FIT2099 Assignment 1

Lab 13 – Group 33

Members: Ong Di Sheng, Kennedy Tan, Mark Manlangit

Design Rationale

REQ1: Let it grow! 🌳

1. **Tree**
   Tree is an abstract class that is an extension from the HighGround abstract class. This class represents a Tree on the map.
2. **Sprout**
   Sprout is an extension of the Tree abstract class. This will be the first type of tree that will be created on the map. This class will be responsible for randomly spawning a Goomba enemy on its location every turn as well as growing into a sapling once it has aged a certain amount.
3. **Sapling**
   Sapling is an extension of the Tree abstract class. This will represent the second stage that a tree can be in on the map. This class will be responsible for randomly spawning a Coin item on its location every turn as well as growing into a mature tree once it has aged a certain amount.
4. **Mature**
   Mature is an extension of the Tree abstract. This will represent the last stage that a tree can be in on the map. This class will be responsible for randomly spawning a Koopa enemy on its location every turn, randomly spawning new sprouts in adjacent fertile tiles every 5 turns and will have a chance every turn to wither and die, which will turn it into dirt.

Don't Repeat Yourself Principle: I created the Tree class as an abstract class in order to define the properties that all Tree subclasses will have. I chose to use an abstract class over an interface to implement the abstraction because all Tree subclasses will be closely related. This will allow Tree subclasses to have many common properties that can be inherited such as the age of a specific Tree as well as the implementation of the reset functionality. All Tree subclasses will therefore inherit these properties, which will reduce duplicated code between the subclasses. This satisfies the DRY principle since we have reduced code redundancy through implementing abstraction with an abstract class.

Single Responsibility Principle: By creating a Tree abstract class and extending this class through subclasses, we avoid creating a Tree GOD class that would have too many responsibilities depending on which stage the Tree might be in. This allows the

subclasses that extend Tree to satisfy the Single Responsibility Principle since we delegate the specific method implementations of each Tree stage into their subclasses.

Open-closed Principle: We may also extend our Tree functionality without modifying the abstract Tree class. For example, if we were to introduce a new stage that a Tree can be in that has its own new behaviours, we would just have to extend the Tree class since this new stage is also a type of Tree. This will allow the new class to use the inherited methods while providing its own implementations. The Tree class has therefore been extended through abstraction while also closing the Tree abstract class for modification, which satisfies the Open-closed Principle.

Liskov Substitution Principle: Since Sprout, Sapling and Mature are subtypes of class Tree, we should be able to replace a Tree with a Sprout, Sapling or Mature tree without disrupting the behaviour of our program. This holds true in our implementation since, for example, all Tree subtypes are able to experience the passage of time through the tick method inherited from Tree while also performing their own specific implementations without problems, thereby satisfying the Liskov Substitution Principle.

REQ2: Jump Up, Super Star! 🌟

1. **JumpAction**
   JumpAction is a class that is an extension from the Action abstract class. This class allows an actor to attempt a jump to a high ground. The chances that an attempt is successful is dependent on the type of terrain the high ground is. It then returns a message of whether the jump was successful or not.
2. **HighGround**
   HighGround is an abstract class that is an extension from the Ground abstract class. This class represents a type of ground that can be jumped to. This class is responsible for performing the actual jump onto this HighGround during a jump action where a successful attempt will move the actor to this particular HighGround's location and an unsuccessful attempt will hurt the player with the relevant fall damage. It is also responsible for giving an actor the option to perform a jump action onto this HighGround as well as turning into Dirt and spawning a coin at its location when the player is invincible.

Don't Repeat Yourself Principle: I created the HighGround class as an abstract class so that the child classes that extend HighGround can utilise the methods defined in the HighGround class. Since most of the logic to perform a jump, add jump actions and handle the destruction of High Grounds is inherited from the HighGround class, we do not need to repeat the implementation in our child classes, which also reduces the amount of code required.

Open-closed Principle: We may also extend our HighGround functionality without modifying the abstract HighGround class. For example, if we were to introduce a new type of high ground that can be jumped, we would just have to extend the HighGround class since this new high ground is also a type of HighGround. This will allow the new class to use the inherited methods while providing its own implementations for specific jump chances and fall damage values. The HighGround class therefore has been extended to allow for a new high ground to be jumped while also closing the HighGround abstract class for modification, which satisfies the Open-closed Principle.

Liskov Substitution Principle: Since HighGround is a subtype of class Ground, we should be able to replace a Ground with a HighGround without disrupting the behaviour of our program. This holds true in our implementation since the Ground tick method and allowableActions method can be used on a HighGround object without any problems.

REQ3: Enemies

a) **Enemy**

Enemy is a new abstract class that extends from Actor abstract class. It contains the general characteristics of the actors that can deal damage to the player. With abstraction, we managed to extend the code functionality without modifying the existing Enemy class. This follows the **Open-Closed principle** which has been defined as, classes should be open for extension but closed for modification. In Assignment 2, our implementation follows the design in Assignment 1. Currently we have two enemies, which are Goomba and Koopa. Goomba and Koopa inherit from the Enemy abstract class. In Enemy class's constructor, WanderBehaviour and AttackBehaviour are added into the Behaviours Hash map in the Enemy class. As they are the same characteristics of all enemies.

