FIT2099 ASSIGNMENT 2

Changes to 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

As we coded, we realised that there are some minor things from our original proposed UML design. This report will give a brief explanation on how the UML is in line with the code we wrote and the changes that we made in the UML will be explained elaborately.

REQ 1: Trees

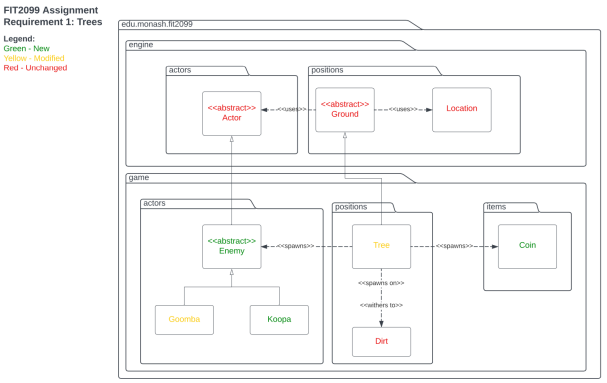

Diagram 1.1.1 Tree UML A1

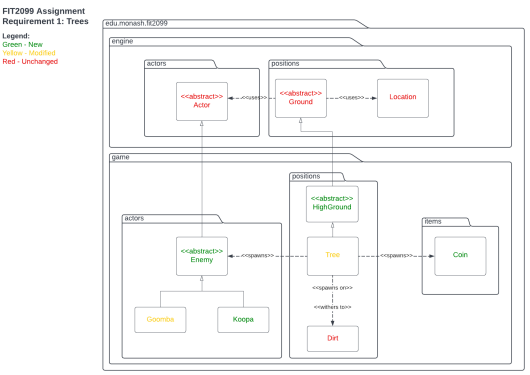

Diagram 1.1.2 Tree UML A2

In the first requirement, **there has been a slight change in the UML** where Tree now inherits from HighGound.

Our group slightly modified the code in the Tree class, which is shown in both UML diagrams. One of the modifications we made is by changing the parent class of the Tree class. Previously, the Tree class was one of the child classes of the Ground Class. We created another abstract class called HighGround that will be useful for the Jump Up requirement (which will be explained later in **REQ 2: Jump Up**). Since Tree is one of the grounds that

can be jumped to by the Player, it becomes the child class of HighGround and HighGround will be a child class of the Ground class.

Coin class is created so that the object coin can be instantiated, as Tree has a probability of spawning coins. The Coin class contains a method to access its value. It also has a resetInstance() method added in order for the coin to be removed from the map and a Coin() constructor which will trigger PickUpCoinAction().

REQ 2: Jump Up

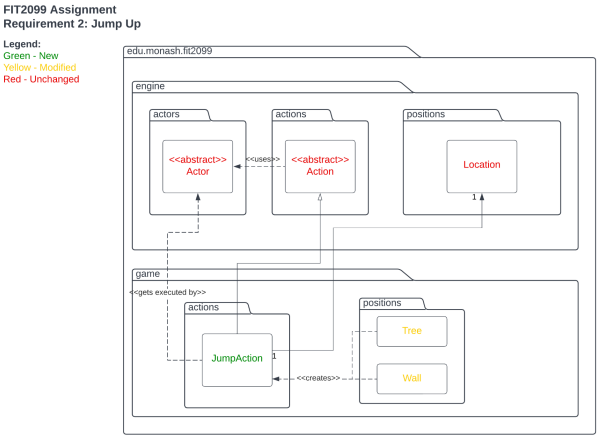

Diagram 1.2.1 Jump Up UML A1

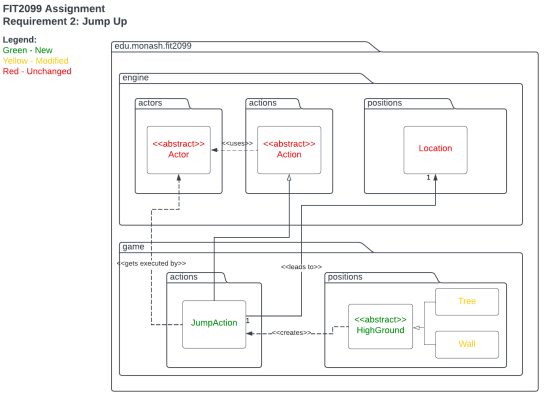

Diagram 1.2.2 Jump Up UML A2

In the second requirement, ***there is a slight change made from the UML*** diagram on the first and second assignment. In Assignment 1, the *UML **does not show** a **HighGround** Class*. Whereas, Assignment 2 shows the HighGround class having a dependency with the JumpAction and an inheritance with the Tree and Wall class. The purpose of adding this class in between is in order to achieve the ***Dependency Inversion Principle (DIP)*** which is preventing JumpAction from having a dependency on many classes. The HighGround class has some abstract methods which will be re-implemented in the child class. For example,

