Updated Classes and Design Rationale

Classes

World

- this controls and manages how the game is run (IMPORTANT).
- runs the main loop that controls the flow of the game.
- At each iteration of the world, draws `GameMap()`.
  - controls state of game map
  - tells player state.
- Should manage multiple `GameMaps()`'s at each iteration.
  - time should flow for all maps.

IMPLEMENTATION DETAILS (GIVEN):

- `run()`: contains the main game loop, defined in a while loop. the condition for this while loop is determined by the return value of the `stillRunning()`.
  - Game will run as long as player is alive.
- Rendering: Renders the map where the player is currently located.
  - Finds the GameMap the player is currently located.
  - Calls map method of `Location` in order to obtain `GameMap` that the location is currently in.
  - `draw()`: next the draw method is called from the GameMap class, and draws each (x, y) position in the game map.
  - Processing: Then it processes each actor's turn and checks if `stillRunning()` is true.
  - `tick()`: then edits the tick.

GameMap

IMPLEMENTATION DETAILS (GIVEN)

- In the `World()` there can be more than one GameMap.
- Purpose is to help the `World()` in controlling and management of game.
- manages players, objects and states.
- Receives orders from `World()`.
- Time should be same and flow for all different maps.

Location

IMPLEMENTATION DETAILS (GIVEN)

- Purpose is to provide the ability to get the exact locations of Actors/entities.
- Simple use of x and y coordinates
- Each position in the `GameMap()` will be defined by the `Locations()`
  - GameMap contains Locations.
- Responsible for telling `World()` and other game entities what is available at position (x,y).
- keeps track of:

    - exits
    - character representations
    - terrain type
    - entity representations
    - tick timers for each item.
- Contains:
    - items
    - ground
    - exits
    - should also keep track of Actor's locations?
- Can move actors, checks if actors can enter location
- can only contain ONE actor at a specific Location.
- location method draw already has resolved display hierarchy.

Exit

IMPLEMENTATION DETAILS (GIVEN)

- Purpose is to describe the surroundings of a given location and represent a route from one `Location()` to another.
- Provides you with the name of the exit, its location and the movement hotKey.
- gives you a list of possible locations that you can move to.

Ground

IMPLEMENTATION DETAILS (GIVEN)

- Purpose is to allow choices for terrain types.
    - wall
    - safe area (floor)
    - sprouts, sapling or mature.
- These terrain types allows different entities to tell whether they can pass through the location or not.

Notable Features:

- want to know whether a certain entity can enter a specific ground.
- Some different ground terrain types may need different abilities,
    - safe hiding area
    - block projectiles etc.

Sprout

Purpose: To spawn Goomba Objects and then Form into a sapling after it exists for 10 turns

- Single Responsibility: To act as an object that spawns Actors with the added responsibility of turning itself into another object
- Open-Closed: Sprout should only be open to extension to the Ground Class
- Liskov's Substitution: Replacing Ground with Sprout should not result in any unidentified behaviour

- Interface Segregation: The sprout object will implement the Dirtable interface which grants it the ability to implement a method to turn itself into dirt if certain conditions are met

ASSIGNMENT 2 [AAUT0001]: Nothing has changed here, implmeentation details are below.
Implementation Details (Sapling, Sprout, Mature)

Sapling

Purpose: To spawn Coin Items and then form into a Mature after it exists for 10 turns

- Single Responsibility: To act as an object that spawns Items with the added responsibility of turning itself into another object
- Open-Closed: Sapling should only be open to extension to the Ground Class
- Liskov's Substitution: Replacing Ground with sapling should not result in any unidentified behaviour
- Interface Segregation: The sapling object will implement the Dirtable interface which grants it the ability to implement a method to turn itself into dirt if certain conditions are met

ASSIGNMENT 2 [AAUT0001]: Nothing has changed here, implmeentation details are below.
Implementation Details (Sapling, Sprout, Mature)

Mature

Purpose: To spawn Koopas at its location, Sprout objects in its surrounding location and to form into Dirt on a 20% chance

- Open-Closed: Mature should only be open to extension to the Ground Class
- Liskov's Substitution: Replacing Ground with mature should not result in any unidentified behaviour
- Interface Segregation: The mature object will implement the Dirtable interface which grants it the ability to implement a method to turn itself into dirt if certain conditions are met

ASSIGNMENT 2 [AAUT0001]: Nothing has changed here, implmeentation details are below.
Implementation Details (Sapling, Sprout, Mature)

Implementation Details (Sapling, Sprout, Mature):

In order to spawn new Actors, and also spawn a new ground object on the current location of ground. These objects will access the location class through the Location parameter in the tick function. Giving it access to methods implemented within the location class. Allowing it to set new grounds, get Actors and add new actors from the current location it is at. It would also therefore allow the object to access its surroundings through the exit functions, granting it access to perform methods on its surroundings as-well through the map object stored in the Location class, and the location method that retrieves the destinations of the current locations exits

Assignment 2: No Changes to the Initial design

Dirtable

I chose to use an interface to represent the objects that need a method to turn themselves into dirt. This is because throughout multiple stages of the game (ie. During the reset function and during when mario is under the consumption of a Power Star, the Wall, Sprout, Sapling and Mature objects can be turned into dirt. So an interface is therefore implemented to avoid the repetition of coding, and takes a location parameter which will set the ground at that current location to Dirt.

