adding HP, adding hit points, etc.). Hence, it is better to put it as association since by doing so, it will not limit by allowing only certain methods using it and also prevent from repeating the same piece of code (***Core Design Principle A - Don't repeat yourself)***.

REQ 7: Reset Game

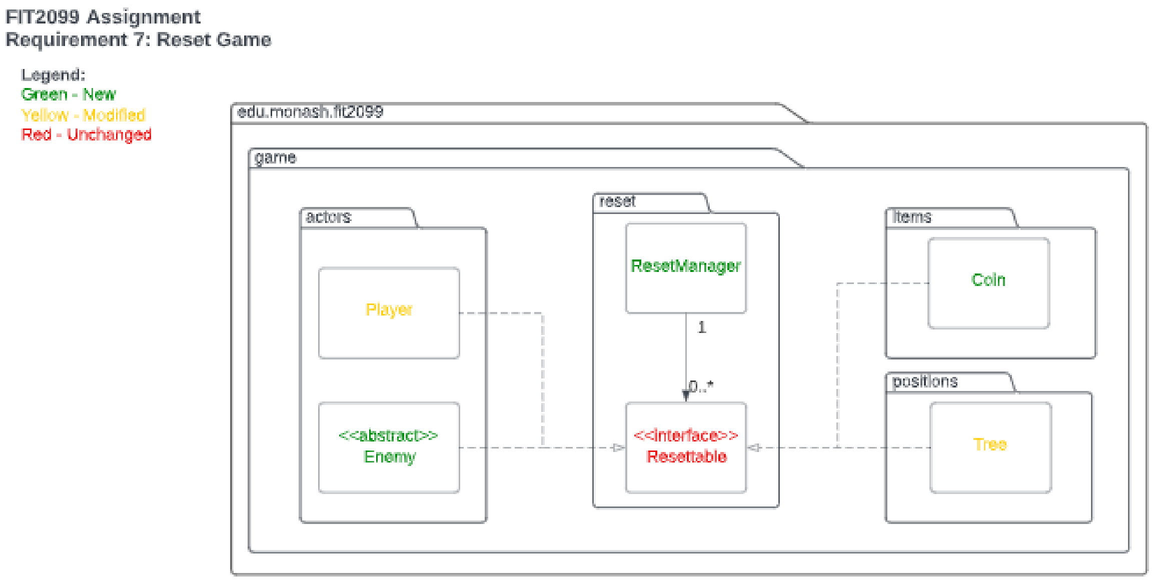

Diagram 1.7 Reset Game UML Class Diagram

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.