# Work Breakdown Agreement (WBA), Group 1 Lab 16.

Group Members:
1. Quan Bi Pay
2. Khoo Ming Jian
3. Vihaan Philip

We hereby agree to work on FIT2099 Assignment 1 according to the following breakdown:

1. **Pay Quan Bi** will be responsible for completing the class diagram and design rationale for:
    a. Requirement 1: Let it grow
    b. Requirement 2: Jump Up, Super Star
    c. Requirement 3: Enemies

    Completion Date: Friday, 8th April 2022

2. **Khoo Ming Jian** will be responsible for completing the class diagram and design rationale for:
    a. Requirement 4: Magical Items
    b. Requirement 5: Trading

    Completion Date: Friday, 8th April 2022

3. **Vihaan Philip** will be responsible for completing the class diagram and design rationale for:
    a. Requirement 6: Monologue
    b. Requirement 7: Reset Game

    Completion Date: Friday, 8th April 2022

** All three members have agreed to review the class diagrams and summarise the design rationale together.*

For sequence diagram each team member is responsible for creating one sequence diagram for an interaction:
1. Player has a conversation with Toad – **Vihaan Philip** is responsible
2. jumpAction.execute(actor, map) – **Pay Quan Bi** is responsible
3. TradeAction of player with Toad – **Khoo Ming Jian** is responsible

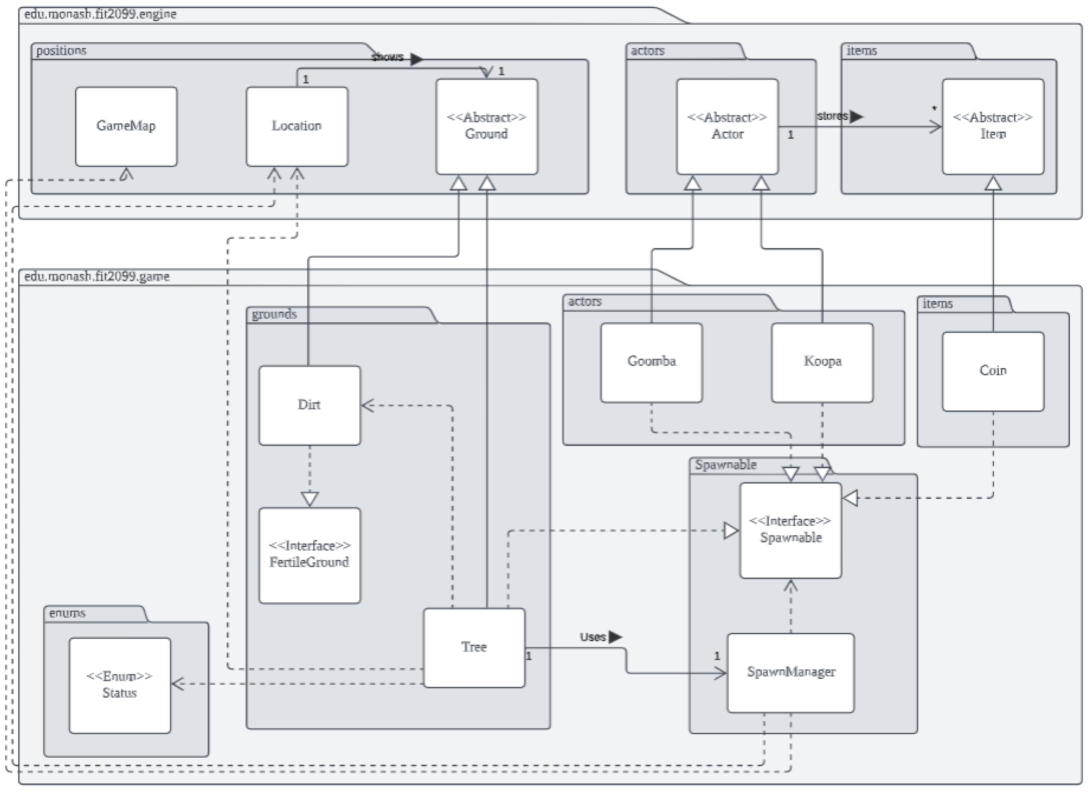
**Signed by (type "I accept this WBA"):**