This also follows the principle of **Don't Repeat Yourself**. In the future, if the enemies have new common behaviours, we just have to add it in the Enemy abstract class, instead of repeating them in Goomba, Koopa and other new enemies' classes. For their own special characteristics, they should be added in their own classes. For instance, Koopa will be changed to dormant state if defeated.

b) **Behaviour**

Behaviour is an interface that is included in the based code which determines the current behaviour of the enemy. It is split into three behaviours which are FollowBehaviour, WanderBehaviour and AttackBehaviour. So that the enemy can only implement the methods they need at that specific situation. This follows the **Interface Segregation Principle** which is defined as a larger interface should be split into smaller ones.

c) **Koopa**

The implementation of the Koopa class is the same as what we design in Assignment 1. Koopa is an extension of Enemy class, which extends from Actor class. Koopa is one of the enemies in this system, but it has different characteristics compared to Goomba. Koopa will go to dormant state by using Status enumeration and it is not allowed to move around. When it is destroyed by the player using a wrench, it will drop a SuperMushroom by DropItemAction. These characteristics make it much different compared to Goomba. So, instead of putting all types of enemies inside Enemy class and using if statements to implement their characteristics, we create classes for every enemy and extends from Enemy class. In this way, also their general characteristic is defined in the Enemy abstract class. In the future, if we need to add enemies to the system, we simply just create classes for them and extend them from the Enemy abstract class.

This satisfied the solid principle, **Open-Closed Principle**. Enemy is the abstract class closed for modification, special characteristics for enemies should only present in their own classes.

d) **DestroyAction**

This is a new class called DestroyAction is newly created in Assignment 2, which extends from the Action in the engine code. In Assignment 2, this action is used to destroy Koopa's shell when Koopa is in dormant state and drop SuperMushroom at that specific location. This action will have its own option for the user to choose to destroy Koopa's shell if the player has a wrench in their inventory. Then, it will display specific results to the console.

This follows the **Single Responsibility Principle**, which defines a class should have only one job. In this case, DestroyAction is only used for the player to destroy the enemies in a certain state that they cannot be attacked but can be destroyed. AttackAction is specifically used for an actor to attack a target, while DestoryAction is used for an actor to destroy a target in a certain state that they cannot be attacked but can be destroyed to turn into a different state.

REQ4: Magical Items

a) **ConsumeAction**

ConsumeAction was implemented as designed in Assignment 1. ConsumeAction is an extension of Action abstract class. This method is invoked when the player wants to

consume the Items that implements Consumable interface in the inventory. Each item consumed has their own effects on the player. Player can obtain capabilities from these items by setting some status to the player. For instance, the player can get an Invincible status once they consume the power star, and they will become invulnerable for a period.

By creating the ConsumeAction method, we follow the principle **Single Responsibility Principle**, which defines as, each class should be responsible for a single part or functionality of the system.

In this case, ConsumeAction only focuses specifically on the consumption of consumable items process. Instead of having all actions in one Action class, we separate ConsumeAction from other actions, which have their own implementation. When the player picks up the magical items on the ground or bought the item from Toad, the menu will have the option to consume them. Then print out the description showing the player consumes magical items in the console.

b) **ConsumableItemManager**

ConsumableItemManager is a class newly created in Assignment 2. This class keeps track of and stores all the items that can be consumed. This class consists of an array list, consumableList of type Consumable, which is an interface implemented by items that can be consumed by the player. It has a getConsumableItem method that loops through the consumableList when the player intends to consume an item from the inventory.

c) **Consumable**

Consumable interface is implemented as designed in Assignment 1. Consumable is an interface implemented by the items that can be consumed by the player. In this interface, we will have the consumeMagicalItems method and addToConsumableItemManager method. As every consumable item has to use these two methods, so instead of creating these two methods in all consumable items' classes, we simply just implement them from this interface, then we just override the methods from the Consumable interface. Instead of having an Items interface and creating a consumable method in it, and for those items that can be consumed implements the item interface and invokes the consumable method in it. We create a Consumable interface, so for those items can be consumed will just have to implements Consumable interface.

This satisfied the **Interface Segregation Principle** which defines as larger interfaces should be split into smaller ones. Classes that need the capability should just implements from it.

REQ5: Trading

a) PickUpCoinAction

PickUpCoinAction is an extension from PickUpItemAction. This class allows an Actor (e.g. Player) to pick up a Coin if the current location where the Actor is standing on has a Coin. It has Coin as its attribute because when the Player executes this action, the Coin must be removed from the location and the value of the Coin which can be obtained by calling getValue method will be added to the Player's balance in the Wallet.

Each instance of PickUpCoinAction can only have exactly one Coin stored as its attribute, so the multiplicity of PickUpCoinAction to Coin is one.

