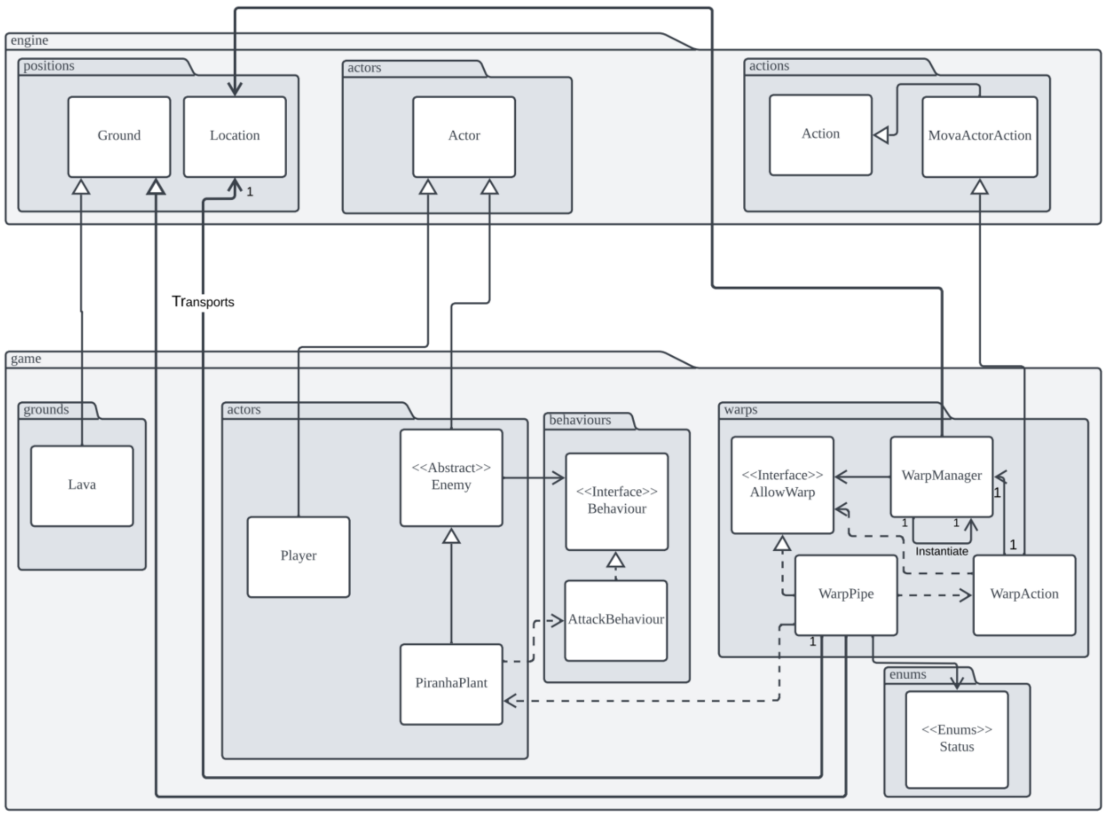
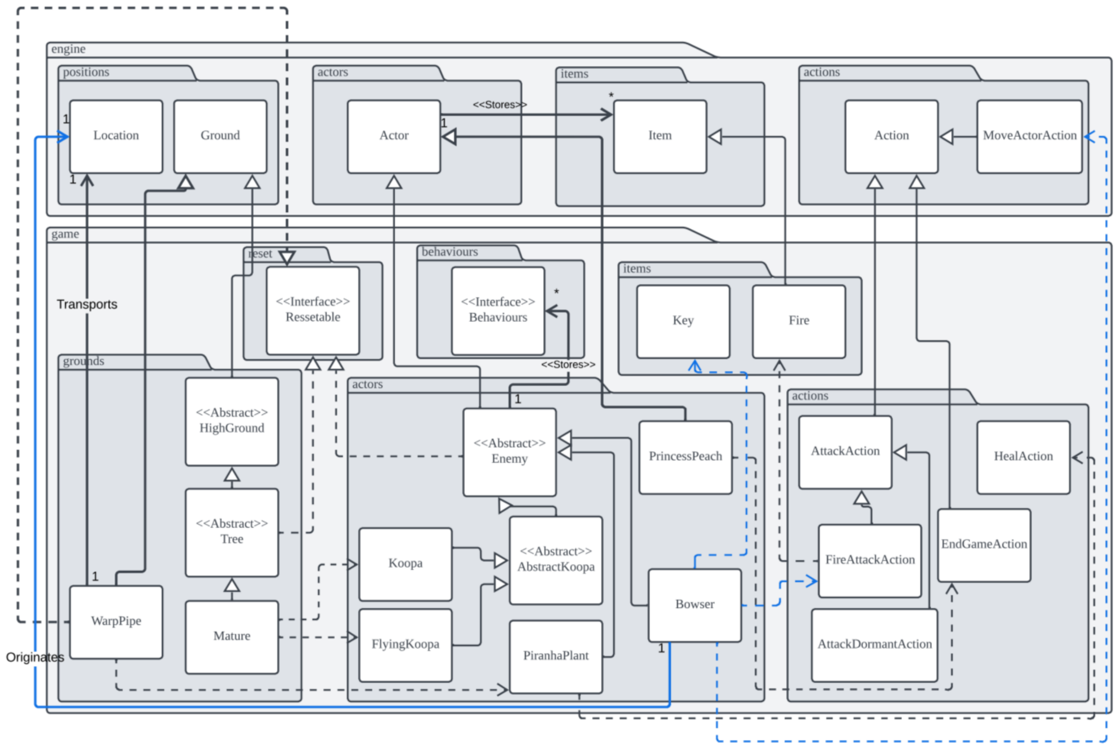
Design Rationale

