FIT2099 Assignment 1 Design Rationale

- Linden Beaumont, Ricky Zhang & Rebekah Fullard

REQ1

How it works:

The tree class extends the Ground class while the Tree class itself is an abstract class. There are three child classes of Tree; Sprout, Sapling, and Mature. Each of these child classes have a dependency with another class in accordance with game functionality; Sprout has a dependency with the Goomba class, Sapling has a dependency with the Coin class and Mature has a dependency with Koopa.

Tree extends the Ground class because it requires methods from the Ground class (such as tick).
The three classes that extend Tree (Sprout, Sapling, and Mature) work in a similar manner, each of the child classes needs some base methods from Tree but require their own methods to carry out game functionality. Sprout will be able to create a Goomba object using the Goomba's constructor, similarly, Sapling will be able to create the Coin and Mature will be able to construct a Koopa object. Mature has functionality where it will be replaced by a Dirt object randomly so Mature will be able to also create a Dirt object to replace itself at its position.

The Sprout and Mature classes also have a dependency on the Locations class so that they may use the containsAnActor(); method in locations to determine if an Actor is on the same location as the Sprout/Mature, and if there is, don't spawn a Goomba/Koopa.

Why it works that way:

Each of the three child classes of Tree are in line with the Single Responsibility principle because each of the child classes have their own unique functionality and responsibilities (i.e: only Sprout can instantiate Goomba, only Sapling can instantiate Coin). It would be possible to design the program so that Sprout, Sapling, and Mature are subclasses of Ground, however it makes sense to include them as subclasses of Tree because this allows them to implement common methods from Tree, while also providing their own specific functionality and responsibilities.
This design choice also follows the Dependency Inversion principle by having Tree as an abstract class of the ground abstract class.

REQ2

How it works:

The Player will be able to use the JumpAction class by instantiating the class and using its method.
The JumpAction class has an association with Location. Through a Location object, a method within the JumpAction class can determine the type of Ground object at the Location. This can be done by utilising the Location class' getGround() method. By determining the type of Ground (i.e: wall/tree type), the appropriate behaviour of the jump can be carried out and the Actor can either be moved to the new Location or stay in the same Location and lose some health (fall damage).

The SuperMushroom class utilises the Player's capabilitySet attribute to make the Player 'capable' of having a 100% success rate for jumps with no fall damage.

Why it works that way:

This implementation follows the Single Responsibility principle. JumpAction and only the JumpAction class carries out the functionality required for a jump. For example, only the JumpAction class carries out the functionality of determining the success rate of a jump and moving the player to the otherwise unreachable Location or removing hit points.
The Open-Closed principle is also present in this design by having JumpAction following the single responsibility principle. For example, if a new subclass of Ground that can be jumped to is added to the game, only the JumpAction class will be modified to accommodate this change.

REQ3

How it works:

Goomba is a subclass of Actor. Goomba's base HP value will be changed so that it will always start with 20HP. Goomba objects will override the getIntrinsicWeapon() method in Actor so that the damage of the intrinsic weapon is 10 and the verb is "kicks". Each time a new turn ends, a method from Goomba will run that will have a 10% chance to remove the goomba from the map.

Koopa is a subclass of Actor. The constructor for Koopa will set its HP value to 100. Koopa will have a boolean attribute that represents whether the Koopa is dormant or not. The execute() method in AttackActions will be updated to check whether the target being attacked is a Koopa.

For implementing the ability of dormant Koopas to only be hurt with a Wrench, when a dormant Koopa is attacked, if the attack hits and the weapon used was not a Wrench, a

message stating that the attack was ineffective will be displayed. If the weapon was a Wrench, the addItem() method for that location will be called, passing SuperMushroom as the item to add to that location and the Koopa will be removed.

Why it works that way:

This design approach aims to follow the Open-Closed principle. Rather than creating an Enemy class that, for example, has a name attribute to determine what type the enemy is (if this approach was used the Enemy class would need to be modified any time a new enemy type was added to the game) this approach allows for extension as future enemies can be created as subclasses of Actor also.

REQ4

How it works:

PowerStar and SuperMushroom are subclasses of Item and implement the Consumable interface. The Consumable interface will state a method to be called when a consumable item is used. This method can then be implemented in each Item subclass that implements the Consumable interface. This method will implement the effects of consuming that particular Item (e.g. PowerStar providing the effect of invincibility to the Player). UseItemAction will have an attribute for the Actor who is using the item, and will be set up so that the effects of using the Item are applied to that Actor.

