FIT2099 Assignment 2 Design Rationale

- Linden Beaumont, Ricky Zhang & Rebekah Fullard

REQ1

How it works:

Tree is an abstract class that has three child classes, Sprout, Sapling, and Mature. The abstract class itself has attributes that contain the x and y values and the amount of turns a tree has played for; these attributes are inherited by the child classes. A method in the abstract class also denotes that actors cannot simply walk into any type of tree (the player will have to jump which is detailed in REQ2 below).

Each child class has its own unique functionality. Each of the child classes override the tick method that is implemented in the Ground class in the engine. This method is run with each turn (i.e: each time the player inputs a move) so therefore all of the trees' main functionality can be programmed into the tick method. With every tick, the turnCount attribute, which is a part of every tree, is increased by 1. This way, the turn amount for each tree can be tracked.

The Sprout's tick method first checks if the turnCount is equal to 10, if it is, the position where the Sprout is is replaced with a Sapling. If the turnCount is not 10, it has a 10% chance to spawn a Goomba (if there isn't an actor on the Sprout) on the Sprout's location.

In a similar fashion, the Sapling's tick method also checks if turnCount is equal to 10 or not, and if it is, it is replaced with a Mature tree. If a mature Tree is not spawned, the Sapling has a 10% chance to spawn a Coin item with 20 value on the Sapling's location.

The final stage of a tree is Mature. In the Mature tree's tick method, it first has a 20% chance to 'die' (i.e: be replaced with the Dirt ground type). If the Mature has not died, then it next checks if the turnCount is a multiple of 5, it does this because every 5 turns the Mature spawns a Sprout in a random surrounding area of the Mature. This is done by getting the exits of the Mature and then adding any exits that have the status of FERTILE_GROUND to an empty ArrayList of class Exit and then randomly choosing an exit from the ArrayList and spawning a Sprout on that exit. The Mature tree finally has a 15% chance to spawn a Koopa on its location.

Changes made during implementation

In the original design rationale the function of randomly spawning Sprouts at the start of the game was not addressed. This function is now handled by the SproutSpawner class. This class has a singular static method that has 3 local variables; the amount of Sprouts to be added to the map, and the maximum height and width of the game map. Sprouts are added all in one while loop, where it keeps looping if the sproutAmount is still greater than 0. The

loop chooses a random x and random y value and checks to see if the ground at that coordinate has the capability FERTILE_GROUND, and if so, spawn a sprout at that location and then -1 to the sproutAmount. If the ground is not fertile then the loop repeats again and a new random x, y coordinate is generated and checked.

Why it works that way:

Single responsibility principle: Each of the tree's child classes calculate their own functionality with each tick. For example, the mature tree's method to spawn random sprouts in the surrounding area is independent of the other tree child classes. This principle also applies to the SproutSpawner class whose only function is to randomly spawn sprouts at the game's initiation.
Open-closed principle: Additional tree types can be simply added without having to change the abstract Tree class. The attributes and methods are already provided in the tree abstract class. Any different implementation can be implemented by overriding a method in the child class, so no modification is necessary.
Dependency inversion principle: This principle is achieved through abstraction of the tree class. Also, the abstract tree class extends the HigherGround abstract class instead of the child classes themselves extending HigherGround.

REQ2

How it works:

Ground types that are able to be jumped to extend the HigherGround abstract class. The HigherGround class itself extends the Ground class in the engine, this is to enable the basic Ground class functionalities for the HigherGround child classes. The HigherGround abstract class stores attributes that contain the success rate for a jump, the fall damage for a failed jump, and the name of the high ground. The respective accessor functions for these attributes are also present in the abstract class. The constructor for the HigherGround class includes parameters for the success rate, fall damage and high ground name so each HigherGround child class may have their own specific values for these attributes.

