Design Rationale (Assignment 3)

REQ1. Lava Zone

Lava

Lava is instantiated on the map using FancyGroundFactory whereby we hardcoded the lava on the map just as we did for Saplings and Walls. Since Lava Zone is generally a new map instead of a whole new world, we just need to create a new FancyGroundFactory and add it to our gameMap. This allows us to utilise the addGameMap function in World to add additional maps for the game.

We adhere to the **Open Closed Principle** whereby we just extend the Ground class without modifying the class itself. We can use the methods already created in the Ground class and extend it by adding what is required.

WrapPipe

As WrapPipe requires a dependency to the normal gameMap and lavaMap, we would instantiate it as normal just like how we instantiated our actors. This allows us to pass in the parameters of the locations of different gameMaps to travel to. To allow our player to teleport back to the last pipe from Lava zone, we added new attributes to the Player class such as a boolean atLavaZone which indicates whether the player is at lavaZone and Location lastTeleport which defines the last pipe the player is at before teleporting to the lavaZone. When a player wants to teleport back to the normal gameMap, our teleportAction would access the player's lastTeleport attribute and move our player back to the lastTeleport location.

REQ2. More Allies and Enemies

Princess peach

Implementing this is easy because this actor does not have any behaviour. Just extend this class from the actor and change the display char. Only one allowable action from the player, that is rescue action if the player has a key.

Bowser

Bowser is very similar to koopa or goomba. Except it has a higher base attack and it does not wander around. It also has another status called FIRE_ATTACK. This is to implement the fire attack on Bowser. In the attack action, actor status will be checked. If the actor has fire attack status, it will leave fire on the ground for 3 turns. By doing so we adhere to **single responsibility principle**. Reset the position after resetting the game is done in tick method by checking whether a game is reset every turns.

Piranha Plant

This is rather easy as well because it only has attack behaviour. However, the challenge lies in spawning the plant at the location of warp pipe. This can be done by in the tick method in warp pipe class. When start will spawn the plant. Also spawn the plant every time the game in reset by checking it every turn in tick method of warp pipe.

Flying Koopa

This class is just an extension of Koopa. Based on **Liskov Substitution Principle**, this flying koopa class can do everything Koopa can do. Only differences is the stats.

REQ3. Magical Fountain

Water

Water is actually using interface Consumable because water behave is similar to magical items. The difference is the consume method that executed when consume action is done. Power water is to buff actor while heal water heal actor in their respective consume method.

Bottle

Bottle are using stack adt from java util library. The stack will contain all the waters. When water is consumed, pop from bottle. If water is added, push to bottle. Also get bottle action is added to get the bottle from toad. Just add the bottle to actor inventory when this action is chosen. Toad can only give 1 bottle as of now.

Fountain

A fountain class is added which act as a parent class from power and heal fountain. Most of the logic is in this parent class, only the difference is coded in the child class. Also, fountain is implementing two interfaces which are drinkable and refillable. This applies the **Interface Segregation Principle**. Although for now it should not be a problem to combine these two interface, but if we want to add something that is only drinkable but not refillable, we will not be able to.

REQ4. Flowers

FireFlower

As FireFlower is part of a power up item, we would just extend the Item class and implement Consumable class, adhering to the **Open Closed Principle.** This allows us to be consistent in implementing our game Items as well as reusing the methods in the parent class.

Since Sapling and Sprout are able to spawn a FireFlower, we added a SPAWN_FIREFLOWER capability so that only game objects with that capability are able to spawn a FireFlower. Since we assume that Sprout and Sapling has a 50% chance of spawning a FireFlower throughout its growth span, we made sure that they do not spawn another FireFlower if there is already an item at its location. This prevents the map from being overly populated with FireFlower.

REQ5. Speaking

Bowser, PrincessPeach and Piranha Plant are extended from Actor abstract class.
Koopa, FlyingKoopa and Goomba are extended from WanderingEnemy abstract class which is extended from Actor abstract class.

Clarification of Association Used

- **Bowser, PrincessPeach, PiranhaPlant, Koopa, Goomba, FlyingKoopa — SpeakAction**

There is an association between all 6 characters mentioned above and the SpeakAction class as an instance of SpeakAction will be implemented in each Bowser, PrincessPeach class and PiranhaPlant class. Within the SpeakAction class contains the 6 separate

Arraylists to save all specific sentences bound to each specific character (e.g Bowser, PrincessPeach and PiranhaPlant).
The sentences for each character are stored in a method which can be called outside of the SpeakAction class, this prevents the potential risk of having a tight coupling problem.

Assignment 2

Design Rationale

REQ1. Let it grow!

Updated:

The relationship between sapling, mature and sprout has been changed to a dependency as each class would just need to use another as input parameter. Wallet class has also been added to store the coins, the relationship is a dependency. The relationship between sapling and coin has been changed to dependency. Instead of creating a pickcoinaction class, we implemented it using pickup action as Coin is portable and if the item is an instance of Coin, we increase our wallet balance.