getJumpChanceSuccess() and getFallDamage() will be overridden in both the Wall and Tree class as they both have different jumpChance and fallDamage. The allowableActions() method is overridden inside the HighGround class so that JumpAction can only be executed with certain conditions.

JumpAction class is used in the program in order to allow execution of the jump action. This is created as all the menu that is in the console is executed with the Action class. The execute() method is overridden to adjust it with the desired action.

REQ 3: Enemies

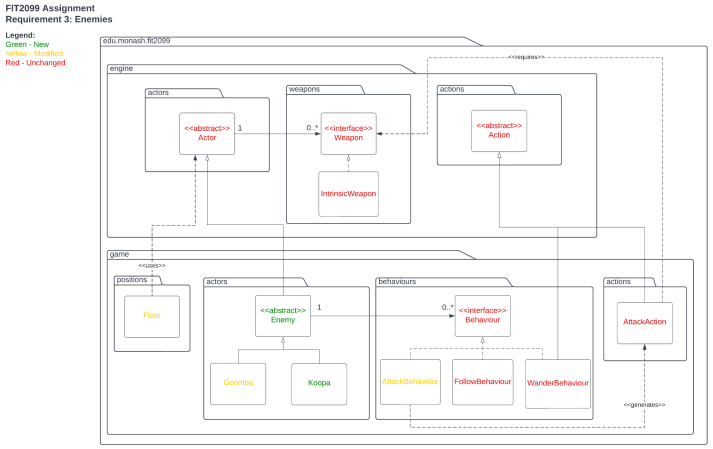

Diagram 1.3.1 Enemies UML A1

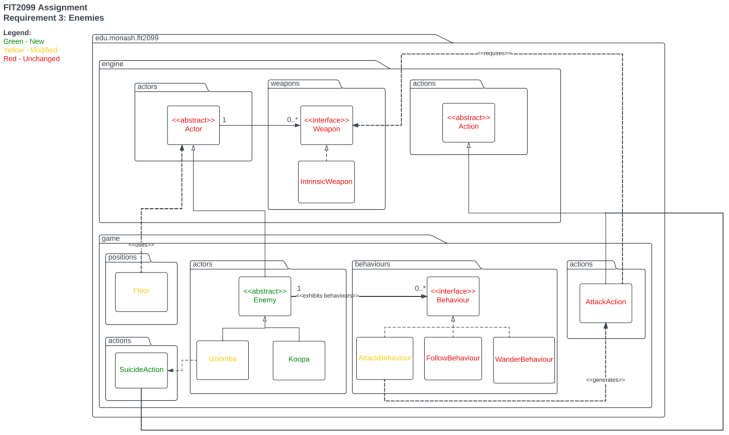

Diagram 1.3.2 Enemies UML A2

In Assignment 2, ***there is a class added to the Enemies UML***. The class that ***previously was not added*** *is the* ***SuicideAction*** *class*. In the requirement, it is mentioned that Goomba has 10% chance of committing suicide. Since this action will only be executed with the help of Action class, a new children class of Action is created, which is SuicideAction. The SuicideAction only removes the Actor in the gameMap, showing that they 'died'. Hence, a

dependency with Goomba Class is created, since an instance of SuicideAction will be called when Goomba commit suicide.

The other changes that are made are to the Enemy and Koopa classes. The Enemy Abstract Class is added since the original code has only 1 enemy, which is Goomba; whereas, there are 2 enemies in the game (with Koopa as another enemy). Goomba class is modified to adapt to the game requirements. The methods contained in Koopa class are similar to what is in Goomba class; some of the slight difference they have is Koopa can go dormant (into its shell) and Goomba cannot.

