Design Rationale

1. **Assignment 2 Req 1,2,3 and 7. Added Util Class to compute hit rate.**
   Throughout the game, we have many classes that involve a mechanism of performing some action, such as Sprout having 10% chance to spawn Goomba, jump success rate, attack hit rate and more. Instead of doing the computations inside the respective classes, we have decided to create a util class that does the job for us. Thus, we have reduced duplicate code which is in alignment with the DRY principle. We have also implemented the Single Responsibility Principle (SRP) by making the computation of hit rates separate from the respective classes.

2. **Assignment 2 Req 1. Tree is an abstract class. Sprout, Sapling and Mature inherit from Tree.**
   We have decided to make Tree an abstract class because we do not want to instantiate any Tree objects. The Tree class acts as an abstract class and provides the foundation for each type of Tree, which are Sprout, Sapling and Mature. Tree has methods that are shared across all types of trees such as reset(), due to trees being able to become dirt when the game is reset. Storing methods that are the same inside Tree instead of inside Sprout, Sapling and Mature is an example of Do Not Repeat Yourself Principle (DRY).

3. **Assignment 2 Req 1. Removed SpawnItem and SpawnActor interfaces from assignment 1.**
   Upon further examination of the engine code, the only method available for us to spawn things(item or actors) is inside tick(). Hence, it is difficult to create an interface to allow us to spawn things that works across different ground types, since each ground type may spawn things at different intervals and or conditions. Even if we did create the interface, the interface would not be flexible enough to be implemented by other grounds. Thus, we just scrapped the idea completely.

4. **Assignment 2 Req 2. JumpableTerrain interface and Jumpable Interface.**
   In the game, we have different types of grounds such as Dirt, Wall and Tree. To differentiate the terrains that can be jumped over, we have implemented a JumpableTerrain interface. Grounds that implement this interface can be jumped over, and the interface can have methods such as getFalldamage(). In the future, if we had more types of grounds and wanted them to be able to be jumped over, those grounds could simply implement the JumpableTerrain interface, thus making code that can be extended easily.

   Next, we have the Jumpable interface. Instead of just coding a jump method inside Player, we decided to implement an interface so that any actor can jump. For example, to allow enemies to be able to jump, the Enemy class just needs to implement the Jumpable interface. Jumpable interface can have a jump method that allows an actor to jump. Although the Jumpable interface is empty for now, we may want to have different types of jumps in the future, such as lowJump() and or highJump().

The two interfaces complement each other nicely as well. To perform a jump, we first check if the ground implements the JumpableTerrain interface. Then, we just check if the actor implements the Jumpable interface. The idea of implementing an interface rather than just coding a method is in line with the Open-Closed Principle. We can make an enemy be able to jump without having to modify the enemy class!

5. **Assignment 2 Req 2. Further analysis on JumpableTerrain.**
   To determine if a ground can be jumped over, we have thought of a few ways to do it. The first way is to use a boolean flag (instance variable or add a capability) to indicate that this ground can be jumped over. However, since we would like to have methods such as getFallDamage() that are common across all grounds that can be jumped over, a boolean flag is not sufficient. Hence, the next idea is using an abstract class.

   An abstract class called HighGrounds that inherits from Ground, and then Tree inherits from HighGrounds and Sprout that inherits from Tree. The idea of an abstract class is a viable solution, however, we would have up to 4 levels of inheritance, which is a code smell. (Sprout -> Tree -> HighGround -> Ground). Hence, we reject the idea of an abstract class due to too many levels of inheritance.

   Lastly, we have our JumpableTerrain interface. Grounds that can be jumped over should implement this interface. It provides similar functionality as the abstract class idea, but now we only have up to 3 levels of inheritance.

6. **Assignment 2 Req 3. Enemy is an abstract class.**
   Since enemies are a common thing in a game such as Mario, it would make sense to implement an abstract class just for the various enemies. This would prevent enemies needing to remake identical methods for enemies with similar properties. For example, it is aggressive towards the player, which would violate the Do Not Repeat Yourself principle. Furthermore, the code would be much easier to implement and maintain in the long run in case more enemies are added in the game.