The JumpAction class handles the actual movement of the player from one location to the high ground location or the fall damage if the jump is unsuccessful. This class extends the Action abstract class in the game's engine and has attributes that store the location of the high ground, the direction, and an attribute of the HigherGround class itself. The JumpAction is added to the list of allowable actions for a ground in a method present in the HigherGround abstract class. In the JumpAction's overridden execute method, the success rate of jumping to the higher ground is retrieved using the accessor and if successful, a new MoveActorAction object is created to move the player to the high ground. If the jump fails, the player loses HP based on the fallDamage attribute present in each HigherGround child class.

Changes made during implementation:

The type of ground does not need to be determined anymore, instead there is an abstract HigherGround class that is the only class that has the JumpAction in its allowable actions. This removes the need of using functions such as *instanceof* to determine if the ground is of a specific class.
It was assumed that the player would have a dependency with the JumpAction in the original design. This however was wrong and instead the ground (HigherGround in this case) would have the ActionList that contains the JumpAction.

Why it works that way:

Single responsibility principle: Only the JumpAction class deals with the calculation of the success/failure of a jump. The HigherGrounds do not have any code that deals with this calculation. The HigherGround abstract simply stores attributes for the success rate and fall damage which will be used by the JumpAction class.
Open-closed principle: If, for example, we wanted to add a new ground type that the player can jump to we simply just make that ground type extend the HigherGround class and add the attributes to the new class and the functionality for the jump will be completed. In this case, only the new ground type extends the functionality without modifying any existing code.
Liskov Substitution principle: Instead of using functions such as *instanceof* to check if the ground is of a certain class and then performing the jump (which was implied in the original design), the ground types that the player can jump to extend a HigherGround abstract class which itself contains JumpAction. This also reduces repetition and increases efficiency.
Dependency Inversion Principle: To reduce unneeded dependencies with the JumpAction class, the HigherGround abstract class was created, therefore, only the HigherGround class needs to have a dependency with the JumpAction class.

REQ3

How it works:

Both Koopa and Goombas extend the abstract Enemy class and implement the Resettable interface.

Goomba:

- *It starts with 20 HP*
- *It attacks with a kick that deals 10 damage with 50% hit rate.*

Starting HP is implemented when the object is initialised.

Attack details are specified in the overridden getIntrinsicWeapon() method.

- *In every turn, it has a 10% chance to be removed from the map (suicide). The main purpose is to clean up the map.*

To do this I used a SuicideBehaviour class and a SuicideAction class. There is a 10% chance that the getAction() method overridden in SuicideBehaviour will return a SuicideAction, otherwise the behaviour will return null. This will always be checked first, before any other behaviours are checked. The SuicideAction calls removeActor() on the Gooma, and menuDescription() returns a String saying that the Goomba has tragically passed away.

Koopa

- *It starts with 100 HP*
- *It attacks with a punch that deals 30 damage with a 50% hit rate.*

Starting HP is implemented when the object is initialised.

Attack details are specified in the overridden getIntrinsicWeapon() method.

- *When defeated, it will not be removed from the map. Instead, it will go to a dormant state (D) and stay on the ground (cannot attack nor move).*

Each turn when the Koopa checks for allowable actions I modified it so some are excluded depending on whether or not the Koopa is dormant.

The AttackAction checks whether the Actor being attacked has the Status CAN_BE_DORMANT. If it does, and it falls unconscious, it gains the Status IS_DORMANT instead, and a message stating what has happened is returned.

When a Koopa is reduced to 0 HP it gains the Status IS_DORMANT. The execute() method in the AttackAction class checks for this Status in the capabilityList of the Actor being attacked. It also checks whether the weapon being used to attack is a Wrench. If the Actor being attacked is dormant but the weapon is not a Wrench, the attack does no damage and a message stating that the attack was in vain is returned. Otherwise, the attack proceeds as normal.

- *Mario needs a **Wrench** (80% hit rate and 50 damage), the only weapon to destroy Koopa's shell.*

Wrench extends WeaponItem. Its constructor passes the wrench details to the super constructor. See above ^ for details on destroying Koopa’s shell with the Wrench.