Assignment 2: The Dirtable interface was scrapped, because the removal of ground during the reset and during the Powerstar are completely different. This is because the deletion of dirt when a Player consumes a powerStar will take effect during its movement action, whilst its reset takes place when the Reset Action is called, and also possesses a boolean which wouldnt be needed for the latter deletion.

Item

IMPLEMENTATION DETAILS (GIVEN)

- Purpose, to act as an object that the Player can use.
- needs to be Portable or nonPortable
    - can be picked up or dropped.
    - stays in players inventory. (not droppable)
- Item may also have a duration – uses `tick()` and overrides it.

Consumables

Purpose: Allows us to consume the item from the inventory in order to unlock buffs for a specified period of time

- Open-Closed: Both the Super Mushroom and Power Star should only be open to extension to the ConsumableItem Class
- Liskov's Substitution: Replacing both PowerStar and SuperMushroom with ConsumableItem should not result in any unidentified behaviour

Implementation Details Both classes should inherit a new abstract class within the Games package called ConsumableItem, this gives the class details in regards to whether or not it is able to be consumed by the player. This helps reduce code repetition as it can be defined in the abstract class and then inherited by both classes.

A new class within the Actions package named ConsumableAction is used that gives the player the option to consume items if it recognises that it has attributes that belong to the ConsumableAction parent class. Otherwise it will not recognise the item as an item that can be consumed (ie. Coin, Wallet, Wrench) In order to grant the player benefits after the consumption of an item. New status can be defined (ie. Possibly referred to as powerStar and superMushroom). That is then added to the capabilities of the person who consumes the item, which would then result in buffs given to the actor

---

Wrench

Purpose: to Act as a item and a weapon used to break the shell of a dormant Koopa Open-Closed: the Wrench class should only be open to an extension of the weaponItem abstract class Liskov's Substitution: replacing Wrench with weaponItem should not

result in any unidentified behaviour Implementation details: Wrench is going to be inherit the WeaponItem class, so it can be stored within the Inventory whilst being classified as weapon. This will allow the getWeapon() function to recognise the wrench as a weapon to be used to attack enemies. In order to kill the koopa shells, we would need to ensure that the weapon used is an instance of a Wrench

---

Wallet

Purpose: to Act as an item that is stored in the actors inventory and is used to store the balance of coins when they are picked up by an actor that has a wallet in their inventory Open-Closed: the Wallet class should only be open to an extension of the Item Abstract class Liskov's Substitution: replacing Wallet with Item should not result in any unidentified behaviour

Implementation Details: The main purpose of the wallet class is to serve as an item with a balance value attached to it. It will be added to the players inventory upon initialization of the game, as the player is the only actor that has use of the coins, but to also make sure that other actors can also pick up the coins. This is because when the player picks up the coins, it will update the balance of the Wallet value instead of picking up the coin and storing it in its inventory. This is most likely going to be achieved by viewing the Actors inventory, and checking if any items they have are an instance of Wallet, if so, that object associated values will be updated, else, the actual coin object will be stored. The wallet’s value will then be used for trading

Assignment 2 Changes:

The purpose of the wallet remained the exact same, however, instead of “checking if any items they have are an instance of Wallet” The methods will instead check to see if the item has the Status “Wallet”. This is to avoid any code sniffing

---

Coin

Purpose: to Act as an item that is stored in the actors inventory unless they have a Wallet item, which it is then added to the wallets balance and not stored in the inventory Open-Closed: the Coin class should only be open to an extension of the Item Abstract class Liskov's Substitution: replacing Coin with Item should not result in any unidentified behaviour Implementation: The coin object will be assigned a particular value that represents the coins monetary value. (ie. 5,10,20,etc) This will be done within the Constructor, after it calls its super back to the Item class. The values of these coins will be dependant on the situation in which the coin is found. For example, coins created through Saplings will be assigned a value of 20 whereas coins dropping from destroyed terrain will only be created with a value of 5. This is manually inputted as we control when the coins are dropped and the situation in which they are created Playing off the wallet class, the Coin acts as an object scattered throughout the map, upon pickup by any actor, the game would first have to check if the actor has a Wallet (which would only be the Player). This is to allow any actor to pick up the coin object without any issues. If the actor does have a wallet, the PickUp action will simply read the value of the coin, update the wallets balance and delete it from the map. However, if the actor does not have a Wallet, it will simply just add the coin item to its inventory without any further actions.

Assignment 2 Changes [JTUC0006]:

When the actor walks over a coin, the actor will be given the choice to pick up the coin as an individual item. Or, if the actor has a Wallet, the actor can directly store the coins value into the wallet. The Pickup Action is not changed, but a new Action is created that is used to store the value of the coin to the wallet and remove it from the map, or from the inventory if the actor stored the coin first

---

Actor

IMPLEMENTATION DETAILS (GIVEN)

- Purpose: Allows us to represent Player, Enemies and non-hostile entities (such as Toad) as actor classes.
- Implementation Details:
  - Hit Points: Already given in the code.
  - Knowing the Actor's hit points will allow us to know what the next actions to take are.
  - Actors Action List: Knowing what actions the Actor can take is vital to gameplay as well.
  - Location will care about actor heights and GameMap will have the logic for whether the Player needs to jump or can simply walk down.