I accept this WBA – Vihaan Philip
I accept this WBA – Pay Quan Bi
I accept this WBA – Khoo Ming Jian

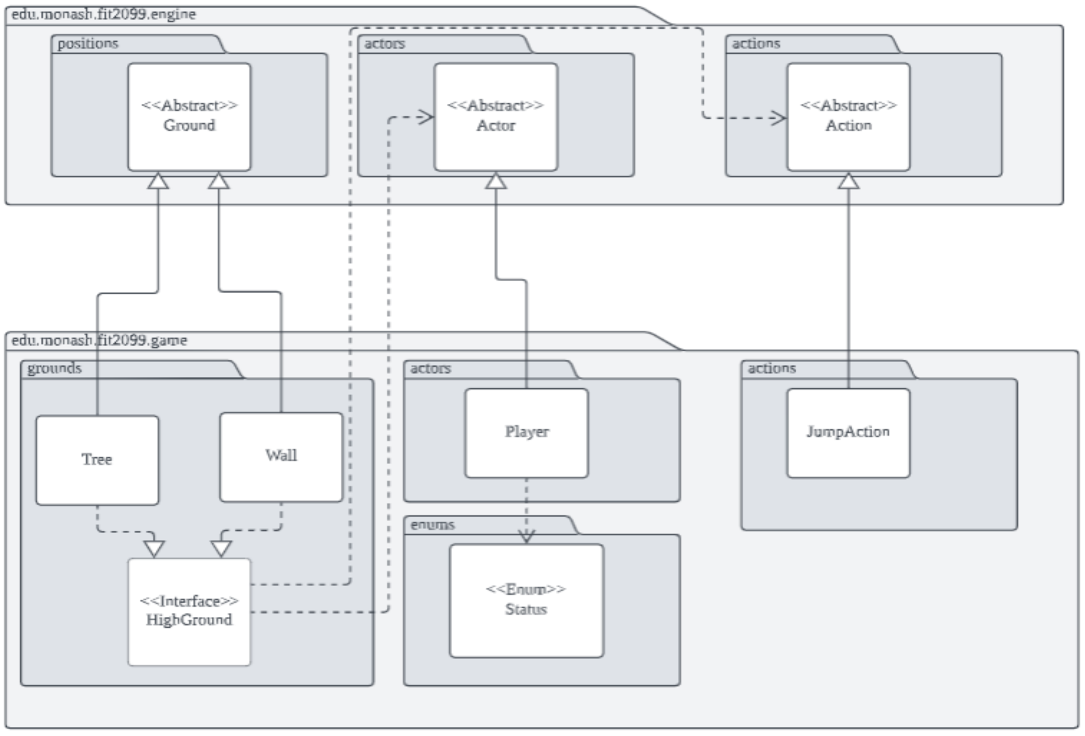
# FIT2099 Assignment 1: Design

## Class Diagrams:

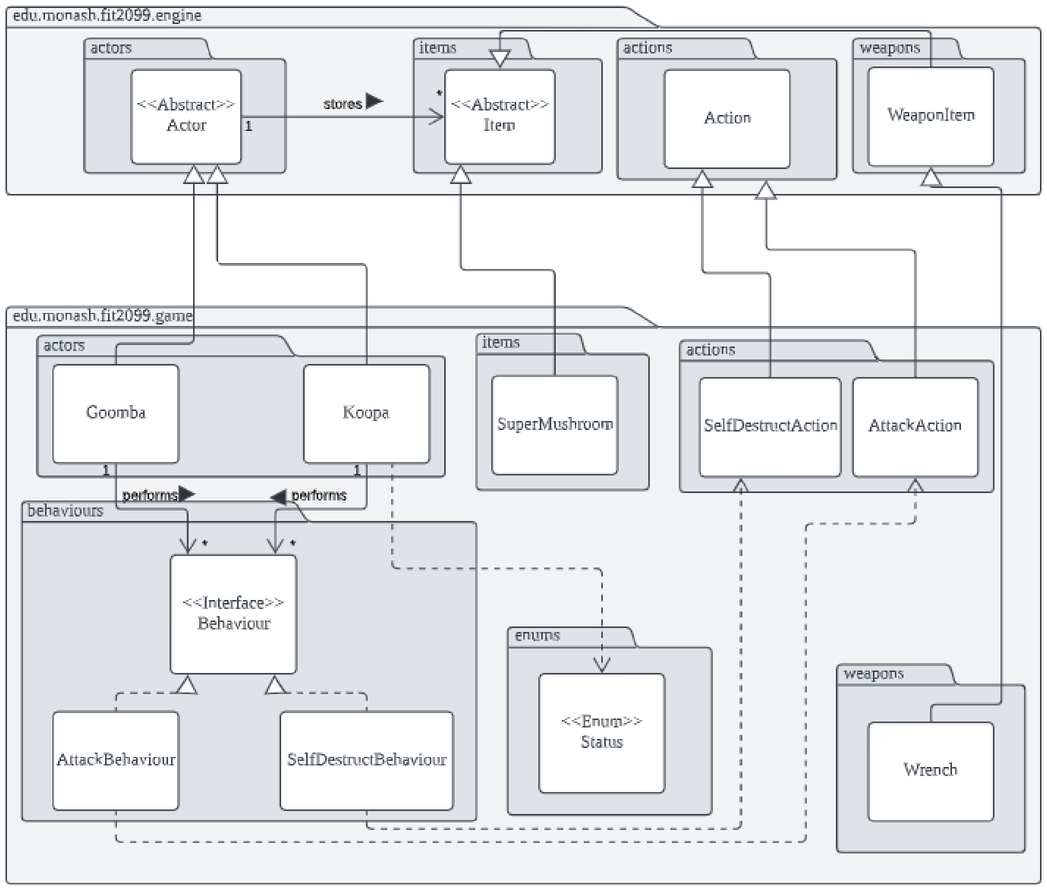
### Requirement 1



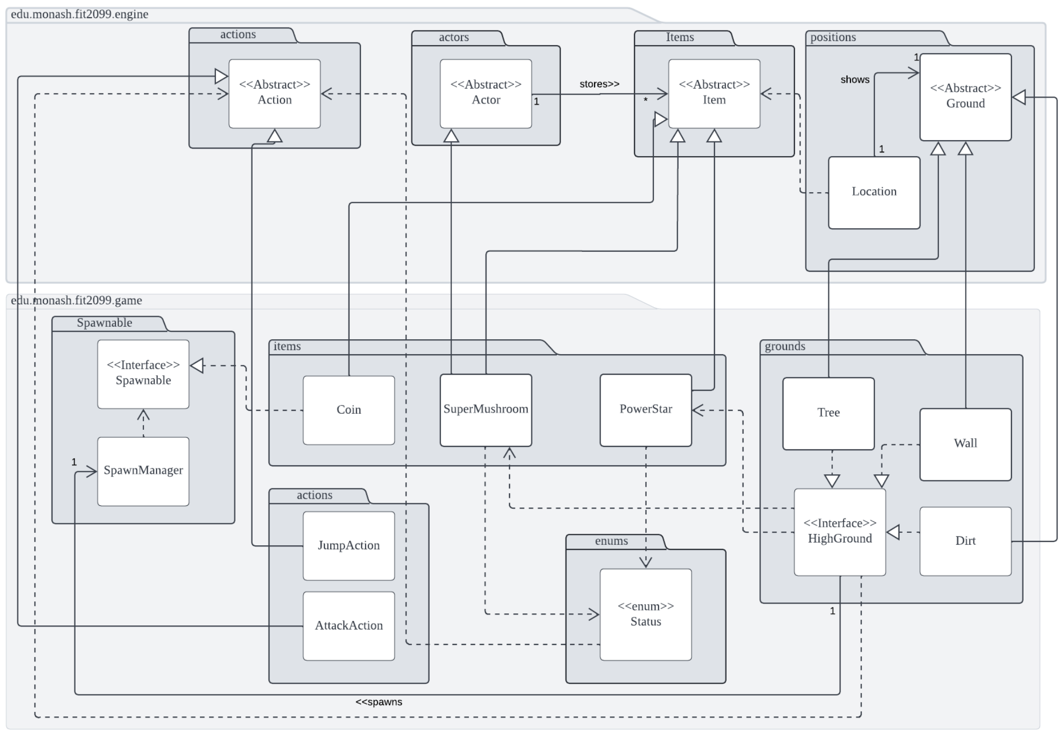
### Requirement 2



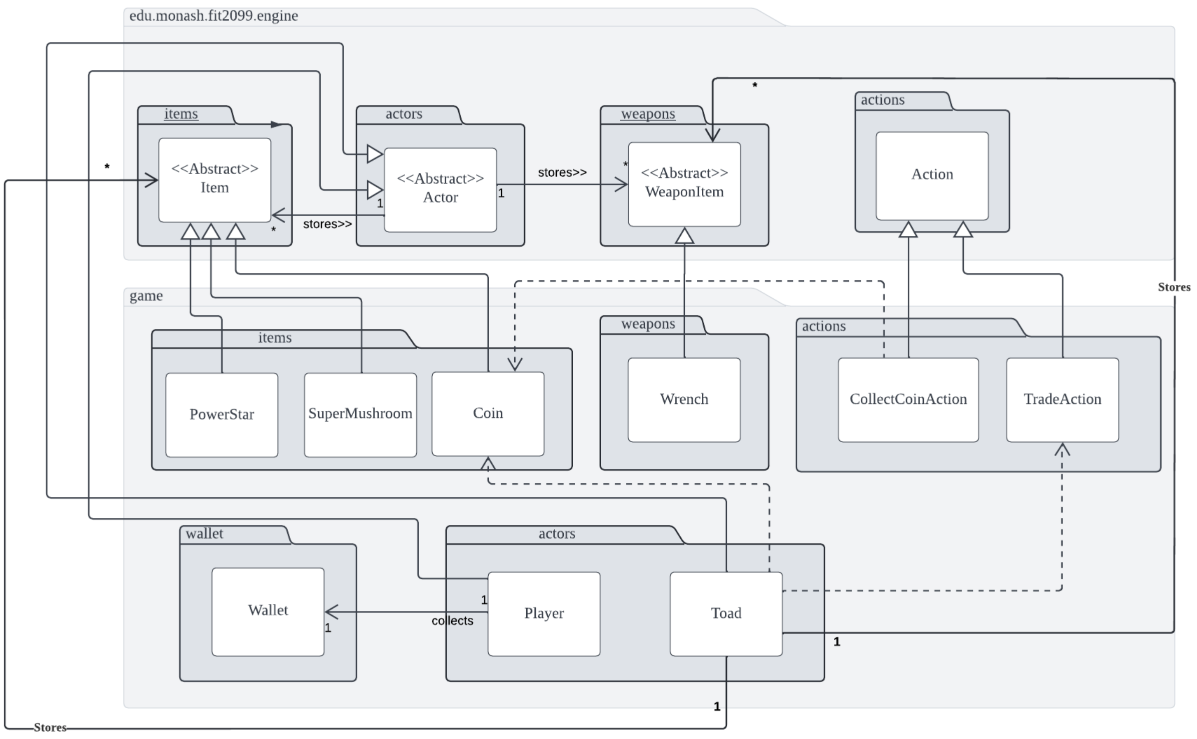
## Requirement 3



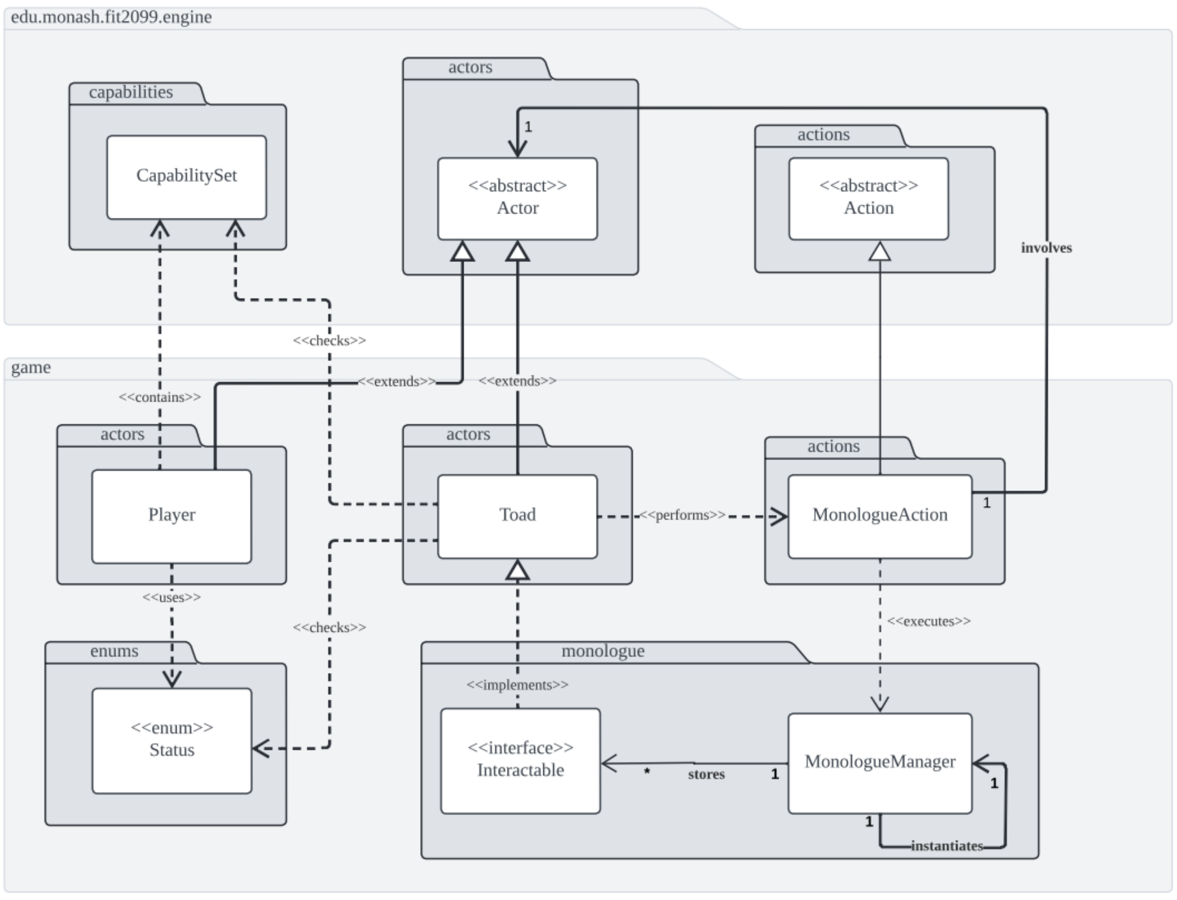
## Requirement 4



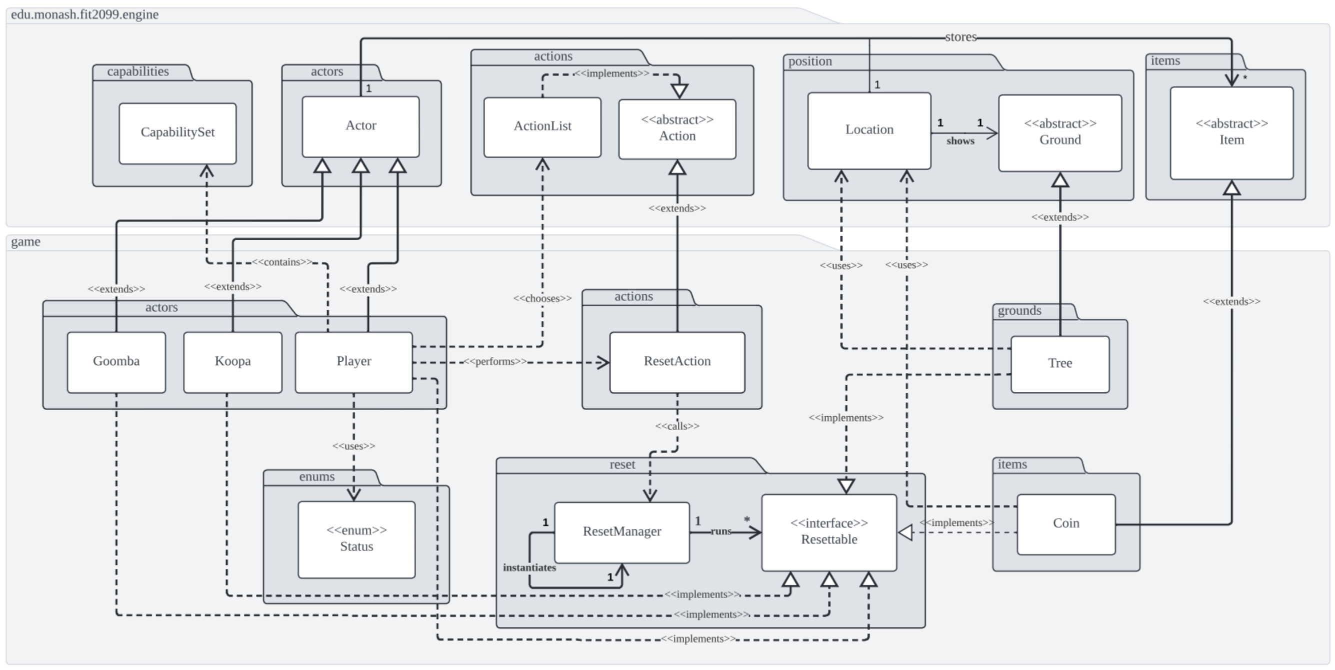