- *Destroying its shell will drop a Super Mushroom.*

To do this I add a SuperMushroom to the inventory of each Koopa when they are initialised. Then by default, when AttackAction is used to kill them it sticks all of their inventory on the ground.

Changes made during implementation:

- Statuses are used to indicate conditions and abilities
- SuicideAction and SuicideBehaviour implemented
- FollowBehaviour has been considered and implemented
- Koopa and Goomba implement the new Enemy interface

Why it works that way:

Single Responsibility Principle: SuicideAction and SuicideBehaviour are their own classes. They each take care of their own responsibilities, which are not managed by the Goomba class itself.

Open-Closed Principle: In the future, the Enemy interface can be inherited by any new enemies that are added to the game. This allows the game to be extended in the future without modifying the existing classes.

DRY: Because both types of enemy have similarities such as the behaviourList, the Enemy abstract class was implemented to avoid repetition by storing commonalities in the abstract class rather than individually in the subclasses

Using Statuses to represent conditions and abilities allowed us to avoid using instanceof.

REQ4

How it works:

The PowerStar and SuperMushroom are Items, thus they extend the abstract Item class. Generally, there is a Consume action for them, which when done adds various Statuses to the Actor who consumed them. Then other classes check for these Status and act accordingly.

Power Star:

- *Higher Grounds. The actor does not need to jump to higher level ground (can walk normally). If the actor steps on high ground, it will automatically destroy (convert) ground to Dirt.*
- *Convert to coins. For every destroyed ground, it drops a Coin ($5).*

To implement this I altered the HigherGround abstract class so it checks for if an Actor is under the effect of the PowerStar and then instead of jumping creates a new PowerStarMoveActorAction. This class extends MoveActorAction. It uses functionality from the parent class to move the Actor, and turns the HigherGround they move onto to Dirt and spawns a $5 Coin.

- *Immunity. All enemy attacks become useless (0 damage).*
- *Attacking enemies. When active, a successful attack will instantly kill enemies.*

To do this I modified AttackAction so when doing the damage calculation it checks if the Actor has the HAS_EATEN_POWER_STAR Status. To instantly kill the enemies I set the damage to a very high amount (over 9000). To receive no damage if the Status is found the damage is set to 0.

Super Mushroom:

- *Increase max HP by 50*

When the ConsumeSuperMushroomAction is called the increaseMaxHp() method is called on the Actor who is consuming the mushroom, and 50 is added to the Actor's max HP.

- *the display character evolves to the uppercase letter (e.g., from m to M).*

Every turn, an Actors getDisplayChar method is called. If the Player has the Status, TALL, toUpperCase() is called on the original displayChar when it is retrieved; otherwise, the original displayChar is returned.

- *it can jump freely with a 100% success rate and no fall damage.*

JumpAction checks if the jumping Actor is TALL. If it is, the jump automatically succeeds and the Actor is moved to the space they were attempting to jump to.

Changes made during implementation:

- Consumable interface not implemented
- Instead of implementing a ConsumeAction, implemented a PowerStarConsumeAction and a SuperMushroomConsumeAction. These classes have an association with their respective Items

Why it works that way:

Single Responsibility Principle: The different Items have very different effects when consumed. Implementing different classes to handle the consumption of different items means that each class only has one responsibility. This avoids having unnecessary if-else statements in a single ConsumeItemAction class.
Another way this could be implemented is by using a Consumable interface that Consumable Items implement, which has a method that allows each item to determine what happens when they are consumed. That method could be called on the associated item in the ConsumeItemAction class when it is executed.
We did not choose that implementation method because we felt this way more successfully followed the Open-Closed principle. If Items are not initially introduced as Consumable, then to make them Consumable in the future requires modification of the existing classes. Our implementation allows for extension without any modification - all that needs to be done is for a Consume_____ItemAction class to be introduced.