Capability

- Purpose: Enabling the ability to inflict special statuses on Actors, or entities
- Implementation Details (GIVEN):
  - Can create an `Enum` to do this, which stores a variety of different statuses

NOTE:

- Actor: the `Player` can get an ATTACK_UP status once they drink a potion
- Item: a weapon might have a CURSED or a POISONED status attached to it.
- Ground: might have a SLIPPERY or WET status attached to it.
- Capabilities: `Status` could be divided into two types
  - Buff: any status that improves performance of actor.
  - Debuff: any status that worsens performance of actor.
  - Temporary: A status that can be removed after a certain number of turns
  - Permanent: A status that cannot be removed the entire game.
  - ==Do not use statuses to represent identities==.

**ASSIGNMENT 2 [AAUT0001]

Action

- Purpose: Represents the action that an actor can perform.
- Allows Player to perhaps attack the Enemy, consume items, and perform other actions.
- Implementation Details (GIVEN):
  - The action that the actor can perform will be given by the object.

- Another `Actor` as the object: If an Enemy stands next to the `Player` the following conversation can occur.
  - Engine to Enemy: What actions is the `Player` allowed to use on you?
  - Enemy to Engine: If the `Player` has the `HOSTILE_TO_ENEMY` status, then they can perform `AttackAction` on me.
  - Engine to Player: `Player`, you can do `AttackAction` on the `Enemy`.
  - `AttackAction` will be displayed to the user.

NOTE: Do not implement all the actions that the `Player` can implement inside the `playTurn()` - violates SRP.

Behaviour

- Purpose: To allow NPC such as Enemy to execute actions. Acts as a wrapper to an action that an NPC can perform, as cannot make decisions based on input.
- Implementation Details:
  - An NPC would have an attribute that stores a list of Behaviour's. each `Behaviour` from the list will return an action. Therefore, the list of `Behaviour`s will result in a list of actions that the NPC can choose from.
  - Behaviour: may return a null action
  - Behaviour: may have higher priority than others.
    - Attack behaviour should be at the top of priority
    - Follow behaviour should be next... etc.

Capable

NOTE: This interface is Capable as Capable sets Statuses and This provides a way for an Enemy to attack another actor.

- Single Responsibility: To provide a way for Actors to interact with each other, say you have an enemy, `Goomba` and it is near the `Player` then you want to allow `Goomba` to attack `Player`.
- Open-Closed Principle: Since the `Actor` class is open to inheritance, this allows enemies such as `Gooba` or `Koopa` to inherit its useful attributes and the `Actor` class stays closed to modifications.
- Liskov Substitution: Enemy classes `Goomba` and `Koopa` as well as the `Player` class can be substituted for the `Actor` class and will not have any undefined behaviour.
- Interface Segregation: This Interface has the single responsibility to allow Actors to attack each other.
- Dependency Inversion: since Liskov's and OCP are already adhered to, this one is also adhered to.

Goomba

- Purpose: Act as a hostile entity towards Player instance, attacks Player, follows player if near. Clears the map if unconscious.
- Single Responsibility: A hostile entity to the `Player`.

- Open-Closed: `Goomba` inherits from the open abstract `Actor` class, which is closed to modifications. `Goomba` itself is the child class and will be the enemy that is hostile towards the `Player`.
- Liskov's Substitution: Since `Goomba` follows the OCP, it also satisfies this Liskov Substitution principle. It can be replaced with the `Actor` class and no undefined behaviour should occur. To ensure this, `Goomba` implements the interfaces: WanderBehaviour, AttackBehaviour and FollowBehaviour to allow it to wander around the map, attack the Player if able to and also follow the Player if able to, respectively.
- Dependency Inversion: Follows from satisfying OCP and LSP. This `Goomba` class should not depend on the three `Behaviour` classes, instead it will depend on the Interface `Behaviour`. The `Goomba` class can hold a collection of `Behaviours` that the Object `Goomba` itself will use to decide its next move.
- Implementation Details:
- Since it is bad to have too many levels of inheritance, `Goomba` will only inherit from `Actor` class and then implement its own attack success chance (hit rate) and removes itself from the map if it has 0HP.
- Inheriting from the `Actor` class is beneficial as it allows the `ActorLocationsIterator` to move the Enemy around.
- if unconscious –> remove from map.
- Sprout can spawn –> Goomba so dependency
    - Cannot spawn enemy if actor standing on it.
- `Goomba` can attack and follow the `Player` so it may have a dependency on the `Player`.

Assignment 2 [JTUC0006] and [AAUT0001]:

Not many changes were made to the Goomba Class. Its movements, Attacks and checking whether or not it has died (Because the only way it will die is if it gets attacked) are done through Actions and Behaviours, so are not done by the Goomba Class itself, but instead is done through a separate entity that controls all attacks and movements made by all enemies There was no mention about how the Goomba commits suicide within the intial implementation. But was done during the playTurn method, so for each turn the Goomba makes, it has a chance of dying

---

Koopa

- Purpose: To act as a hostile towards the Player class, it attacks the player with more damage than the Goomba
- Single Responsibility: To act as another hostile enemy towards the `Player` with the added responsibility of dropping `SuperMushroom`s.
- Open-Closed: Goomba should only be open to extension
- Liskov's Substitution: Replacing Actor with Koopa should not result with undefined behaviour.
- Interface Segregation: Like `Goomba`, `Koopa` will implement the `Behaviour` interface to allow the class to be able to Follow, Attack and Wander around map.
- Implementation Details:
    - Hit Points: starts with 100HP

    - Attacks with punch that deals 30 dmg, 50% hit rate.
    - When defeated, Map display icon changes to `D`.
    - Mario needs `Wrench()` to destroy the shell.
    - When shell is destroyed, Super Mushroom is dropped:
        - map icon should display SuperMushroom icon
    - Unconscious: Cannot attack, follow or wander around.