REQ 4: Magical Items

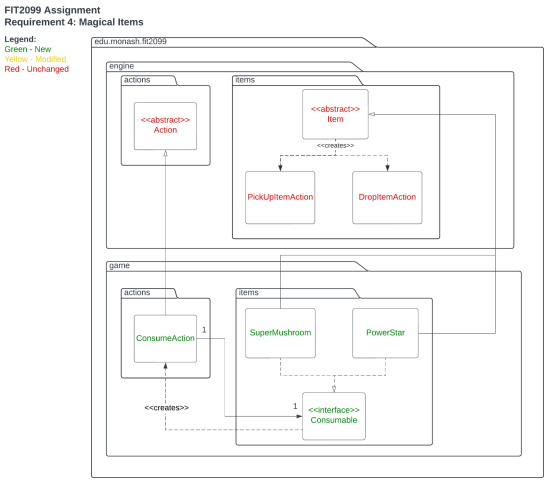

Diagram 1.4.1 Magical Items UML A1

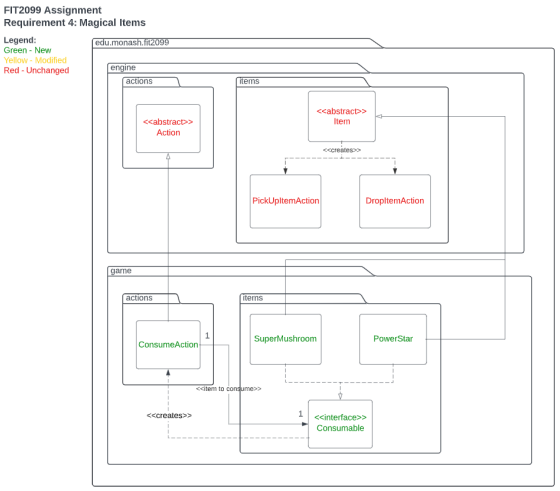

Diagram 1.4.2 Magical Items UML A2

There are ***no changes made in the UML*** from Assignment 1 to Assignment 2. The SuperMuhroom and PowerStar Class are created to allow an instance of these 2 items to be created. It implements an interface Consumable where it contains some methods of consumes and power ups that will be done if it were to be consumed. There is also a dependency of ConsumeAction with the Consumable interface since it will need to know which

consumableItem it should execute. Whereas, there is an association between the ConsumeAction and the Magical Item since the ConsumeAction has an option to be executed when the Magical Items are instantiated.