Why it works that way:

This design approach follows the Open-Closed principle because the PowerStar and SuperMushroom classes extend the Item class. Any more items that are added to the game can also extend this class without any need to modify the Item class.

The Consumable interface helps this design to be able to be extended because if any more Items that are consumable are added to the game they can also implement the methods from this interface. This also helps provide clarity to the code.

REQ5

How it works:

Coin is a subclass of Item. This class will store one attribute, and Integer ‘coinValue’. PickUpCoinAction is a subclass of Action. This class will override the execute() method from Action. The method will be implemented such that it updates an Integer attribute of Player called ‘wallet’. This attribute will track the total amount of money the Player has at any given time. The Coin will not be stored in the Player’s inventory.

Toad is a subclass of Actor. Toad has an ArrayList<Item> attribute that contains each of the Items he can sell. When the Player takes the BuyAction, that action ‘targets’ Toad and the relevant Item from his list of Items that he sells is appended to the Player’s inventory.

Sapling has a Coin attribute. This is set to null when the Sapling is instantiated (indicating that the Sapling has no currently spawned Coins). With each turn, there will be a check to see if the Coin attribute in each Sapling is null. If it is not null, a coin cannot spawn that turn. If it is null, a coin may randomly be spawned.

Wrench, SuperMushroom, and PowerStar implement the Sellable interface. This interface includes a method for setting the price of the Items.

Why it works that way:

The implementation of this requirement attempts to keep the principle of Single Responsibility. The task of the PickUpCoinAction class could, in theory, be made a part of the PickUpItemAction because the Coin is an Item. However, approaching it this way would violate the Single Responsibility Principle because the result of picking up a Coin is different to the result of picking up any other Item (not added to the inventory, value($) updates an attribute held by the player).

As such, it makes sense to create a separate class to handle picking up Coins so that PickUpItemAction can continue to do its job and PickUpCoinAction can have a specific job also.

The Sellable interface helps this design to be able to be extended because if any more Items that are sellable are added to the game they can also implement this method. This also helps provide clarity to the code.

REQ6

How it works:

ConverseAction is a subclass of Action. This class involves two Actors. The one who is initiating the conversation (i.e. the Player, an Actor), and the one who will speak (i.e. Toad, an Actor subclass implementing CanSpeak).

Toad will implement the CanSpeak interface. It will provide the implementation for the method speak(), which will define the process to determine which of Toad’s lines is randomly said when the Player takes the ConverseAction. Toad’s lines will be stored in an ArrayList<String> attribute of Toad. The speak() method will check the Player’s inventory for the Wrench and check the Player to see if they’re under the effects of the PowerStar. This information will inform which options from Toad’s stored lines can be chosen.

Why it works that way:

It would be easy to simply implement an attribute and a method in Toad that allows him to speak, however, if we need to expand the program later to include more Actors that can speak, it would become difficult to differentiate between Actors that can or cannot speak.

As such, it makes sense to include an interface for speaking Actors. This allows the subclasses to implement the speak() method according to their own restrictions and makes it clear in the code which Actors have the ability to speak.

The ConverseAction class can also be made clearer by involving one regular Actor (i.e. Player) who takes the action, and one CanSpeak Actor (i.e. Toad) who speaks.

REQ7

How it works:

ResetAction is a subclass of Action. When the Player chooses to take ResetAction, the execute() method is called. This method will call the run() method of the ResetManager. In this method, the ResetManager will loop through each Resettable instance in its stored List<Resettable>. For each object, the resetInstance() method will be called. In each class that implements Resettable, the implementation of this method will be defined.

- For enemies (Koopa and Goomba), this method will call removeActor().
- For Player, this method will retrieve the capabilitySet of the Player, and remove any capabilities relevant to the SuperMushroom or PowerStar.
- For Coins, this method will call removeItem().
- For Trees, this method will call setGround(Dirt).

Why it works that way:

Having Tree, Coin, Koopa, Gooma, and Player implement Resettable follows the Open-Closed principle because it ensures that ResetManager doesn't need to be edited each time a new type of resettable class is added to the program. Resettable can be implemented in as many classes as needed without any need to modify ResetManager.

General

The overall approach to this design aims to follow the Interface Segregation principle because all classes that implement an interface will make meaningful use of all methods declared in the interface.