Assignment 2 [JTUC0006] and [AAUT0001]:

Not much was changed compared to the initial implementation of Koopa during Assignment 1. The details state everything that the Koopa does and was how it was implemented

---

Jump Requirement

- Purpose: The `Actor`, `Player` will need to be able to move to higher ground level such as on top.
- Implementation: Using an interface `IHeights` which has an integer attribute for the height and a method to set the height. By having the different types of `Ground` (Sprout, Sapling, Wall and Mature) implement this interface we can give each object a height as well as having the `Player` implement this interface. Next using the `actorCanEnter(Actor actor)` and overriding it, we can place conditionals that check if the actor is at a lower level than the ground type of its destinations. If satisfied then it adds the jump capability to the CapabilitySet. This allows the actor to jump using the `JumpAction` class and if the jump option is selected, calculates the chance of success and executes the logic behind the jump.

Before checking if the actor is at a lower level, implementation will check if the actor has a SUPER_MUSHROOM_STATUS effect on it that allows it to have a 100% success rate for jumping. If the status is in effect then the Player can just move to the higher ground without any problems.

The `ConsumeAction` changes the chance of success for a jump to 100%. This action will have a dependency on the `IHeights` interface. As this interface

By using an interface that adds a heights capability it ensures that the children classes of Ground and Actor follow Liskov's Principle, as these two parent classes are open to extension but not modification. Otherwise, adding height attributes to each child class would violate this principle as well as make implementation and maintenance more difficult in the future.

As `Koopa` when unconscious becomes a shell, and when the shell is destroyed it becomes a `PowerMushroom` there will be a dependency between these two classes as well as a dependency between `Toad` and `PowerMushroom` as Toad sells the `Item` to the player.

ASSIGNMENT 2 [JTUC0006] and [AAUT0001]:

The interface “IHeights” was scrapped and was replaced by the Jumpable interface. This controls the success of attempting to jump to the particular ground, and the damage it takes when failed.

Each class will call add a new action to their allowableActions if there are no actors currently on that ground. This will be used to implement the JumpAction, where it will

check if the jump was successful, and if so, will move the player to the higher ground, and if not, will not move the player and will instead damage the player.

If a player consumes a Super Mushroom, none of the Success rates will be changed, instead, it will simply bypass the need to check the grounds probability, and go straight to moving the player to the target location

If a player has consumed a Powerstar, the jump action will be replaced by a new movement action which will simply set the target located to dirt. Spawn a coin and then move the actor to the target location

---

Toad

-Purpose: Toad is a friendly NPC actor that spawns in the centre of the map surrounded by brick walls. The Toad provides the player with the ability to trade the player's collected ‘coins’ in exchange for items. Additionally, the toad has a set of available dialogue options that is randomly presented to the player when they speak with the toad

-Implementation: The Toad will have in its available actions method a speak action and a trade action for all the marketable items. When the toad creates the items for all the trade actions it will set the portability for every item as false.

- Single Responsibility: to provide a trading entity
- Open-Closed: `Toad` inherits from `Actor` which is open to extension and closed to modification. So this satisfies the OPC.
- Loosely coupled `Wallet` to trading to the `Player` is not tightly coupled and allows other entities to be added to the game and allows them to trade as well.
- interface Segregation: Implemented the interface `Marketable` to give certain `Items` the ability to be tradeable. This is good as they do not rely on other interfaces they don't need such as `Behaviour` to be tradable.

Trade Action

-Purpose: TradeAction represents an action an actor can perform to obtain a specific item in exchange for their ‘coins’.

-Implementation: The action will store a specific ‘Marketable’ item for which it is selling. The action works by performing its validation of whether the actor has a wallet/sufficient balance after it has been executed as to present all the tradeable actions to the actor. The execute action works by checking to see if the player has a wallet, if so it gets its balance and returns "You don't have enough coins!" if there is insufficient. Otherwise the balance is decreased and the item is added to the actors inventory

- Single Responsibility: responsible for presenting an action that an `Actor` can perform to obtain an `Item`. Utilises `Coin`s to commit the trade.
- Open-Closed: `TradeAction` is extendable such that it can be extended to implement being able to sell items in exchange for currency. Also, it is open to allow more NPCs to trade with.
- Liskov's Substitution: `TradeAction` inherits from `Action` which is open to extension and thereby adheres to this principle. The `TradeAction` can be substituted with `Action` and will not throw undefined behaviour.

Assignment 2 [JTUC0006]: No further changes made to the TradeAction class, and acts the same as what was implemented during assignment 1

Wallet

-Purpose: Acts to store the actor's entire balance of collected ‘coins’ which is on the actor. -Implementation: The wallet is created as an extension of an item. This is so that a wallet can be added to an actor's inventory. This is to allow the implementation to work with all existing infrastructure which accepts arguments of actors which is seen throughout actions.