Class Diagrams

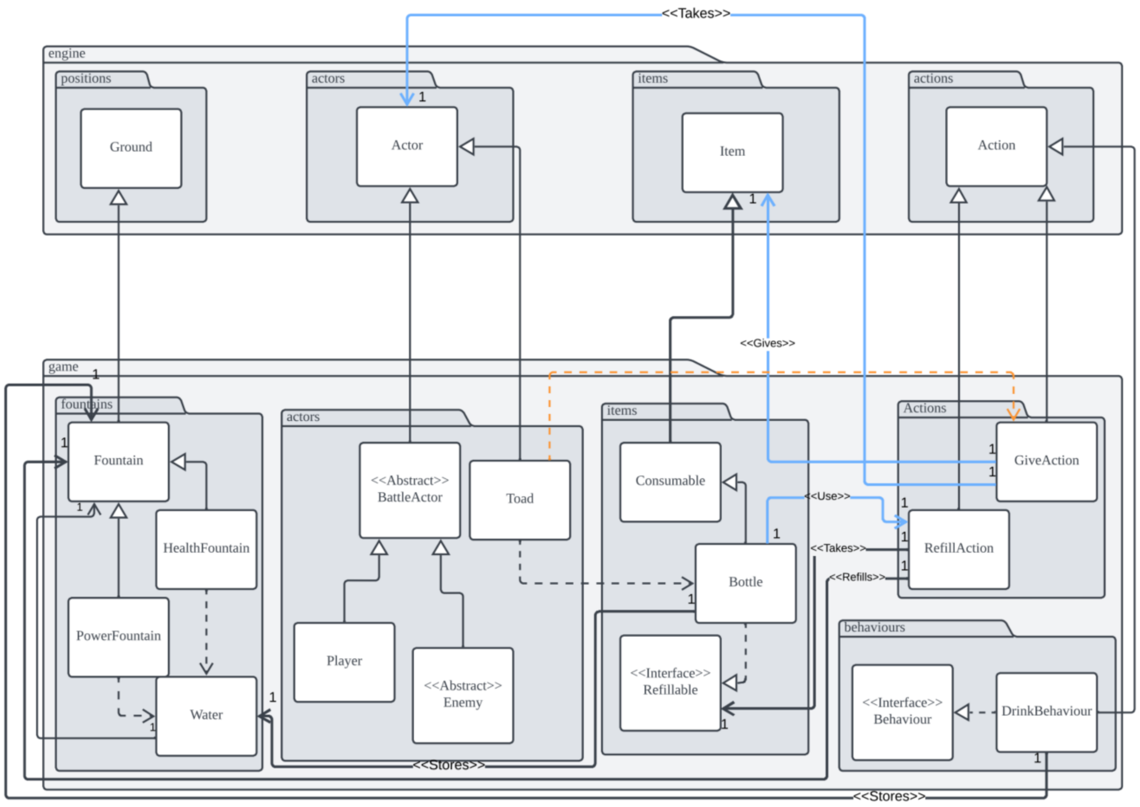
Task 1



Task 2

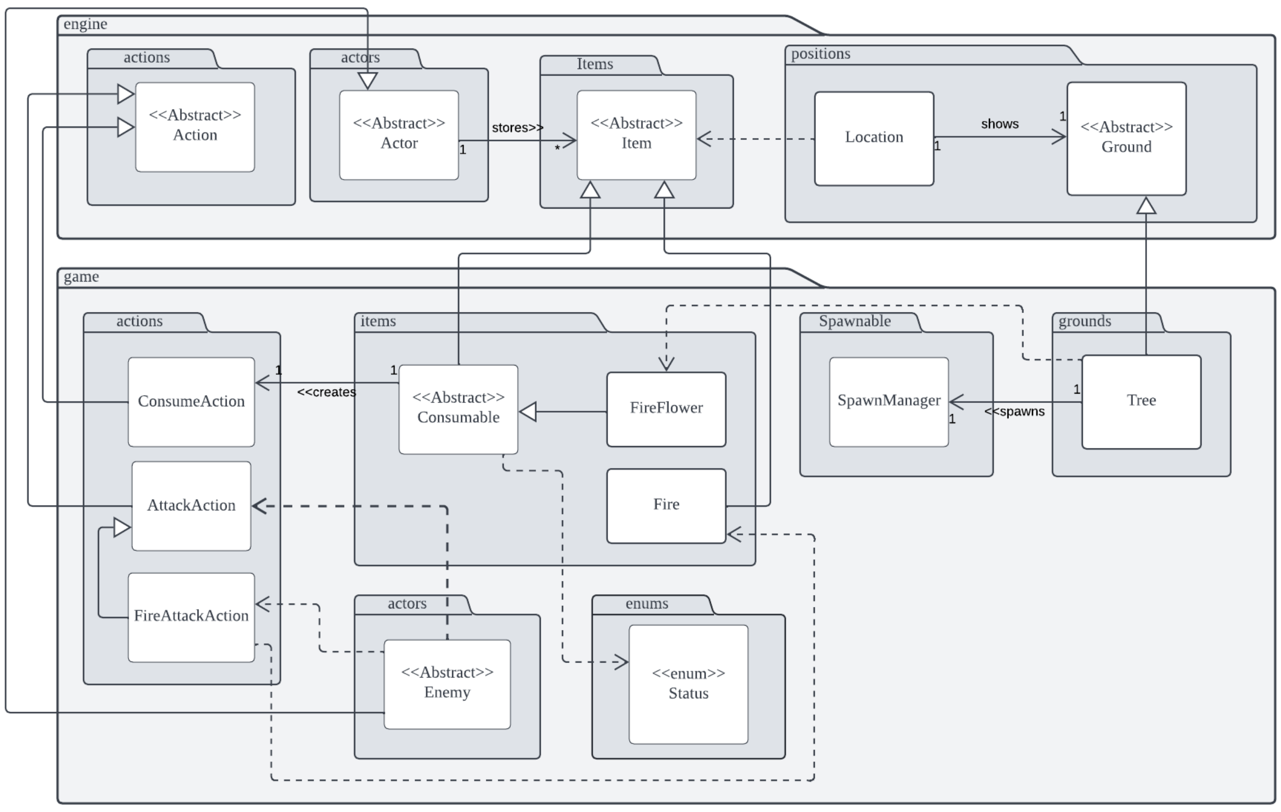


Task 3



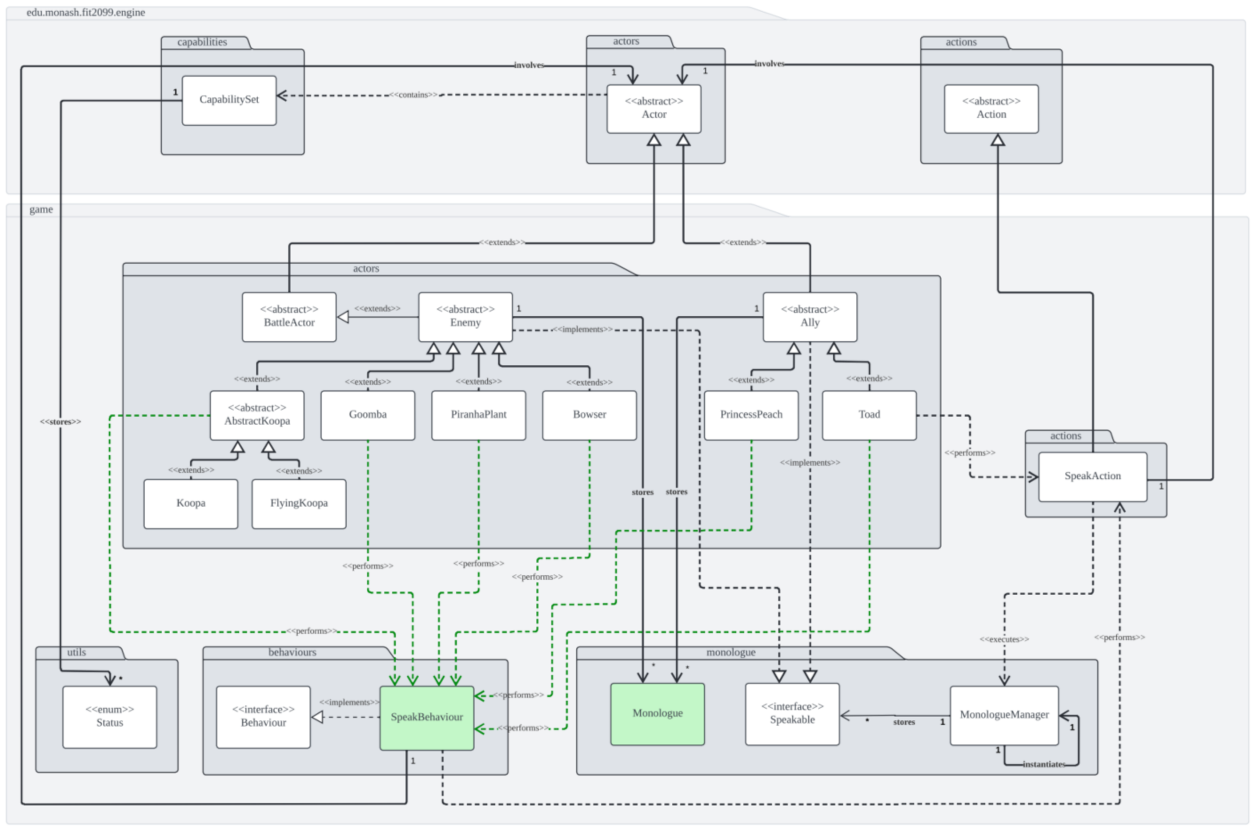
<<Takes>>
engine
positions
Ground
actors
Actor
items
Item
actions
Action
<<Gives>>
game
fountains
Fountain
HealthFountain
PowerFountain
Water
actors
<<Abstract>> BattleActor
Toad
Player
<<Abstract>> Enemy
items
Consumable
Bottle
<<Interface>> Refillable
Actions
GiveAction
RefillAction
<<Use>>
<<Takes>>
<<Refills>>
behaviours
<<Interface>> Behaviour
DrinkBehaviour
<<Stores>>
<<Stores>>

Task 4



engine
actions
<<Abstract>> Action
actors
<<Abstract>> Actor
stores>>
Items
<<Abstract>> Item
positions
Location
shows
<<Abstract>> Ground
game
actions
ConsumeAction
AttackAction
FireAttackAction
<<creates
items
<<Abstract>> Consumable
FireFlower
Fire
Spawnable
SpawnManager
<<spawns
grounds
Tree
actors
<<Abstract>> Enemy
enums
<<enum>> Status