7. **Assignment 2 Req 3. Enemy has a relationship with Behaviour.**
   This is because all enemies in the game have an attack behaviour, wander behaviour and follow behaviour. Hence it would make sense that the abstract class which all enemies will inherit contains an arraylist of behaviours, indicating what actions an enemy should take in its current state. This complies with the reduce dependency principle as any new enemies created will not need to create another dependency with Behaviour interface

8. **Assignment 2 Req 3. DestroyAction has a relationship with SuperMushroom**
   I believe DestroyAction should have a relationship with SuperMushroom because whenever a Koopa is killed (shell destroyed), it calls DestroyAction, indicating it was

destroyed. DestroyAction will then remove the shell and spawn a SuperMushroom for the player to pick up in its place. Hence, the way I would like to implement it is whenever a koopa is killed, a new instance object of SuperMushroom to replace it.

9. **Assignment 2 Req 4. MagicalItem is an abstract class**
   Magical items are special power ups that can provide unique buffs to the player, they can be either bought from Toad via trading, or picked up by the player from the ground. MagicalItem interacts with ConsumeAction as only magical items can be consumed by the player and it also introduces a counter for items that have a limited buff duration, such as PowerStar. This complies with the Don't Repeat Yourself principle as all magical items that inherits MagicalItem no longer need to code out the action which allows interaction between these items and ConsumeAction.

10. **Assignment 2 Req 4. ConsumeAction has a relationship with MagicalItem and Player**
    Whenever a player consumes a magical item, ConsumeAction will then perform a check on which magical item has the player consumed. Once discovered, it will return the appropriate status (buff) and stat changes provided by the consumed magical item and return it to the Player. Then, it will perform a check to find if the consumed magical item was from the ground or the player's inventory and promptly remove/delete it.

11. **Assignment 2 Req 5: Super Mushroom, Power Star and Wrench implements the Tradable interface.**
    Tradable interface is created so that any object that can be traded in the future can implement this interface. Instead of creating a Tradable Item class, it would be much more efficient if we use the Tradable interface. For instance, if we want a weapon which is not an item to be traded, then the weapon can simply implement the Tradable interface to have the methods that are required to perform trading. In addition, the weapon can also add features in its class without changing the existing code in the interface or other classes. This complies with the Open Closed Principle, where the interface is open for extension but closed for modification.

12. **Assignment 2 Req 5: Trade Action extends the Action abstract class to allow the player to have an action to buy a tradable item.**
    When the player is in the surroundings of the Toad, the Toad allows the player to have a TradeAction to buy an item. In order for the TradeAction to perform the transaction, it needs to get the balance from the player's wallet and the price of the tradable object. If the balance is enough to trade an item, the item is added to the player's inventory. Thus, the trade action depends on the player's wallet and the tradable object to perform the transaction.

13. **Assignment 2 Req 5: Coin inherits from Item abstract class.**
    Coin extends the functionality of Item abstract class because the coin is a physical object on the ground which is an item. To pick up the coin, we need a PickUpCoin Action to pick the coin up and add the amount of the coin into the player's wallet, then remove it from the ground. The reason why we add the amount into the wallet

instead of adding the coin itself into the inventory is that we do not want the player's inventory to be filled with coin instances.

14. **Assignment 2 Req 5: PickUpCoinAction inherits from the Action abstract class.**
    The PickUpCoinAction should be inherited from the Action abstract class because PickUpCoinAction can do the task provided in the Action abstract class. It should not be inherited from PickUpItemAction class, because the PickUpItemAction adds the item into the player's inventory but the PickUpCoinAction is not intended to add the coin into the inventory, instead the amount is added into the player's wallet. So if the PickUpCoinAction class extends the PickUpItemAction class, it cannot do the task of its base class, aka PickUpItemAction, which in this case breaks the Liskov Substitution Principle(LSP).

15. **Assignment 2 Req 5: Wallet class is used to store the balance of the player's wallet.**
    Wallet is separated as its own class because it has its single responsibility to only deal with the balance of the player, e.g. update the balance when the player picks up the coin from the ground, or when the trading with toad occurs. This enforces the Single Responsibility Principle (SRP) where one class has only one responsibility. If we do all the balance updates in the player class, the player class will become a large class which has many responsibilities to carry, thus it violates the design principle of single responsibility. In addition, one actor can have only one wallet to store the balance they currently have. When the player is first instantiated, a static factory method of getInstance() will be invoked to create a wallet for the player. After the first instantiation, the wallet instance will be updated throughout the game.

16. **Assignment 2 Req 6: Monologue class stores a list of sentences that are available and performs the logic to choose a random sentence.**
    When the player is in the surroundings of the Toad, the player has an action to speak to the Toad, so we need a Speak Action that extends the Action abstract class. When the speak action is executed, the monologue will pick a random sentence to be spoken out. The monologue is basically a manager class that manages the speak action, which in this case stores a list of available sentences and manages what sentence to speak out. The sentence that is spoken is dependent on the items that the player is holding at the moment. To check if the player is holding a specific item to avoid a specific sentence, we can add the capability to the item and check if the player has the capability as what the item has. Instead of getting the inventory from the player and checking if the item exists in the inventory which requires more methods involved, checking with capability directly can reduce the dependency of codes.