## Requirement 5



## Requirement 6



## Requirement 7



## Class Responsibilities:

**Status**
An Enum classes that contains all the buff/debuff status as well as the capabilities of the objects

**Dirt**
A ground subclass that is responsible for characteristics (attributes) and behaviours (methods) of the dirt (ground type)

**FertileGround**
An interface that is responsible for characteristics (attributes) and behaviours (methods) of the fertile ground (any ground can be used for planting)

**Tree**
A ground subclass that is responsible for characteristics (attributes) and behaviours (methods) of the tree (ground type)

**Goomba**
An actor subclass that is responsible for characteristics (attributes) and behaviours (methods) of the Goomba (a type of enemy)

**Koopa**
An actor subclass that is responsible for characteristics (attributes) and behaviours (methods) of the Goomba (a type of enemy)

**Spawnable**
An interface that is responsible for characterising the behaviours of a spawnable object. This is currently used for objects that can be spawned/grown from the tree

**SpawnManager**
A manager that is responsible for handling the spawnable objects, i.e. when to spawn and where to spawn the objects

**Coin**
An item subclass that is used to handle the characteristics (attributes) and behaviours (methods) of the coin.

**Wall**
A ground subclass that is responsible for characteristics (attributes) and behaviours (methods) of the wall (ground type)

**HighGround**
An interface that is responsible for characteristics (attributes) and behaviours (methods) of the high ground (can only be passed using jump action for example)

**Player**
An actor subclass that is responsible for characteristics (attributes) and behaviours (methods) of the player/mario.