The reason why we do not choose to use PickUpItemAction to pick up the coin is because the PickUpItemAction will store the item in the actor's inventory. In this case, we only want to get the value of the coin being picked up and add the value into the balance in the Wallet but not store the coin in the actor's inventory. Due to a different implementation for the action of picking up a coin compared to other items, we choose to create a new class PickUpCoinAction that inherits from PickUpItemAction to deal with coin object only.

PickUpCoinAction satisfies the **Single Responsibility Principle**, because it only picks up a coin and not deals with any other items. This is to separate the action of coin being picked up from other items, which have their own implementations.

b) Tradable

Tradable is an interface in which all items that can be bought from the Toad such as Wrench, SuperMushroom and PowerStar should implement. In this interface, there are getPrice() and (UPDATED for A2) addCapabilityDuringTrading() method signature without the body and the concrete implementation for this method must be done by every Tradable items. This is

because every Tradable items have their own price and (UPDATED for A2) capabilities that should be added specific to the TradeAction (e.g. Power Star and Super Mushroom which are bought from the Toad should not be dropped to the ground). The price should be made as a constant in each of the Tradable item class to **avoid excessive use of literals** and this would make the understanding of the code and maintenance much easier in the future.

Every class that implements Tradable interface will need to override the getPrice() and addCapabilityDuringTrading() method. This would satisfy **Open-Closed Principle** as each of the Tradable item can have their own price or specific capability that should be added in TradeAction without modifying the existing code.

c) TradeAction

TradeAction is an extension from Action abstract class. This class allows the Player to buy Tradable items such as Wrench, SuperMushroom and PowerStar from the Toad. (UPDATED for A2) It stores an item of type Tradable as its attribute so that it is able to get the price of the item and add specific capability to the item during trading process by using getPrice() and addCapabilityDuringTrading() method respectively.

When the Player executes this action, it will first get the balance of the Player in the Wallet and check whether the Player is able to buy the item by accessing its price through the getPrice() method. If the transaction is successful, the item will be added with some capabilities specific to the TradeAction before being inserted to the Player's inventory and the balance in the Wallet will be deducted too, otherwise an error message will be displayed to the user.

TradeAction satisfies the **Single Responsibility Principle**, because it only focuses on the trading process between the Player and the Toad. This is to separate the action of trading from other actions, which have their own implementations.

d) Wallet

Wallet is a class that holds an integer attribute called balance which is the amount of money that Player has currently. When the Player picks up a Coin, the value of the Coin will be added to the balance of the Wallet. The balance attribute should be made as static since it

will be used in the static getBalance() and subtractBalance() method. This means that the balance of the Player can be retrieved without instantiating Wallet instance by using the static getBalance() method during TradeAction. The same goes for subtractBalance() when the item is purchased successfully.

Wallet satisfies the **Single Responsibility Principle**, because it only deals with the current balance of the Player and not dealing with any other items.

REQ6: Monologue

a) SpeakAction

SpeakAction is an extension from Action abstract class. SpeakAction will store an Actor object as its attribute so that it can access the name of the actor performing the SpeakAction / speaking the sentence. Each SpeakAction is responsible for handling an Actor object, so the multiplicity for SpeakAction to Actor is one. Besides that, there is a constant string array storing all the possible sentences that can be spoken by the Toad. When executed, it will undergo multiple checks before displaying the message to the user. The first check will determine whether the Player holds a Wrench in the inventory. If the Wrench is in the inventory, a constant number using static final keywords can be used to indicate that the first message should not be displayed by adding to the ArrayList of Integer which stores the index of the messages that should be excluded. The same goes for the situation when the Player is in the Power Star status. After the checks, we will call to the getRandomMsg() method by passing the previous ArrayList so that the message can be displayed correctly based on the capabilities of the Player. The use of constant in SpeakAction class allows us to **avoid excessive use of literals**.

SpeakAction satisfies the **Single Responsibility Principle**, because it only focuses on the speaking process between the Toad and the Player. This is to separate the action of speaking from other actions, which have their own implementations.

REQ7: Reset Game

a) ResetAction

ResetAction is an extension from Action abstract class. ResetAction will call upon the ResetManager.getInstance().run() in the execute method to reset the game. Therefore, every

Resettable instances that are added to the resettable list will execute their own resetInstance() method.

(UPDATED for A2) Since ResetAction can only be executed once throughout the game, therefore a static boolean variable named resetFlag (originally set to false) will be set to true in the execute method to indicate that the reset command has been executed and should not be available in the console anymore.

ResetAction satisfies the **Single Responsibility Principle**, because it only focuses on the reset process when the user enters hotkey 'r'. This is to separate the action of reset from other actions, which have their own implementations.

b) Resettable

Resettable is an interface where Enemy, Player, Coin and Tree should implement since they are involved during the reset process. Every class that implements the Resettable interface should override resetInstance() method to reset their abilities, attributes or items. Since Enemy, Player, Coin and Tree would need to implement the Resettable interface that we have just created, therefore, we can have one ArrayList of type Resettable in ResetManager instead of having multiple ArrayList (1 for Enemy, 1 for Player and so on). Thus, the **Dependency Inversion Principle** is fulfilled.