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.

DESIGN RATIONALE

REQ 1: Trees

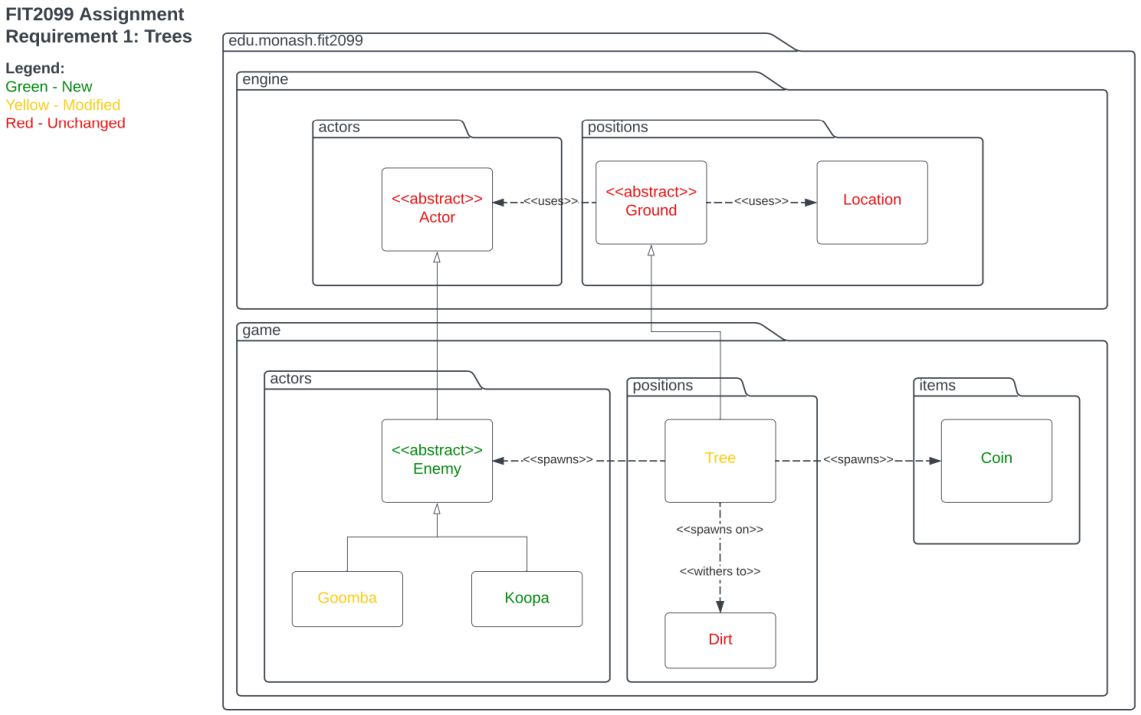

Diagram 1.1 Tree UML

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

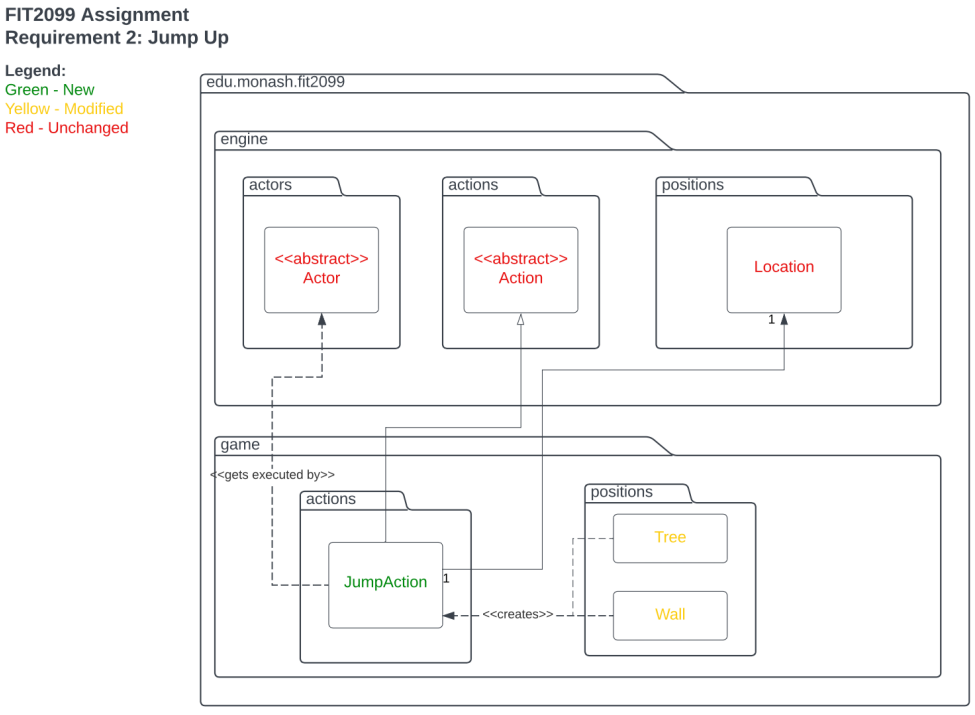

Diagram 1.2 Jump Up UML