REQ5

How it works:

The Wallet is a class that manages the amount of money the Player has. The Wallet is a singleton, as there never should be more than one. The Wallet value can be increased or decreased using mutator methods, and its current value can be retrieved using a getter.

Toad extends Action and implements CanSpeak. Toad overrides the allowableActions() method to allow the Player to take the SpeakAction, or three different types of BuyAction - each allowing them to buy one of the Items Toad can sell.

BuyAction extends Action. It has an attribute that stores the Item that can be bought using that BuyAction, and the price that the Item can be bought for. When the BuyAction is taken, the execute() method checks the amount of money stored in the Wallet. If that amount is smaller than the price, the String that is returned tells the Player “You don’t have enough coins!”, and they do not receive the Item. If they do have sufficient funds, the price is deducted from the amount in the Wallet, the Item is added to their inventory, and the returned String says that they have bought the Item.

Changes made during implementation:

- BuyAction doesn’t store any Actors, but it does store the Item that it is selling
- PickUpCoinAction extends PickUpItemAction
- Sellable interface not implemented. Cost stored in BuyAction
- Wallet implemented as a class instead of an attribute of Player

Why it works that way:

The execute() method that is overridden in BuyAction means that it is not necessary to store the Actor that is taking the Action.

PickUpCoinAction extends PickUpItemAction because it is a variation on the Action of picking up an Item. Implementing it this way allows the class to use parts of the PickUpItemAction such as removing the Item from the Ground. If it was not implemented this way, it would have violated the DRY principle because it would have been repeating aspects of the code already present in the engine.

The Sellable interface was not implemented because the price for an Item is stored in the BuyAction. This allows for expansion of the game if another seller that sells Items for different prices is added. So the prices of Items are set by the Actor selling them, not the Item itself.
This considers the Open-Closed principle, because even if Items are added that initially aren't able to be sold, these Item subclasses will not need to be modified to make them sellable.

Wallet has been implemented as a Class instead of an attribute stored by Player. This has been done so that it does not violate the Single Responsibility principle be having the Player need to manage its own Wallet. Instead, as a singleton, the instance of Wallet is simply called and each time it needs to be updated or read from.

REQ6

How it works:

CanSpeak is an interface to be implemented by Actors that have the ability to speak. It has one method, speak(). This method takes an Actor as an argument so that speech can be tailored to that actor.

Toad defines the functionality of the speak() function such that it checks for a Wrench in the Player's inventory, and checks for HAS_CONSUMED_POWERSTAR in the Player's list of capabilities. It then uses this information to randomly choose an appropriate line of Toad's dialogue to return. The lines of dialogue Toad can say are stored in an ArrayList<String> and initialised when the Toad is constructed.

Changes made during implementation:

- Canspeak doesn't involve an actor that is being spoken to as an attribute

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. The CanSpeak interface allows the program to be easily extended in the future, and increases code readability by clearly showing what Actors can and cannot speak.

REQ7

How it works:

ResetAction extends Action. ResetAction overrides the execute() method so that when the Action is taken the run() method of the ResetManager is called. This method runs through the resettable list stored in ResetManager and calls the resetInstance() method on each resettable instance stored in the list.
Classes that are resettable implement the Resettable interface. In each of these classes, the resetInstance() method is implemented so that it performs the necessary actions to reset each instance.

- For enemies (Koopa and Goomba), this method calls removeActor()
- For Player, this method retrieves the capabilitySet of the Player, and removes the HAS_EATEN_POWER_STAR and TALL Statuses.
- For Coins, this method calls removeItem()
- For Trees, this method calls setGround(Dirt) 50% of the time

Changes made during implementation:

- None

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

Please note we understand that we have two cases of non-object oriented programming. The tracking of the time a PowerStar can affect the Player is currently managed in the Player class, which violates the Single Responsibility Principle. I believe that to implement this correctly we would have needed to use the getNextAction() method. We also have a two-way association between PowerStar and ConsumePowerStarAction where they are each associated with each other.