Tree abstract class has an addneighbour method to find available dirt. This is implemented in Tree instead of Mature class directly so that not only can Mature class can inherit the method, other tree classes can inherit it as well if needed to.

Tree is made an abstract class so it acts as a base class which can be extended by other subclasses such as sapling, sprout and mature but the Tree class itself cannot be instantiated.

Spawn Enemy.

Since Sprout and Mature will spawn a Goomba and Koopa respectively, the former classes will have an association to the latter classes with a one-to-one relationship. Each of the Sprout and Mature will contain a Goomba and Koopa attribute as the spawn method in Sprout and Mature can potentially access the attribute, making it a strong relationship.

Growing of trees.

Since the trees (Sapling, Sprout and Mature) have a life cycle, we design it in such a way that they will have an association to the tree of their next cycle. For example, Sprout will have an association with Sapling which will have an association of Mature and Mature will have an association back to Sprout. It is done this way so that each tree will have an attribute of the next tree they will grow into. Dependency is not used as we need to store an attribute of the next tree cycle since a tree would need to know about the next tree it will grow into and access its method to instantiate that tree.

Coin.

As Sapling has a chance to drop a coin, we would just need to create a Coin object. It has an association as Sapling would need to specifically create a $20 coin, thus requiring access to the Coin object's attribute, therefore making it a strong association.

REQ2. Jump Up, Super Star

Updated:

JumpAction class extends Action class in engine directly to implement its method. The player and SuperMushroom classes have been removed as they are redundant. JumpAction would only need a strong association to all jumpable interfaces for jumping. Sapling, Sprout and Mature now extends the Tree abstract class in the game folder.

JumpAction.
JumpAction has an association with the SuperMushroom class. It will get an instance of the SuperMushroom object to determine the success rate of each jump. It will also access the SuperMushroom attribute to get its capabilities which give the player 100% success rate in jumping as shown in REQ4. Hence it is a strong relationship because the JumpAction method can access this attribute. Since magical items are not the focus of this REQ, the implementation of magical items will be shown in REQ4.

We created a JumpAction in the 'game' package that extends MoveActorAction in the engine package which will inherit the methods to move the actor on top of the objects in the Ground class (eg: Sapling, Sprout and Wall) when it performs a jump action. Using the **Dependency Inversion Principle**, we implemented a Jumpable interface which is extended by the Ground, Sapling, Sprout and Wall so that instead of having an arraylist of jumpable high grounds in our JumpAction, we only need an arraylist of type Jumpable in our JumpAction class. So, if we ever need to create a new high ground class that is jumpable, the new class would just need to implement the Jumpable interface. Thus we do not need to modify the source code of the JumpAction class.

<u>REQ3. Enemies</u>

Actors attacks Actors
Since we know that player and enemy extends Actor abstract class and implement the actackedBy method, we would just need an association from AttackAction to the Actor abstract class so that any class object extended from the Actor abstract class can be attacked by each other. This is especially useful such that when a new enemy is added, it just needs to extend the Actor abstract class to apply the entire attack process, hence applying the **open-closed principle** where the enemy class can be extended without modifying the class itself.

Enemy Behaviours
Instead of directly extending the enemy base class to include behaviours, we added a more meaningful interface called Behavior which is extended by AttackBehavior, WanderBehavior and a FollowBehavior. This would make our design more robust whenever we need to use Behavior at other classes such as for a new non-playable character class that is not part of the enemy. This allows any new classes to implement the behaviours as intended without modifying the base class itself. This is a part of the Liskov Substitution Principle when we ensure the robustness of polymorphism.r

Destroying Koopa's shell
Since Mario requires a Wrench to destroy Koopa's shell, we have a dependency from ActackAction to a Weapon class in our engine package, this is to access the capabilities(attributes) of the specific weapon used. If the weapon is a wrench, it will have the ability to destroy Koopa's shell.

List of Actions
Each actor would have a list of executable actions. Therefore, we made an abstract Action class to apply the open-closed principle. Then, the ActionList class will contain attributes of our Action class. This is done so that a new class would just need to extend the Action class

to obtain a list of executable actions instead of modifying the code in the class to include dependencies of a list of actions, thus making the Action class extendable without any modification.

REQ4. Magical Items

- Each magical item will have one and only one capabilities set.
- The player class will hold the consumed magical item until certain conditions are met. For example, 10 rounds limit has reached, or receive damage. Then the consumed item will be removed.
- The consumed magical item will be active hence we can use the capabilities set from the magical item after consumed.
- One way we can take care of disposal of consumed items is using tickers to count the round for power stars. For super mushrooms, we can compare this round hp vs last round hp for player to know whether any damaged is received
- Each capability will be in at least 1 capabilities set but capabilities set may or may not contain each capability.