- Single Responsibility: To take in `Item`s and convert to currency. In this case it takes in coins that add to the balance. Subtracts from balance once a trade is completed.
- Open-Closed: Wallet inherits from the `Item` class which is open to extension. Wallet implements a balance attribute which allows the player to use currency. Wallet itself can be extended by a child class that wants to store other types of currency. e.g., TokenWallet which could possibly collect an `Item` - `Token`, and convert these to points or another type of currency.
- Liskov's Substitution: `Wallet` can be replaced by `Item` without any undefined behaviour.

<> Marketable

- Purpose: Adds the ability for any item that implements the interface to be able to be sold/traded.
- Implementation: Adds to the items a specified trading price that the toad will sell the item for. Additionally, it will also contain a getter method to retrieve the price from the item.
- Single Responsibility: Makes an `Item` tradable and gives the `Item` a price.
- Interface Segregation: This interface has the sole responsibility of allowing an item to be tradable at a given price. All `Item`s in the game currently need these properties so this does not violate the Interface Segregation Principle.

Assignment 2 [JTUC0006]:

No further changes made to the Marketable interface, and acts the same as what was implemented during assignment 1

SpeakAction

-Purpose: Causes any ‘speakable’ actor to ‘speak’ which inturn prints their message into the console.

-Implementation: The action targets a specific speakable class that for when the action is executed it will display in the console the classes desired dialogue. On execution the method calls the speakables method to retrieve a string of dialogue from which the speak action displays the text in the console.

- Single Responsibility: Allows an `Actor` to be able to display dialogue on the console. Instead of implementing speaking actions separately for each `Actor`

that could've been capable of speaking, makes use of an interface to give specific `Actor`s this capability.
- Open-Closed: `SpeakAction` inherits from the `Action` class which follows the principle that the `Action` class is open to extension but closed to modification.

<> Speakable

-Purpose: Instantiates a class with its corresponding dialogue and provides the ability to return a given line of communication. -Implementation: Instantiates every class that inherits with an independent set of various lines of dialogue. Provides a method to call upon one of the given lines of dialogue which can be dependent on the class's, and further, any of the world's circumstances/requirements. In the case of toad, it will be instantiated with 4 various lines of dialogue. Upon being called it will select a random line of dialogue, however, only out of select lines depending on the caller actor who will be passed as a parameter. The first line "You might need a wrench to smash Koopa's hard shells." will only be possible if the actor does not have a wrench in their inventory. The second line "You better get back to finding the Power Stars." will be another possibility if the actor has not consumed a power star and is not experiencing any of its capabilities.

- Single Responsibility: Gives a class a predefined set of dialogue to be chosen at random and displayed on the console.

Assignment 2 [JTUC0006]:

Instead of being instantiated with the 4 various lines of dialogue, it will first only be instantiated with the 2 base lines, and will then check the actor's status and items. It the actor currently has the status of being under the consumption of a PowerStar. It will add the related line, this is the same as when the items being carried by the Actor have been found to contain the Wrench Status. Meaning the final line will be added. It will then call a random line of dialogue after all conditions have been checked

---

<> Resettable

- Purpose: Upon the game being reset, all classes that implement this resettable will be affected and from which they can perform an action specific to that actor or its class.
- Implementation: The resettable interface will have two separate execute methods. One for the execution of an action specific to that object called 'excecuteInstance' and another static method used for the execution of an action that is performed upon all the object that are instances of that class 'excecuteClass'. Both methods will be passed the parameters of the game map. After execution 'excecuteInstance' will return a boolean on whether there is further execution required for the entire class. For example, the player will only need the execute instance to heal the player to their full health, however a tree will use the excecuteClass method to iterate through all the locations and change the ground where a tree occurs to dirt. This drastically improves the speed compared to every individual tree object searching through the game map for its one occurrence.

- Single Responsibility: Allows ResetManager to not have to implement an instance of to test what to perform on any object. Implementation of the class would be closed to modification following the Open-Closed Principle.

ResetManager

-Purpose: Keeps track of every resettable class in the game. Upon being called the reset manager will go through every ressetable object and execute its specific action.

-Implementation: Implemented as a singleton class, every time a class implements a ‘Resettable’ interface they will be added to the resetmanager. This easily keeps track of every object that needs to be reset without needing to check if it is an instance of. When the Resetmanager is run it will iterate through all the objects and run the excecuteInstance method. If it returns true, meaning the object has additional reset functions for the class, the method adds the object's parents to a set from which it later iterates through and runs their respective executeClass methods.

- Single Responsibility: Responsible for going through resettable each object and resets it if needed.
- Reduce Dependencies (ReD): relaxes the need to find specific objects for each resettable instance which reduces the dependencies needed.

ResetAction

- Purpose: A callable action that will allow for a game map to be reset. Will only reset the game if the actor has not reseted the game before.
- Implementation:. Upon being executed the action tests to see whether the actor calling the action does possess the “RESSETED” capability. Otherwise, if it doesn’t it will be added to the actor and have the ‘ResetManager’ run its reset method.
- Single Responsibility: Simple enforces a refresh of the current GameMap if it has not been reset before.