Task 5



General

Task 1

Inside the new map, we have a new ground type, which is lava. Therefore, we created a new class named Lava inherited from Ground. This class is responsible for behaviour of the Lava ground, i.e. inflicts damage to the player standing on it, no enemy can enter the ground, etc. Furthermore, to simplify the creation of the maps, we introduce the text file containing the maps as the input instead of hard-coded the map in a list of strings. This approach makes the code cleaner and improves the readability. Other than that, it helps to improve the maintainability since we only need to edit a specified text file for the map modification, if any. On the other hand, we thought of implementing the map class for each specific map since each map would have different behaviours. However, that would be over-engineered the tasks since we would not need the map with in-depth details inside this last assignment. Note that warp pipe can be either a ground or an item based on the engine code. However, we decided to make it a pipe since we want to make sure that no enemies other than Piranha Plant can be on the pipe. Other than that, we decided to have the player 'jump' onto the pipe instead of moving to the pipe. This is only achievable using the engine code with minimal effort by having the pipe as a ground instead of an item.

Bear in mind that we have a warp manager to manage the warping action. This is because every warping would have to access both the original pipe and destination pipe and their specific location. Without the warp manager, we would have extra dependencies and it is hard to extend new features in the future where we have multiple maps requiring multiple warp options. Hence, with the warp manager, these features could be extended easily.

Task 2
This task is mainly on creating new enemies and npcs. The new enemies are Flying Koopa, Piranha Plant and Bowser while the new npc is Princess Peach. Each of them is implemented as a new class since they have their own specific behaviours. We note that flying koopa has similar characteristics with normal koopa except the flying ability. Therefore, in order to reduce repetition of code and follow LSP, we create a new abstract class for koopas called AbstractKoopa that generalises the common behaviours of the Koopa while the children have different abilities. Note that Bowser has quite a few dependencies based on the given specifications. This can be treated as a boss enemy. A better way to extend this is to create a BossEnemy class which can be extended easily in the future for boss type enemies. However, in this assignment, we do not need that since this is the last assignment which requires no further modification and hence the changes would be over-engineering. Other than that, we have a timer for fire which allows this to be extended easily for different kinds of fireAttackAction. HealAction is related to the ResetAction for specific enemies. HealAction is initially designed to be used in Requirement 3 as well. However, requirement 3 would have a better design without using HealAction and hence HealAction is tightly coupled with Actor in this design which is fair since only actors can be healed. EndGameAction in this design is highly coupled with Princess Peach since the message is designed to end with saving Princess Peach. This is fair since we only have 1 ending plot in the game.

Task 3
To accommodate the feature where fountains can increase the basic damage of the actor, we create a new class called BattleActor (for actors involved in battle) which contains the damage stat. Initially, the design requires another class to store the stats of the battle actors. However, since we only have 1 stat, i.e. the damage to be stored, it is considerably fine to use primitive data type to store the damage value. Furthermore, we note that there is no specific behaviours related to the battle stats other than changing the values, creating a new class would mainly be an information container which is bad in oop design. As such, we prefer the above approach. However, in order to maintain extensibility and maintainability, it is suggested to have another class for handling these battle stats provided that this is not over-engineered since the stats may not be used in the future for this being the last assignment. The above design would have a fountain inherited from the ground which would produce water with different effects. Since the effect of the water comes from the fountain, we do not need to create different classes for different water from different fountains. This would increase numbers of small classes whose existence is purely serving the fountain effect. This would increase the memory usage and at the same time increase the number of dependencies beside the number of classes, resulting in low maintainability. Hence, the above design is preferred where the fountain would provide the effect to the water. The water class can handle all the effects from different fountains.