17. **Assignment 2 Req 7: Reset Action requires a Reset Manager to reset the game.**

Reset Manager is a singleton manager that does all the reset tasks to the instances that implement the Resettable interface. Only one instance of reset manager will be instantiated throughout the game and the player can only reset the game once. For any classes that are resettable, they will be appended to a list of resettable instances and being reset by the Reset Manager when the hot key for reset ’r’ is chosen by the player. The Reset Action then calls the reset manager to reset the instances by invoking the run method in the reset manager class. In order to allow the player to reset the game only once, when the reset manager first runs the reset tasks, there is a counter to keep track of the number of reset being invoked. If the counter is 1 means the game has been reset once before, then the player will not have the Reset Action anymore. This design abides by the Single Responsibility Principle where the reset manager is only responsible to reset the game. If the reset tasks are implemented in the player class, then the player has too many responsibilities to deal with, thus it violates the Single Responsibility Principle.

—--

Beginning of Assignment 3.

1. **Assignment 3 Req 1. Lava is its own class. Lava inherits from the Ground class.**
   At each turn, Lava does damage to the player that stands on it. Thus, we can override the tick method to hurt the actor standing on it. Lava is different from other ground types, so it belongs to its own class. This is related to the Single Responsibility Principle (SRP).

2. **Assignment 3 Req 1. Warp Pipe implements the Jumpable Terrain interface.**
   A warp pipe can be jumped over. Thus, by implementing the interface, we do not have to rewrite the code used to check if an actor can jump. This is in line with the DRY principle.

3. **Assignment 3 Req 2. GroundKoopa and FlyingKoopa inherits from KoopaType abstract class.**
   A flying koopa is just a regular koopa with an increased max hp and the capability of flight. Hence, the Koopa class from assignment 2 has been converted into an abstract class named KoopaType while the creation of a normal non-flying koopa is relegated into GroundKoopa class. The abstract class’ purpose is to comply with the Don’t Repeat Yourself Principle (DRY) as both flying koopa and normal koopa share a lot of code and methods.

   Additionally, there is no flyableTerrain interface and flyCapable interface. The reason is that we do not want to over engineer the game. If Mario needs to fly, then we will have a flyableTerrain and flyCapable interface. However, at this point in time, Mario cannot fly. Thus, we decided that it is sufficient to just use a capability as a boolean flag to mark actors that can fly (Flying Koopa).

4. **Assignment 3 Req 2. Bowser has a dependency with Key.**
   In order for Mario to obtain the key to free Princess Peach, he must defeat Bowser and get it from him. Hence, we add a key into Bowser’s inventory at the start of the

game and once Mario defeats him, Bowser will drop the key from its inventory for Mario to pick up.

5. **Assignment 3 Req 2. PrincessPeach has a dependency with Key and WinAction.**
   Whenever Mario approaches Princess Peach, she will perform a check to see if Mario's inventory contains a key. If yes, she will allow Mario to interact with her, freeing her and performing WinAction (the win condition).

6. **Assignment 3 Req 2. AttackAction has a dependency with ArsonAction.**
   Whenever an actor attacks and has the capability ARSONIST, AttackAction will call ArsonAction to ignite the target's current ground on fire. The benefits of having AttackAction call ArsonAction is to allow an actor to perform two tasks simultaneously, which is attacking the target while setting the target's ground on fire.

7. **Assignment 3 Req 3: Bottle is a magical bottle that stores infinite capacity of magical water.**
   Bottle is a magical item that stores a list of magical waters(e.g. Power water and Healing water). The RefillCapable interface is implemented by the Bottle class because the bottle can be refilled with magical waters. Thus, the logic of filling up the bottle should be overridden in the fill method in Bottle class. The benefits of having this interface is that in the future if other items are capable of being refilled, they can simply implement this interface, thus there's no need to make modifications to the interface but to implement the method itself in its own class. Therefore, this complies with the Open Closed Principle.

