Justifications Assignment 3 - Group 3

Requirement 1 - Lava zone (Philip Ooi)

This task required two new elements into the game, an alternate lava map and a warp pipe which allowed the player to be able to travel to the secondary lava map.

For the implementation of the secondary game map. This was instantiated within the application class. All map implementations will be siloed into the application class in order to reduce dependencies. The maps will then be accessed through the application class through static variables, allowing for further decoupling of the classes. This allows the game maps to be accessed system wide without the need to parse the map to different classes which would cause large amounts of repeated code in any future class that requires the map which adheres to the DRY principle.

Pipes were implemented using a combination of two techniques. Pipe extended the ground class which allowed it to inherit all the class attributes and methods allowing us to implement the abstraction principle of object oriented programming. Lava was then easily implemented by changing the map and running it through the ground factory, adding a new Lava() into its parameters. The teleportation method of the pipe was done through the usage of the static GameMap that was created in the application class as mentioned before. This allowed us to pass the game map to pipe without the creation of new methods to parse the information into the class which would have caused a large amount of dependency. The coordinates of the pipe in order to facilitate teleportation back to the correct pipe was done through a singleton class called Teleportation singleton. This singleton class was built in the data type of a stack. This was done in order to adhere to the open close principle in preparation for future warp locations and depth of warping using a Last In First Out strategy.

This is achieved through creating two different stacks, one for map and one for locations and stacking each warp onto each stack. When it is time to warp back, the top of each stack is popped off and the player is warped into that location. This allows for future implementations of maps as the new locations could be added to the stack, keeping the order of the warps.

Requirement 2 - Enemies and Allies (Amogha Raviprasad and Philip Ooi):

Amogha

This task introduced a few different enemies (Bowser, Flying Koopa and Piranha Plants), and a new ally (Princess Peach).

For Bowser's implementation, he was introduced as a new enemy that extended from the enemy class. Much like the earlier enemies from assignment 2, he has the same follow behaviour and attack behaviour (but omits the wander behaviour in the constructor, as he is always guarding Princess Peach). In particular, Bowser has a key in his inventory that is initialised in his constructor. This means that when he is killed, he drops this key, which the player can then pick up to use later.

The Flying Koopa was also added as an enemy, but there is a much different implementation here. There is already a Koopa class, with Flying Koopa having the same implementation (except having a different health and also being able to fly over high ground), so we had an abstract class called AbstractKoopa that both Koopa and Flying Koopa extend from. This is

to prevent redundant code (therefore following the DRY principle). The abstract class had the majority of the implementation, with Koopa and Flying Koopa having different constructors. In particular, Flying Koopa had a CAN_FLY status added to it, so that in the higher ground classes (walls and trees), these grounds could check for an actor with a CAN_FLY status and would therefore allow a simple move instead of a jump.

Piranha plants do not have a follow or wander behaviour, as they are stuck on a pipe. Instead, they merely implement attack behaviour in the AllowableActions() method, so that they can attack the player in close proximity. The piranha plant is spawned in the Pipe class, where we have a counter to check for the second turn, and then spawn a piranha plant on each pipe. Once the piranha plant is defeated by the player, it is removed from the map (in the AttackAction class) and the pipes are then open for teleportation.

Princess Peach is an ally and therefore does not have any wandering, following or attack behaviours in her class. Instead, in her AllowableActions() method, she checks for an actor next to her. If that actor has the HAS_KEY status, then we create a new object of the WinGameAction() class. Note that the player gets the HAS_KEY status when they kill Bowser and pick up the key that he drops. This occurs through a Key class, where upon pickup, the capability is passed onto the player, as it is added to the key in its constructor. Princess Peach can then be talked to if the player has the HAS_KEY status (implemented via a simple if check) and the game finishes.

Philip

Reset functionality was implemented into the Bowser and Piranha plant where the two classes implemented the Resettable interface. This allows the class to adhere to the abstraction principle and allow the classes to inherit methods through the class which allows for easy implementation of a new resettable item by only overriding its resetInstance() function allowing it to run properly, with the registerInstance() method adding it to the reset ArrayList().

Requirement 3 - Magical fountains (Ollie Hiscoke)

This requirement introduced a few new elements into the game including two magical fountains and their corresponding waters as well as a bottle for the player to store the water.