From the above design, we also include the extensibility of the bottle by implementing the interface refillable. This allows that any items in the future that can be refilled can be extended easily from the interface. Note that the refill action is tight with the refillable since any refillable supposedly uses the refill action to refill. We can also generalise the refill action by having an abstract ground class that can generate different fluids for collection and a fluid class to represent such fluid. Then fountain and water can extend from the abstract class and fluid class respectively. This would reduce the coupling between water and fountain via abstraction. However, this would be over-engineering since we would never need that for this last assignment. However, in large-scale projects, the design is preferable than the current implementation for maintainability and extensibility. Note that the implementation of giveAction is general, i.e. every single item can be given from one actor to another actor as long as the action is performed on the item. We only use it, in this context, to hand the bottle over to the player. Trade action cannot be used here since the item does not cost any money and the item is limited, i.e. only 1 in a game. Hence, using Trade action would be inappropriate since it violates the original responsibility of the trade action. As such, giveAction is created to preserve the SRP principle.

Task 4
For this task, the Fire Attack effect would only last for 20 turns which is similar to the INVINCIBLE effect from Assignment 2. Hence Power star and the new Fire Flower Consumable is implemented from their own respective classes to last a certain amount of turns. This is done using the tick method. The count out be declared in the Fire Flower class. Once an actor consumes Fire Flower, the count would be declared as 0. After each turn, the tick method would check if the actor has FIRE_FLOWER capability. If actors have it, the count would be increased by 1. Once the count is more than 10, the capability would be removed and Fire Flower Consumable would be removed from inventory. To ensure the use of the tick method after ConsumeAction is called, Consumable would be added to inventory and the Consumable’s ConsumeAction would be removed. Only after the count reaches the amount of turn stated, the item would be removed from inventory. However, this would not be implemented this way if the tick method is not required. Items would be immediately removed at the ConsumeAction if Tick method is not required. FireAttackAction would also be used by bowser, hence in execute of FireAttackAction, it will check if actor is Bowser, if actor is bowser, super.execute will be called to attack with FireAttackAction and AttackAction at the same time.

Task 5
For task 5, each listed character (Princess Peach, Toad, Bowser, Goombas, All Koopas, and Piranhas) are required to talk or speak every “even” or alternating turn. The characters are given a few monologues in which they are able to randomly choose from in order to display or speak in every turn. To implement this functionality, two new classes are created from the original Monologue function done in requirement 6 of Assignment 2. These new classes include Monologue class, which will store a monologue that a speakable actor can perform, and also the conditions in which the monologue can be spoken. Besides that, a SpeakBehaviour class is also created in order to perform a SpeakAction for each speakable character every alternating turn. Lastly, an abstract class named Ally which inherits from Actor class is also created in order to accommodate allies such as Princess Peach and Toad to be able to inherit the behaviour functionality.

Inheritance and Polymorphism and Single Responsibility Principle
Note that new classes are created for every single new entity which has their own behaviours and responsibilities. As such, we follow the SRP principle. Other than that, every new class is extended from the existing classes, if applicable in order to make use of inheritance and polymorphism for maintainability and extensibility of the system. For example, we create WarpPipe inherited from Ground since WarpPipe is technically a ground type that has a specialty that allows players to travel between different maps according to the design. Hence, we can generalise the behaviours using Ground and have specific behaviours implemented in WarpPipe. This would follow SRP and at the same time make use of DRY principle.

Note that the warp manager is a singleton. However, it does not violate SRP since it only stores the information of the allowWarp objects and their location and allows the client to access the information whenever required. It has only 1 responsibility to change in this context. However, as the system grows in the future, it may violate SRP when it is used to handle more information between the allowWarp objects and their respective behaviours. Other than that, it also exhibits the characteristics of connascence of identity. Nevertheless, in this context, it is fair to implement the singleton since the pipes stay forever and never deleted or added once the maps are initialised. This means that the manager would always store the same list of pipes, only with their location changing due to certain actions. Other than that, SRP is preserved since the class has only single responsibility as per this implementation.

Similarly, in task 2 and 3, we make use of the polymorphism concept and SRP to create new classes. For example, we have Bowser inherited from Enemy and Refillable is implemented for items that can be refilled.

In task 5, the new Monologue class also abides by the SRP as it is only responsible for handling one monologue and all its conditions that come with it. The SpeakBehaviour class also has one responsibility which is to ensure that the speakable actor which has this behaviour only speaks in alternating turns, this makes it easier in the future to modify this functionality in the future easily (e.g. make the speakable actor speak every 3 or 5 turns). Lastly, the new abstract Ally class inherits from Actor class adding the speak functionality so that it does not have to be implemented by its child classes individually, instead the super() function can be called to accommodate for this function.

