FIT2099 Assignment 3 Design Rationale

- Linden Beaumont, Ricky Zhang & Rebekah Fullard

REQ1

How it works:

DamagingGround - Is an abstract class that extends ground. Subclasses of DamagingGround hurt actors for a specified amount when they walk on top of them.
Lava - Is a subclass of DamagingGround that hurts the player for 15 damage when walked on. Only actors with the LAVA_WALK status are able to walk on lava (i.e: Mario).
Pipe - Each pipe object stores 3 Location attributes; pipeLocation (location of current pipe), lavaPipe (location of the pipe in the lava map), and initialPipe which is the stored location of the pipe where mario came from. The initialPipe attribute for each pipe is set as null in the constructor. The pipe in the lava map will have its initialPipe attribute set by TeleportAction with each teleport from the original game map. Each pipe also has an attribute of type Pipe (itself). This attribute corresponds to the pipe in the lava map and is used by the TeleportAction. The pipe also stores the lava map as a GameMap attribute which is used as a parameter in the instantiation of TeleportAction. Pipe also extends the HigherGround abstract class since the player needs to jump to the pipe.
TeleportAction - The TeleportAction class stores the lava map attribute as GameMap and an attribute of type Pipe from the pipe mario is going to jump through. The actual action for teleporting between pipes starts with checking if the initialPipe attribute of a pipe is null. In the case with pipes on the original world, they will always be null. If the attribute is null then it checks if there is an actor (piranha plant) on the pipe and if there is, the actor is removed from the map. After teleporting to the lava map, the initialPipe attribute of the lava pipe is set to the location of the pipe. When teleporting back to the original map from the lava map, the actor is moved to the initialPipe location (in the original map).

Why it works that way:

The Pipe is kept separate from the TeleportAction in that the pipe is not concerned with the actual action of teleporting the player but instead that is left for the TeleportAction class. This follows the single responsibility principle.
Pipe extends the HigherGround class which allows it to be able to be jumped to without modification of the HigherGround class or JumpAction. Similarly, Lava extends the DamagingGround class without modification to the DamagingGround class. This follows the open-closed principle. This also results in minimal repetition, because the functionality defined in the parent classes is used by the child classes.

REQ2

How it works:

Princess Peach - Peach is an Actor that is spawned at a particular Location when the game starts. She cannot move, attack, or be attacked. She implements the CanSpeak interface so that you can speak to her to end the game. The ability to speak to her can only be achieved if the player is in possession of the Key that drops when Bowser is killed. Her allowableActions() method checks if the other actor has the Status CAN_END_GAME (which they only have while carrying the Key) and if they do, they will have an EndGameAction, which speaks to Peach and ends the game.
Bowser - Bowser is an Enemy that is spawned at a particular location when the game starts. He will not move or attack until the player approaches. When he attacks, a FireGround (extends DamagingGround) is spawned on the Location of the target. Once the player has approached Bowser, Bowser will follow them and attack whenever possible. When the game is reset, Bowser returns to his original position, is returned to full health, and no longer follows the player.
Piranha Plant - Piranha Plant is an Enemy. On the second turn of the game, Pipes spawn a Piranha Plant on their Location. This Piranha Plant will never move and will attack the player if they approach. Once the Piranha Plant on a Pipe is defeated the player can get on and go through the Pipe. When the game is reset all Piranha Plants that are still alive on the map gain 50 to their max HP and are fully healed to the new maximum.
Flying Koopa - Flying Koopa is a Koopa. It has more health than a NormalKoopa, and can fly over HigherGrounds when wandering and following. The program has been changed so that there is now an abstract class Koopa that extends Enemy, and FlyingKoopa and NormalKoopa extend Koopa.

Why it works that way:

The implementation of DamagingGround as an abstract class avoids unnecessary repetition because it allows for more types of ground to be added without needing to implement the code for damaging an Actor separately in each class. It also follows the OpenClosed principle by allowing more classes that extend DamagingGround to be added without needing to modify DamagingGround.
By using the Status CAN_FIRE_ATTACK to allow classes to perform a FireAttackAction, this follows the Open-Closed principle because the program needs to be extended so that new classes can make a FireAttackAction, rather than modifying the existing code to check for instances of each type of class, those classes can simply use addCapability(Status.CAN_FIRE_ATTACK).
The implementation of the abstract class Koopa, with subclasses NormalKoopa and FlyingKoopa avoids violation of the Liskov Substitution principle because FlyingKoopa does not just extend the original (concrete) Koopa class. By implementing another level of abstraction, NormalKoopa and FlyingKoopa use all of the common functionality defined in Koopa, without resulting in any need to implement conditional logic in the parent class to check if the Koopa is actually a FlyingKoopa.

REQ3

In order to allow the power fountain to modify the base attack damage of actors I had to create a new type of actor which can have a modifiable intrinsic weapon. This is a perfect example of the open close principle since the actor class was open to extension (but not modification). Ideally, the abstract class Actor would allow for this. This implementation also follows the DRY principle because it removes the need to override the getIntrinsicWeapon() method in each Actor subclass.