REQ 5: Trading

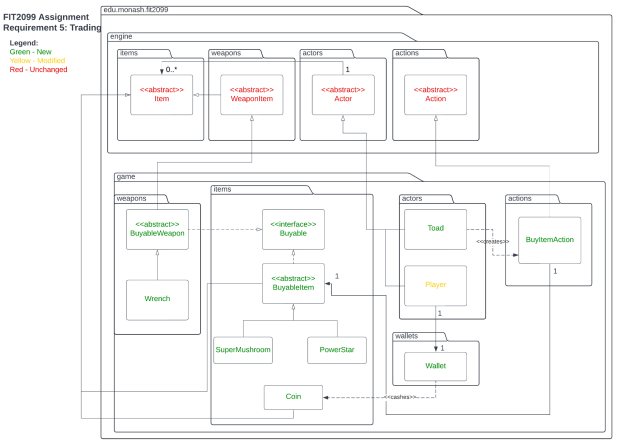

Diagram 1.5.1 Trading UML A1

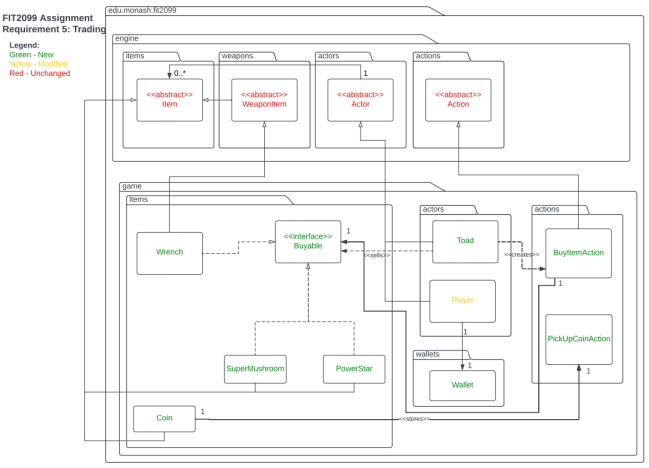

Diagram 1.5.2 Trading UML A2

In the fifth requirement, ***2 classes are deleted from and a class is added to the initial UML diagram*** that has been created. An ***abstract BuyableItem class and BuyableWeapon was removed*** since it just became superfluous. The PowerStar class and SuperMushroom can implement Buyable interface immediately instead of accessing it from BuyableItem class; BuyableItem class will not have any other methods except the ones overridden from the Buyable interface; even so, it will be empty and will be overridden again to the SuperMushroom and PowerStar class. The Buyable interface works similarly like the Consumable interface; hence, it can be implemented in a similar manner by the SuperMushroom class and PowerStar class. This is the similar case with the BuyableWeapon

abstract class, where the Wrench class can just implement Buyable instead of being a child class of BuyableWeapon. Hence, this class is also redundant, and hence removed.

A ***PickUpCoinAction Class is added*** to the code, so it is updated in the UML diagram. This class is another class that inherits from the Action class. Although there is PickUpAction, a PickUpCoinAction is created instead since the picked up coins should not be put in the inventory, instead, its value is added to the player's wallet for purchase. The Coin has an association with the PickUpCoinAction Class, where every time the object is instantiated, it will give the options and execute it if that option is chosen.

REQ 6: Monologue

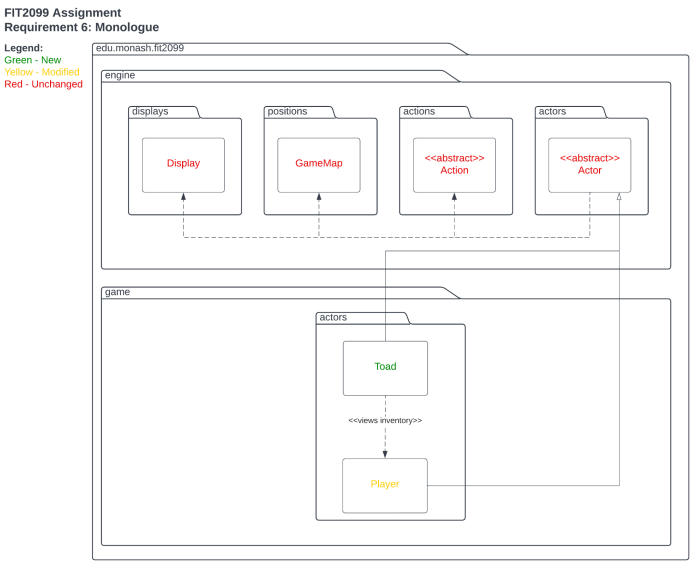

Diagram 1.6.1 Monologue UML A1

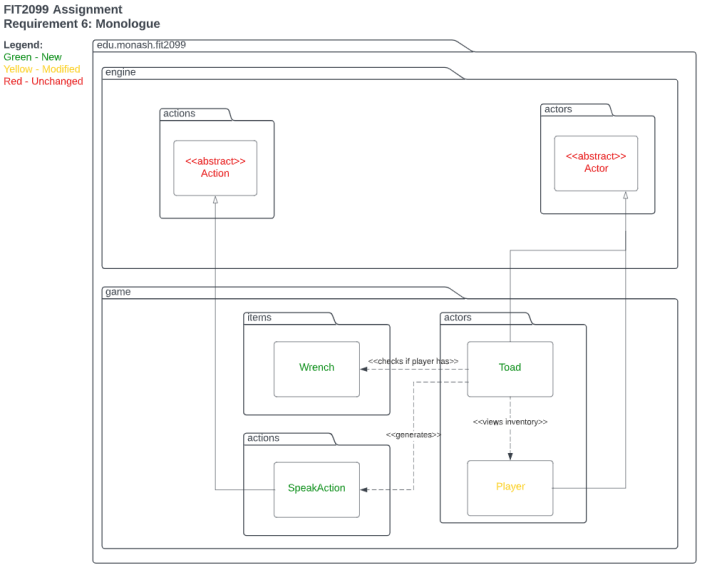

Diagram 1.6.2 Monologue UML A2

In the next requirement, ***there is a minor change made in the UML diagram***. To display the monologue in the console, ***a new SpeakAction Class is created***. This class is used to show

the monologue done by the Toad; hence, there is a dependency of SpeakAction with Toad. The SpeakAction class executes by returning the monologue that is part of the string that Toad can possibly mention. By adding the SpeakAction class, the ***Single Responsibility Principle (SRP)*** is achieved. Initially, it was intended for the monologue lines to be printed directly via the Display class, whereas upon gaining further understanding of the game engine, we made the realisation that we can implement a SpeakAction to return a string of the monologue line, which makes better use of the game engine.