Assignment 2 [JTUC0006]:

The reset function performs has expected compared to the implementation within assignment 1. As it will add all objects needed to be reset to a Manager, and upon resetting the game. The player will be assigned the Reset Status. Meaning it can no longer perform the action to reset the game

Assignment 3

FountainAction

Purpose: To act as an action for allowing the player to fill their bottle with water, which can then be consumed in future turns This can be done by identifying the bottle within the players inventory. Once the bottle has been identified, the action should then Push through a new instance of the water, which will be represented as an item. This new item is then apart of the bottles stack

- Single Responsibility: Allows a player to fill up their bottle with different types of water by visited it respective fountain

- Liskov's Substitution: Replacing Action with FountainAction should not result in any undefined behaviour.

BottleConsumptionAction

Purpose: To act as an action for allowing the bottle holder to consume the water item held at the top of the stack Since the bottles level is represented through a stack, the stack will be filled with items representing the water. The action will call the .pop function to retrieve the water item at the top of stack, since all water items implement the consumable interface, it will call the waters consumptionEffect function to apply changes to the player

- Single Responsibility: Allows a player to consume the water within their bottle
- Liskov's Substitution: Replacing Action with BottleConsumptionAction should not result in any undefined behaviour.

Fountains

Purpose: To act as a ground that will allow for the FountainAction to be accessed by any actor who walks within the general vicinity of it Implementation: The fountains will be split into two classes who both extend the Ground Abstract class, each fountain class will be given the FountainAction, however, the water it provides will be respective of the type of fountain, the Health fountain will provide healing Water, and the Power fountain will provide power water

Single Responsibility: Acts as a object that will provide access to the fountainAction to actors

- Liskov's Substitution: Replacing the fountains with the ground object should not result in any undefined behaviours

---

Water

Purpose: to act as an item that will be stored within an actors bottle through the FountainAction, and will then be consumed through the BottleConsumptionAction to give the actor buffs Implementation: This item will extend the items parent class to give it the characteristics of an item that can be stored. It will also implement the consumable interface to allow it to call the ConsumptionEffect function, this is because each water has its own special effect, the Healing Water will heal the player, and the power water will increase the drinkers base attack damage. The water item is then stored in the bottles stack, and then will be called upon by the BottleConsumptionAction, in which its consumptionEffect will be called Single Responsibility: Acts as a item that will be consumed to provide special effects Liskov's Substitution: Replacing the water with Item will not result in any undefined behaviour Interface segregation: This will implement the consumable interface which allows the water to call the consumableEffect function

---

Bottle

Purpose: The bottle will be an item that will be used to store water items within its stack. The bottle will then retrieve the water from the top of its stack, and consume

it to give the actor special effects Implementation: The bottle will have a stack structure, which operates through Push and Pop. Every time the actor uses a FountainAction, a new water will be pushed into the bottles stack. And when the player uses a BottleConsumptionAction, the water from the top of the stack will be popped and consumed by the player

Single Responsibility: Acts as a item that is only used to hold multiple water items
Liskov's Substitution: Replacing the bottle with Item will not result in any undefined behaviour

---

CraftAction

Purpose: This is an action that will allow the player to craft a selection of unique weapons given that they have all 3 required items within the actors inventory Implementation: The action will first check if the actor has all 3 items (the Tree Bark, Goomba Leather and Shell Fragments). If the actor does indeed have all 3 items, it will craft the selected unique weapon (The actor will be given the option to craft all weapons within the Anvil). Once the weapon has been placed within the actors inventory, the crafting items will then also be removed. If the actor does not possess the items, it will return nothing and state that the actor does not have the items necessary to craft

Single Responsibility: Acts as an action that will allow the actor to craft weapons
Liskov's Substitution: Replacing the CraftAction with the Action will not result in any undefined behaviour

---

Anvil

Purpose: This is a ground object that will give the player the action to craft unique weapons given that they have the required items Implementation: This will be initialized with multiple CraftActions, all linked to different unique weapons that the actor can craft. This will be accessed by the actor when they walk in the general vicinity of the actor Single Responsibility: Acts as a ground that will give the player the ability to craft their unique weapons through actions Liskov's Substitution: Replacing the Anvil with the Ground class will not result in any undefined behaviours

---

Weapons

Purpose: This is a group of items that represent unique weapons that will have their own special effects within combat. Implementation: There are currently 3 unique weapons within the game, Waluigis Lucky Blade, which will have a chance to critically strike upon each hit to deal additional damage, Warios Mining Pickaxe to drop a coin with a random value upon each kill, and King Boos Lucky Scepter, which will heal the wielder for a portion of the damage done after each hit. All of these special effects will be done within the attack action depending on which weapon is currently selected by the getWeapon() function. Which will always return the weapon that is first encountered within the inventory. All of these weapons will have their own class, with unique display characters, hit rates and damage values

---

WeaponParts

Purpose: to act as filler items that serve as pre-requisites before a player can obtain a unique weapon through crafting Implementation: There are three items that are classified as weapon crafts that are needed before a player is able to craft a new weapon, this includes Shell Fragment, Goomba Leather and Tree bark. They are all obtained through several different methods and have a rare drop chance, or can be bought outright from Toadette for a large price. The goomba leather will drop from killing goombas, the shell fragment can be obtained by hitting the koopa shell without actually breaking it (ie. Using the wrench) and the tree bark will drop from trees. These are all just bare items that serve no purpose other than to be checked during the craft action, but are portable, meaning the player must pick them up and carry them to the anvil

Single Responsibility: Acts as a item that will be checked before a player can craft a weapon Liskov's Substitution: Replacing the WeaponPart items with Item will not result in any undefined behaviours

---

Flying Koopa

Purpose: To act as another hostile enemy to the Player, but one that can follow the player over walls and trees (higher ground).