By having an overarching abstract fountain class I was able to reduce the complexity of my code, because if in the future 100 more types of fountains were added the rest of the classes would not need to account for each specific fountain, but just the one overarching class. This also follows the Dependency Inversion principle because the DrinkAction and FillBottleAction classes can interact with the abstract class Fountain instead of needing to rely on the concrete classes. This also avoids repetition as all common functionality is defined in Fountain so it doesn't need to be repeated again in each subclass.

A similar thing can be said as to why there is only one refill bottle action and one drink action even though there are multiple types of water than a fountain can contain. This means the actions do not need to check what type of fountain is being filled/drank from, as they only need to interact with the abstract class so that the Dependency Inversion principle is maintained.

Because there should only ever be one Bottle that has been implemented as a singleton, however the Player still needs it in their inventory to be able to use it. The Bottle uses the existing java class Stack to store and access the contents of the Bottle. This use of an existing class eliminates the complexity that would be added to the program by creating our own Stack-like class. It also results in less code being written, because if an ArrayList(for example) was used for this purpose, accessing and utilising the correct elements would be more complex. As it is, we can simply use the existing methods in the Stack class.

CREATIVE REQ4

How it works:

Corpse - A class that extends enemy which can be 'reanimated' to a zonbi. Actors which have the capability HOSTILE_TO_PLAYER are able to be turned into corpses upon death. The actor is removed from the map and replaced with a corpse in the execute() method in AttackAction. An Actor with the capability CAN_REANIMATE will have the ReanimateAction added to their ActionList when next to a Corpse. The corpse is removed from the map after 5 turns and with each turn it checks if the timeRemaining attribute is equal to 0 before removing from the map, if not, it minuses 1 from the attribute.

ReanimateAction - Extends action class and removes the corpse from the map and replaces it with a zonbi.

NecromancyWeapon - A weapon that extends the WeaponItem abstract class. The NecromancyWeapon adds the CAN_REANIMATE status which thus allows the ReanimateAction to be used on corpses. The NecromancyWeapon also checks if the player has the status KILLER, and if so, increases the damage and chance to hit of the weapon.

Zonbi - A class that extends actor. The zonbi class checks if there is a surrounding actor that has the status HOSTILE_TO_PLAYER, and if there is, moves towards that actor (e.g: a koopa or goomba). The zonbi has the AttackAction and checks if the actor it has moved to contains the aforementioned status. The zonbi will then attack the actor if it has the status. The zonbi has the WanderBehaviour so it will wander around if there are no actors around the zonbi.

RuinWall - A simple class that extends the HigherGround class. This class is used for the walls that surround the necromancy weapon. The ruin wall has a low success rate for jumping and a high fall damage (10% success rate, 40 damage)

GuardKoopa - A simple enemy that stands in front of the necromancy weapon ruins. It extends the koopa class and has 100 hit points. This koopa does not have any wander or follow behaviours so it will just stand guard at the ruins.

Why it works that way:

Implementation of this functionality follows the open-closed principle since the newly added classes did not require modification of existing classes.
The dependency inversion principle is also being followed since lower-level classes are being abstracted from higher-level classes such as zonbi, which extends actor.
Having specific classes for specific functionality (i.e: zonbi, corpse, guardkoopa) satisfies the single responsibility principle. Functionality for zonbi and corpse are split within different classes as well as the GuardKoopa class having its own functionality while still extending the Koopa class.

CREATIVE REQ5

How it works:

Shovel is an Item that can be bought from Toad. Shovel has the CAN_DIG status, so when a Player is on a SoftGround (x) they will have the ability to dig there. The allowableActions() method of SoftGround checks for this status. If present it will give the Actor a DigAction that they can choose. The DigAction is an Action that, when executed, has a 75% chance of spawning a Coin that can be picked up but a 25% chance of spawning Spores in a 3*3 square centred on the Actor's Location. The Spores damage Actors by 5 HP each turn they end on a Spores Ground. After 10 turns the Spores disappear. A SoftGround can only be 'dug' once - it turns back to Dirt after a DigAction is taken on it.

Why it works that way:

This REQ followed the Open-Closed principle because Spores and SoftGround extend Ground without needing to modify that class.
This REQ follows the Single Responsibility principle because each class has its own specific responsibilities. For example, the DigAction can spawn Spores, but does not manage any of the effects of the Spores, nor their longevity on the map. That functionality is maintained by the Spores class.
This REQ maintains the Dependency Inversion principle because each of the new classes (Shovel, Spores, SoftGround, DigAction) are abstracted by parent abstract classes. Because of this, adding these classes has not created additional associations/dependencies for high-level classes.
The implementation of this REQ has also included the addition of the Buyable interface. In Assignment 2 feedback it was noted that BuyAction should not be able to sell any type of Item. The Buyable interface allows the implementation of Shovel to fix this problem. BuyAction now has an association with Buyable instead of Item - so it can only sell Items that implement the Buyable interface. This means that the implementation still adheres to Dependency Inversion principle because particular Buyable Items are abstracted from BuyAction by the Buyable interface. It also means that it maintains the Open-Closed principle - because any Item that implements Buyable can be sold by BuyAction without any modifications to BuyAction.