REQ 7: Reset Game

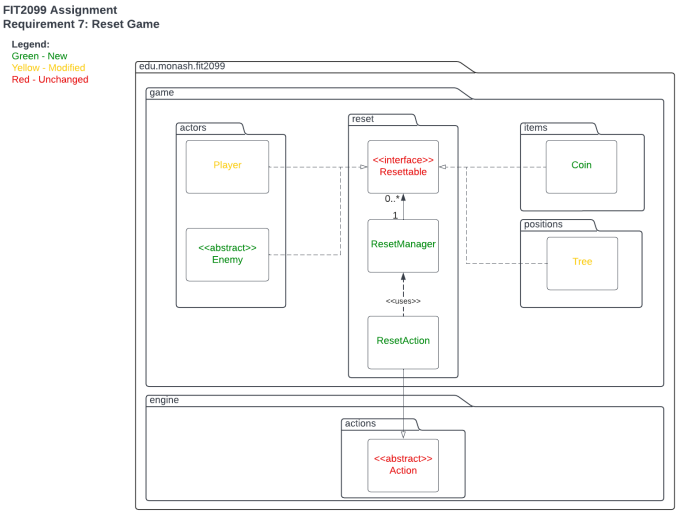

Diagram 1.7.1 Reset Game UML A1

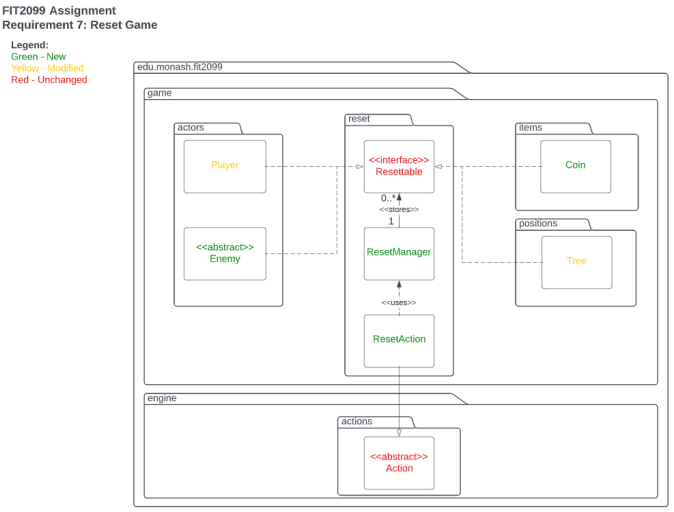

Diagram 1.7.2 Reset Game UML A2

In Requirement 7, there are ***no changes made to the UML diagram*** in Assignment 2 from the UML diagram in Assignment 1. As seen from the UML diagram, the ResetAction is created to allow execution when the user chooses to reset the game. It has a dependency with ResetManager where it will use the method from the ResetManager. Meanwhile, the ResetManager class has an association with the Resettable interface as it will get all the

resetInstance of the classes that have implemented the Resettable interface. The collected resetInstance will be run in the ResetAction. A notable modification made to the code for the Reset requirement was that instead of having the resetInstance method inside the Resettable interface have no parameters, it is now passed the GameMap as a parameter. This allowed for the Enemies to be directly removed from the map (as opposed to setting their health to 0), and allowed for Tree's and Coins to be directly removed from the map locations.

Additional Notes

There is a class that is added for convenience but does not fit in any of the requirements. The class RNG is a class that is used to calculate probabilities. Since there are many requirements requiring probability (such as grow() in Tree, Goomba playTurn() for suicideChance, etc.). Instead of repeating the calculations, an RNG class contains a static method to simplify code in other classes (by not repeating the calculation) and static is used so that an instance of the RNG class does not have to be created every time a calculation needs to be done. The RNG class was omitted from the UML diagrams as it is purely related to the implementation of the features, and is similar to an enum class; it is not necessary to understand the relation *between* game classes.