Open-Closed Principle
This principle is also preserved since we make use of inheritance whenever possible. Furthermore, we notice that every single extension does not require modification from the parent class. For example, if we have a new fountain type, we only need to create a new class that inherits the fountain class. Other than that, if we have a new enemy, we also need to extend a new class from the enemy. Furthermore, when we decide to change the inheritance of the enemy, we only change it based on the enemy class and the children class are not affected. This is the usefulness of this principle and inheritance concept. Note that the use of BattleActor is a better design compared to the previous assignment 2 design since it can now handle the actor involved in battle well with different buff and debuff. A better design, as suggested above, is to use a class to store the stats and handle the buff and debuff. However, that would be over-engineering in this case since the battle system is

simple, i.e. attack and hitpoint are used. Therefore, current design is enough to abide by the open-closed Principle. As stated in Assignment 2, different types of attack could be extended from the attackAction. A good example here is the fireAttackAction. This shows that open closed principle is indeed followed in the above design, including previous design in Assignment 2. All the classes/interfaces like fountain, refillable, battleActor, allowWarp, abstractKoopa are designed to follow this principle. For example, note that if we have a new type of Koopa which can swim instead of fly, we only need to have a new class extending from abstractKoopa. This ensures the extensibility and maintainability of the system. Same goes for Task 5, if we have a new type of character which assists the player, the character can directly inherit from Ally.

Liskov Substitution Principle
Note that following the above design, all the subclasses can replace their parent classes for the operation requesting their parent classes. To avoid violation of this principle, abstractKoopa is created for extending flyingKoopa. However, certain operations in the system still violate this principle. This principle is maintained because it allows polymorphism to be carried out easily. For example, every single action from the engine codes require actor, location, ground and map as the arguments instead of their children classes. However, those children classes can successfully be integrated into the methods since they follow LSP. However, this also creates limitations based on the engine code besides those explained in Assignment 2. For example, refillAction requires arguments being passed as the child classes, i.e. Fountain and Refillable instead of Ground and Item. This creates the problem where following LSP would result in unusable refillAction directly from the implementation of engine code. This has to do with the connascence of type (the arguments should have the same type as the parameters). This arises in the implementation due to the specialty of the children classes being more important and valuable in certain actions, i.e. refillAction in this context. The solution proposed in the above implementation is to use downcasting based on the capabilities of the ground class (Note that using capabilities would resolve the issue of downcasting where not all the ground classes are fountain. This avoids the use of instanceof which does not follow LSP and polymorphism). This is a better approach than using a singleton manager to handle the refillAction. This is because refillAction is unlike warp action which has a defined destination. refillAction is concerned about refillable, fountain and fluid. Hence, a manager that is designed to handle the refillAction would violate SRP. As such, the current design is better in terms of maintainability and extensibility.

Interface Segregation Principle
All the interfaces implemented above are small and have only 1 responsibility. For example, allowWarp is used to handle those objects that allow the player to perform warp action. The methods implemented are to change the destination and check whether the scenario allows the changes (destination change). This indeed is a small interface that handles only 1 responsibility. This maintains the extensibility of the system by allowing multiple inheritance using small interfaces. Other than that, the classes are easily maintained via interface and abstract classes.

Dependency Inversion Principle
Note that singleton manager and interfaces are designed to follow this principle. For example, we can view the singleton manager, warp manager as the client and all the objects

that provide warp action as the server codes. To reduce the dependency, allowWarp interface is created as an abstraction that channel manager to the objects that allow warp action which in this case is warp pipe. This ensures the extensibility with similar reasons as the behaviours and enemies. Note that no matter how many objects are created for having capability to do the warp action, the manager does not need to bother about the new classes created since it only needs to make use of the abstraction, allowWarp in this case to achieve its responsibility. On the other hand, those objects also do not need to concern how the high level objects could use them since they only need to follow the contract specified in the interface. Hence, abstraction plays a vital role here to increase the maintainability of the system.