Implementation: The Flying Koopa is given the ability to "Fly" through the use of Status, Ground checks that if the actor has this status or not. If the Actor has the status then the Ground lets them pass. Aside from the above, the Flying Koopa is almost identical to the Koopa enemy. When the Flying Koopa is defeated it drops to the ground and becomes a dormant Koopa shell. The Player can then attack this shell and when using a Wrench, can break the shell, which then drops a Magical Item - Super Mushroom.

Single Responsibility: The Flying Koopa is an Actor created solely for the purpose of having another enemy to attack Mario (the Player). It is created from the Actor abstract class. Open-Closed Principle: The Flying Koopa is class that is open to inheritance and the child classes can make use of its concrete methods, but it is closed to modifications, It utilises the Status to make it closed to modification but open to inheritance. Since in this case the Status enum acts as an interface. Liskov Substitution: The Flying Koopa can be replaced by its parent class without any code breaking.

Piranha Plant

Purpose: To act as another hostile enemy to the Player, however this one guards the WarpPipe's from the Player.

Implementation: The PiranhaPlant spawns on top of the WarpPipe after 1 tick in the game world. It has a powerful attack that deals 90 damage but once it is killed it reveals the WarpPipe once again for the player to use. To implement the ability to spawn after one tick, the WarpPipe only spawns the PiranhaPlant in the tick method and ensures that the tick is greater than 1 to spawn the Enemy. The Piranha Plant is unable to move so to implement this I decided to not implement the ability to wander around or the ability to follow. This is done through not giving the Piranha Plant those behaviours/actions. When the game is reset the Piranha Plants that are still alive are given a 50 hit point increase in health and healed to max HP.

Single Responsibility: To be an Actor that is rooted to the spot and only hostile towards the Player. Interface Segregation: In order to implement increasing the max hit points and healing the Piranha Plant, we make the class implement the Enemies interface which deals with resetting the class. This interface is solely responsible for these actions and is not polluted with other capabilities.

Bowser

Purpose: To act as the final boss in the game. Bowser is a super enemy with a lot of hit points and the ability to drop fires with his attacks. This makes him more difficult to defeat than normal enemies. When defeated, Bowser drops a Key item that enables the player to pick it up and unlock Princess Peach's handcuffs. This will end the game and the player will win.

Implementation: Bowser Is an enemy that drops a key upon defeat. This is implemented in the AttackAction class as it deals with when Bowser gets attacked. This also follows the SRP as the attack action only has to deal with attacking an actor and the result of attacking the actor. Next, Bowser is able to drop a fire on a successful attack that lasts 3 ticks and does 20 burn damage. To implement this I added a Fire item class that represents a fire object and created a Flammable interface for the Fire class to implement. Doing this allowed the Fire class to have the single responsibility of just being a fire that is not portable and only lasts 3 seconds. The Flammable interface allowed the fire to apply damage to any Actor standing on its location while it was still burning.

Single Responsibility: To be an Actor that is Hostile to the Player. The class for Bowser is loosely coupled to his ability to drop Fire on a successful attack as well as drop a key. This is because Bowser is not directly responsible for these events. Open-Closed Principle: Bowser utilises the openness of the Actor class which thereby makes Bowser Open to inheritance. This also means that Bowser is closed to modification and instead interfaces can be used to add changes to any of the child classes. Interface Segregation Principle: Bowser is not tightly coupled to his ability to drop Fire on a successful attack since that is handled by the AttackAction class. Fire adheres to interface segregation since it utilises a single interface that is solely responsible for burning a player and applying the damage.

Princess Peach

Purpose: To act as an Actor to save to end the game. When approached by a Player who retains a Key from Defeating Bowser, Princess Peach will give the option to win and end the game. Otherwise, she just stands there for fun.

Implementation: Princess Peach is an Actor and so inherits from Actor. Her main goal is to check that the Player has a Key so that she can be freed and end the game. This is done through her allowable actions method where she checks if the Player has an item with the Status KEY and if they do, she adds an action to end the game. Then the Player can choose to end the game. If the Player chooses to end the game, it executes the EndGameAction which removes the player from the map after printing a winning display message thereby ending the game.

Single Responsibility Principle: Responsible for checking if the Player has a Key item in their inventory. Princess Peach does nothing else except check that the Player has

a Key and presents an option to end the game. The ending of the game is handed off to another class EndGameAction. Open-Closed Principle: Princess Peach is open to inheritance, this means that we can have another actor that can also end the game. This child class can be modified as well through the use of interfaces. Liskov's Substitution Principle: Princess Peach can be replaced by the Parent class and all code will work normally.

Slot Machine Action

Purpose: To act as the action to allow an actor to gamble with the slot machine. Provides the implementation for how much is won or lost by the actor and correspondingly adjusts their wallet. Further, the action displays the corresponding results in the console.