8. **Assignment 3 Req 3: The abstract Fountain class extends the abstract WaterSource class, and the WaterSource class extends the abstract Ground class**
   Water Source is a ground that allows other classes that are also a source of water to extend this water source abstract class. By having this abstraction, we can reduce the dependency between different classes. For example, the FillAction does not need to know what concrete class of water source is involved in filling the water, instead it should only depend on the abstract water source to execute the logic behind the fill action. This complies with the Dependency Inversion Principle(DIP) because the fill action class depends on the abstraction layer of the water source. The Fountain which is a type of water source becomes the subclass of the abstract Water Source class. In addition, the power fountain and health fountain extends the Fountain abstract class because both of them are a type of fountain.

9. **Assignment 3 Req 3: Player has a fill action to refill the bottle with magical water when the player stands on the fountain, and has a drink action to drink the water.**
   Both FillAction and Drink Action extends the Action abstract class. When the fill action is executed, the magical water from the fountain will be filled into the player's bottle so that the player can drink the water to gain extra powers. Since fill action is not only to fill the bottle, it can also be used to fill different containers, so for different containers there's a different fill method. Since the container that is refillable should

implement the RefillCapable interface, each container should have its own fill method. After filling the bottle, the player will have the action to drink the magical water from the bottle. To drink the magical water, the consume method in the magical water class will be executed.

10. **Assignment 3 Req 3: Enemy has a drink behaviour that allows him to have the drink action to drink the magical water from the fountain.**
    The enemy has drink behaviour as the first priority if he stands on the fountain. At the moment when he stands on the fountain, he can perform a drink action to drink the water from the fountain if the fountain contains the water. Once he drinks the water, he cannot drink the water anymore if he doesn't move away from the fountain(wander). Each time, the enemy can drink up to 5 water slots depending on how much water is remaining in the fountain. The behavioural priorities for the enemy is Drink Behaviour → Attack Behaviour → Follow Behaviour → Wander Behaviour. The Drink Action extends the abstract class Action so that the enemy can perform the action if he has the drink behaviour. The reason why the drink action is created instead of using the consume action is because drinking involves liquid type of item e.g. water, whereas consuming involves solid type of item, e.g. super mushroom. If we use consume action to perform drinking, we will need to modify the consume action class quite a lot which violates the Open Closed Principle and the class will have too many tasks to deal with, which violates the Single Responsibility Principle.

11. **Assignment 3 Req 4: SwimmableTerrain interface and Swimmable interface.**
    The rationale behind this is exactly the same as Jumpable Terrain and Jumpable from Assignment 2. A ground that can be swam over (River) should implement the SwimmableTerrain interface. It checks if an actor can swim by checking if the actor implements the swimmable interface. In the future, actors that can swim only need to implement the swimmable interface. Grounds that can be swam over only need to implement the SwimmableTerrain interface. This is in line with the Open-Close principle (OCP). Although both interfaces are relatively empty now, we decided that actions that Mario can perform are important to the game and must be carefully designed and engineered.

12. **Assignment 3 Req 4. Added Ability Manager class.**
    Wet status has a timer of 20 turns before it expires. We could implement the countdown inside Player's playTurn method, but the code would not be extensible when the player has multiple statuses. Thus, we created an abilityManager class that handles the timer for statuses that have a countdown (WET and INVINCIBLE). The ability manager class separates the countdown of statuses from the actor. Thus, this is in line with SRP. Any statuses that have a countdown can just be added into the ability manager without having to modify the actor's playTurn, thus this complies with OCP.

13. **Assignment 3 Req 5: Bucket implements RefillCapable interface.**
    Bucket and Bottle are both refillable items in the game, hence they share some common methods. Thus, it would make sense to create the RefillCapable interface to ensure standardised method names and arguments are used and overridden. By

using abstraction of the RefillCapable interface, we can have a fill method inside the interface to allow bottle and bucket to be filled up respectively. The layer of abstraction ensures that fillUpAction doesn't need to know the concrete implementation of how the bottle or bucket is filled up. Thus, this is related to DIP.

14. **Assignment 3 Req 5: River inherits WaterSource abstract class which has a relationship with FillAction.**
    There are multiple water sources in the game, which are PowerFountain, HealingFountain and River. All of these water sources commonly share the capability of letting the player refill up either their bucket or bottle depending on the water source using FillAction. Hence creating the abstract class WaterSource would allow FillAction to filter out the types of grounds to only include refillable water sources.

15. **Assignment 3 Req 5: PourAction has a dependency with Bucket, Fire and Location.**
    When the player is near a ground which is set on fire while holding a bucket filled with water, they can perform PourAction which takes in the specific fire used in this interaction along with the fire’s location. PourAction then puts out the specified fire, converting it back to the previous ground while emptying out the bucket.