Clarification of Association and Dependency Used

- **DormantKoopa — DestroyShellAction — Super Mushroom**

This uses dependency because Dormant Koopa instance will not contain DestroyShellAction class as an instance variable, It will simply use it in a method. DestroyShellAction instances contain item instances, hence association is used.

- **Player — ConsumeAction — magical items**

Player will call consumeAction but there is no instance of it in player instance hence it is just a dependency. ConsumeAction to magical items is an association because consumeAction will have the item in its instances.

- **Resettable — magical items**

We want to make magical items buff resettable hence they need to implement resettable.

- **Capabilities set — item**

Each item instance may or may not contain a capabilities set. For example, it makes sense for magical items to contain capabilities set because they can do many things. But it does not make sense for items like coins to have a capabilities set because coins do not do much. Hence, association and multiplicity in the diagram is suitable.

REQ5. Trading

When player and toad are next to each other, trading can occur. Trading action doesn't really return an object, but instead it returns a permission for the player to buy an item. If insufficient wallet balance, player cannot buy that item, else balance is deducted from getting the item. TradeAction is responsible for deducting the balance while players are responsible for getting the item.

Clarification of Association and Dependency Used

- **Toad — tradeAction — player**

1 condition is needed to trigger trade action. 'Toad' needs to be beside the player. But trade action class does not contain toad and player. Hence, In our diagram, we include toad and player as a dependency to trade action. Trade action will also need to decide whether the player has enough balance for an item, hence a dependency of player to trade action is needed.

- **Sellable interface — sellable items**

Sellable items will need to implement the sellable interface to determine which items are sellable and which are not.

- **tradeAction — sellable**

This is a dependency relationship because trade action will not contain these items objects inside the class, it will only use magical items objects as method parameters to query for price.

- **tradeAction — wallet**

This is a dependency as well because tradeAction will deduct balance from wallet but wallet is not an instance variable of trade action class.

- **Coin — Wallet**

Coins will increase wallet balance if player pick up coins. Dependency because wallet will not store coins.

- **Player — Item**

Player instance will hold those items hence the association.

REQ6. Monologue

Toad will speak a total of four different sentences which is dependent on what item the player holds (if any). The four different sentences are stored in the monologue class.

Clarification of Association and Dependency Used

- **Toad — MonologueAction — PowerStar**

There is an association between Toad and MonologueAction class as an instance of Toad will be implemented in MonologueAction. If the player has a power star active, the Toad must not print out the second sentence “You better get back to finding the Power Stars.” . This would require a dependency in our diagram between PowerStar class and MonologueAction.

- **Toad — MonologueAction — Wrench**

There is an association between Toad and MonologueAction class as an instance of Toad will be implemented in MonologueAction. If the player has a Wrench and interacts with the Toad, the Toad must not print out the first sentence “You might need a wrench to smash Koopa's hard shells.” . This would require a dependency in our diagram between Wrench class and MonologueAction.

- **MonologueAction — Monologue**

This is an association between MonologueAction and Monologue as MonologueAction class will create instances of Monologue class.

REQ7. Reset Game

Updated:

There is now a direct relationship between Tree class and ResetManager class instead of to all of its subclasses, this is to practice open-closed principle where we just need to reset the Tree class instead of individually resetting the sub-classes of Tree class. Some coins may still be seen on the map even after reset, this is because there still exist some trees that are not turned into dirt, thus still being able to spawn new coins.

Based on our UML class diagram, in order to reset the game, we will implement four abstract classes (Action, Actor, Ground, and Item). Each abstract class will be extended based on the requirements as will be explained below.

Clarification of Association and Dependency Used

- **Player - ResetAction - ResetManager - Resettable <Interface>**

When a player clicks on 'r' and activates resets, it will activate ResetAction via an Association which then activates the ResetManager via another Association. ResetManager is implemented by the Resettable Interface.

- **Trees have a 50% chance to be converted back to Dirt**

In order to reset the trees (Sprout, Mature, and Sapling) into Dirt, ResetManager which connects to the tree classes via an association will destroy them. Each of the tree classes (Sprout, Mature, and Sapling) are connected to Dirt class via an association as they may implement an instance of Dirt class to be converted to dirt.

- **All enemies are killed.**

Koopa and Goomba classes will each have an association with the ResetManager class as they will implement a method which will kill the enemies.

- **Reset player status**

This is an association between Player class and ResetManager. ResetManager will reset the status of players.

- **Heal player to maximum**

This is an association between Player and ResetAction class, ResetAction class will have an attribute of the ResetManager class (an association relationship which is shown in UML class diagram).

- **Remove all coins on the ground (Super Mushrooms and Power Stars may stay).**

This is an association between the Coin class and the ResetManager class, this is to reset the coin count. The ResetManager class has an attribute of type Coin in order to reset the coin count.