For the water section of the fountains, an abstract water class was created with function relating to all waters. Having this abstract class follows the open closed principle as if a new water is to be created in the future, it can just extend the water class and have the required methods and attributes without having to edit the water class itself.

The fountains were created as extensions of the ground class in order to allow actors to enter (mostly player to collect water) and in order to access the tick method. This allowed for water to be created on the fountains when an actor is standing on it and if there is no other water on top of the fountain. This meant that the option to pick up water would only appear once in the menu per turn when the player is standing on it and would continue to create unlimited water.

The bottle was made as a singleton instance as there is only one required which is in the player's inventory from the start. Having this singleton instance allows for the bottle's contents to be updated and manipulated throughout the game without the need to create a new bottle every turn. The stack implementation of the bottle also allows it to act as a bottle would somewhat by taking the last entered water first when consuming.

Two new actions were created for both filling the bottle and drinking the water. Both extend action in order to get the methods from them. The fill up bottle action uses the singleton instance of the bottle and the water the player is trying to collect and adds the water to the contents of the bottle at the top for the player to use. The drink water action also uses the singleton bottle and pops off the top water and applies the effects to the player.

Requirement 4 - Fire Flower and Fire Attack (Amogha Raviprasad):
The fire flower was added as a separate class extending from the Item class, as it can be picked up and used to get a fire attack. Firstly, simple if statements were added to sprout and sapling to spawn the fire flowers. When the sprout was grown into a sapling or a sapling into a mature, there would be a 50% chance that a fire flower would be spawned on top of the plant. The player is then able to pick up and thus consume a fire flower that has been spawned on a plant. The fire flower has a capability FIRE added to it, which can be passed onto the player once they pick it up (meaning they now get its effects). Additionally, the Fire flower has a tick method which checks for the number of turns the player has used it for. Once 20 turns are over, the flower is then removed from the player and the player no longer has the capability FIRE. While the 20 turns are not over yet, the player has the ability to do a fire attack.

Fire Attack is a different version of a normal attack, where a normal attack occurs but a fire is dropped at the target's ground. This fire deals 20 damage and remains for 3 turns. This fire attack effect, when provided by the fire flower, lasts 20 turns (as checked in the Fire flower's tick method). In the player class, there was a simple for loop to check through the items in the player's inventory. If the item had a status of FIRE, meaning it is a Fire flower, the player then gets the effects of this fire flower by adopting the status. From then on, if the player attacks, there is a check in AttackAction for the status FIRE, which enables the player to then drop fire. There was some consideration here in implementing a new class called FireAttackAction, where Fire would be dropped while the attack occurs. However, this would have resulted in a lot of repeated code, thus a simple check was just added into AttackAction.

Fire itself is a new class that extends from ground. Within this Fire class is where the damage is set, as well as the check for 3 turns, where the previous ground is then set on the fire's place. It is important to note that when the Fire is set within the AttackAction class, the current ground at that location is noted and passed in as an argument to the Fire class. This way, the Fire can be reset with the previous ground (as it could have been a Wall, Dirt, etc.) after the 3 turns are over.

The above implementation of the FIRE status is also used by Bowser, as he has a fire attack as well. By having a separate Fire class, we are thus able to prevent repeated code (Bowser and Player don't have to both implement Fire separately), adhering to the DRY principle.

Requirement 5 - Speaking (Ollie Hiscoke):
In order to allow actors to speak, an interface called speech capable was created and all actors that had this ability implemented this interface. This interface only had one method in it which was the speak method. This was to ensure that the code followed the interface segregation principle by only having important interfaces and ensuring they were not too large and only represent one quality, in this case speech.

An interface was chosen as all actors would have the same method however the implementations of this method would be different for each as they all say different lines. This was also done to ensure that the open closed principle was followed by allowing other actors in future to implement the speak function without need to edit any current classes.

Finally, the random function was used to ensure the line the actor says is a random one of the provided lines. Switch statements were avoided in this section to remove the chances of a code smell in case of further development down the line. The speak function was called in playturn of all actors, only if the added age attribute was a multiple of 2. The age was also incremented here to avoid problems with this during execution.