Implementation: Slot Machine action is implemented as an extension of the Action class as well as an implementation of the TradeAction. The action defines a subclass ‘SlotElement’ to separate responsibilities in methods as well as allow for class extension potentially in the future. The program randomly generates the number of required symbols which in this case is 3. Then a method determines all the corresponding matches, which is 1 line in this case, and adjusts the players wallet accordingly if its a win. Further upon the execution completing it returns a graphical representation of the slot machine in the console with each element's symbol and coloured lines on the matching lines

Single Responsibility: Acts as an action that will allow the actor to gamble coins Open-Closed Principle: Slot Machine action is extendable such that it can be extended to work with various types of configurations of slot elements as well as extending the number of play rows/coloums which can easily be extended with various different paylines Liskov's Substitution Principle: Replacing the SlotMachine action with another action will not result in any undefined behaviour

---

Slot Machine

Purpose: This is a ground object that will give the actor the action to gamble with the slot machine. The object acts as the physical entity of the casino machine. Implementation: The Slot machine is implemented as an extension of the ground type abstract. It is instantiated with the gambling capability to discern that this entity is capable of being gambled with from an actor. It is responsible for providing a SlotMachineAction to the actor.

Single Responsibility: Acts as a object that will provide access to the slotMachineAction to the actors Liskov's Substitution Principle: Replacing the Slot Machine with another ground will not result in any undefined behaviour

---

Maps Manager

Purpose: To generate all required maps for the application and separating the responsibilities from the application class. It acts as a singleton method to prevent multiple of the same maps being spawned. It also tracks all gamemaps as such allowing for the name of the maps to be provided for the warp action due to the ‘to string’ method not being able to be edited in the engine as well as provide functionality for the reset action to look through all the available maps in a world. Implementation:

The Maps Manager is implemented as a factory using a singleton method. Its responsibility is to generate the world maps i.e. the OverWorld and Lava Zone. Instantiates two maps to determine if a map is created and what a map's corresponding name is. Get methods are created for generating each corresponding available map if its not already constructed, otherwise the factory returns the already existing map.

Single Responsibility: Provides the sole responsibility of managing all the warp destinations and assigning new targets to each warp

---

Lava

Purpose: Ground type used in the Lava Zone to separate areas. The lava hurts any actors that happen to stand upon it. Implementation: The Lava is implemented as an extension of the Group type abstract class. It is assigned additionally the damage count of 15 hit points for any actor that stands on it for every tick. As such damage to an actor once per tick that stands on the lava the tick method is used to determine whether the actor is standing on the ground and hurts them correspondingly. Additionally, if the actor has a power star affect it does not take any damage

Single Responsibility: Acts as a object that will provide the foundation of the Lava Zone and independently hurts any corresponding actors that happen to stand on it Liskov's Substitution Principle: Replacing the Lava with another ground will not result in any undefined behaviour

---

Warp Manager

Purpose: Manages every warp’s corresponding target that is linked to and assigns new warps to a corresponding target that is situated in a separate map. Implementation: The Warp Manager is implemented as a factory using a singleton method. Its responsibility is to track and assign every warp a corresponding target. As such it returns upon request target location to teleport an actor to from a specific warp source. Upon a request being called for warp where a target has not been it randomly assigns a warp that is situated on a separated on a different gameMap which is of the same ground type to the source. Single Responsibility: Provides the sole responsibility of managing all the generation of gamemaps in the world and provides the ability to determine a name of a game mao

---

Warp Action

Purpose: Responsible for warping an actor from specific source location to its corresponding target warp location. Also displays to the actor the corresponding GameMap of where the warp is directed before going through the warp. Further, the action manages removing an actor from the target location to happen there be one present at the point of warping Implementation: The Warp Action is implemented as an extension of the Action class. It is instantiated with a direct association to a WarpDestination representing the respective source of where the action is called. The warp action also has the responsibility after being executed to set the targets of the warp manager of the source and target warp destination to each other respectively. This allows for the actor to warp back to the previous warp location after warping.

Single Responsibility: Acts as an action that will allow the actor to warp to a difference location Liskov's Substitution Principle: Replacing the warp action with

another action will not result in any undefined behaviour

Warp Destination

Purpose: Acts as an Abstract class that any warpable ground is extended from. This allows for the number of possible warp objects to be further extended in the future allowing for ground types such as doors, sewer drains, caves ect to be implemented and to be easily integrated with the warping system of the game Implementation: Warp Destination extends from the Ground abstract class and sets itself as an abstract class. It extends from the ground type by instantiating a location parameter to be now instantiated in the parameter. This allows for the warp manager to now immediately assign as a possible warp target that can be called towards.

Single Responsibility: Open-Closed Principle:
Interface Segregation Principle: Liskov's Substitution Principle Dependency Inversion Principle: Makes it such that lower exact implementations can work with the warp manager without the specific details of those lower classes

---

Warp Pipe

Purpose: Acts as the spawn point for all Piranha plants as well as being a warp point location that can be warped from and towards. Its responsible for the spawning of new piranha plants as well as providing a warp action to an actor. However, the warp action should not be present while a piranha plant is present. Implementation: The Warp Pipe extends from the Warp Destination abstract class as well as ResettableInstance and Jumpable. It is instantiated with a boolean count to discern whether it has spawned a Piranha plant. It is also responsible for restricting whether the actor present can warp. It does such by checking if there is a player on top and if so whether its the actor calling the action

Single Responsibility: Acts as a object that will provide access to the warpaction to actors as well as providing the spawn location for all piranha plants. Liskov's Substitution Principle: Replacing the Warp Pipe with another corresponding warp destination will not result in any undefined behaviour