**JumpAction**
An action subclass that is responsible for handling jump action, i.e. when can the actor jump and to where can the actor jump etc.

**Behaviour**
An interface that is used to create the actions in order to achieve certain objectives by the actor. For example, wanderingBehaviour would be used by the enemy to chase after the nearby player.

**AttackBehaviour**
A class implements Behaviour to perform the attack action based on certain objectives achievable by the actor. At current stage, it is used to attack the player automatically

**SelfDestructBehaviour**
A class implements Behaviour to perform the self-destruct action based on certain conditions specified by the actor.

**SelfDestructAction**
An action subclass that is responsible for handling the self-destruct action, i.e. which actor is conducting the action, what is the consequence of this action etc.

**Toad**
An non-playable characte in which the player can trade or interact with.

**ResetAction**
An action subclass which is responsible for executing the Reset Manager to reset the game. This can only be done once throughout the game.

**Interactable**
An interface responsible for determining which monologue to execute when interacting with the player.

**MonologueManager**
A manager which is responsible for accessing the monologue methods from classes which implement the Interactable class.

**MonologueAction**
An action subclass which is responsible for executing the Monologue Manager to return a monologue to the player.

## Design Rationale:

### 1) Making use of the Inheritance concept

In our implementation, we have decided to make use of the concept of inheritance. The concept of inheritance helps reduce the amount of code we need to write through the usage of abstract classes. For example, given a group of classes with similar functionalities, instead of writing similar code for each and every one of those classes, we instead just code one abstract class containing abstract methods which must be implemented by all the classes. Then, we create subclasses extending from the abstract class which overrides those abstract methods, allowing them to implement the functionality of those abstract methods in their own unique way. This design improves both maintainability and scalability of the program besides reducing repeated code.

Compare this design to another design where inheritance is not applied. For example in Requirement 1, we have decided to extend the Tree and Dirt classes from the abstract Ground class as these two classes have similar functionalities. If we do not apply the concept of inheritance we quickly realise that the DRY ( Do not repeat Yourself) principle is broken as we are forced to code the similar methods in Tree and Dirt classes respectively. Now, imagine if we have many classes which implement similar methods to the Tree and Dirt class, this would then raise issues on maintainability and scalability of the program.

Another advantage of using the above design is that we could make use of the dynamic polymorphism to adapt the common functionality to different subclasses while maintaining the Liskov Substitution Principle. With the polymorphism used, we can adapt dependency inversion principle since the parent class, Ground, is an abstraction used to link high-level (map and world) and low-level modules (subclasses like Tree, Dirt and so on).

Similar explanations can be applied to the relationship that make use of the inheritance concept. Note that inheritance would also follow SRP since every child class should have their own responsibility. For example, in the above design, Tree is responsible for Tree behaviour and its characteristics while Dirt is responsible for Dirt characteristics. Even though they share the same parent class, Ground, their responsibilities are different.

**2) Making use of Interfaces**

We have also decided to utilise interfaces in situations where we need a group of classes to share similar methods without implementing multiple inheritance to accommodate for this. This is to avoid situations in which encapsulation and polymorphism may fail due to multiple inheritance. This design also reduces the number of dependencies we may have on a particular parent class.

Imagine that we have another design such that we do not need the interface. For example in Requirement 1, we decided that Dirt and FertileGround should have an “implement” relationship. Another design would be to only add the capability of FertileGround to all the fertile ground classes, i.e. Dirt in this context. Then, a class behaviour depending on the fertile ground would have numerous dependencies based on the number of fertile ground classes created. Other than that, we may need extensive use of literals to check whether the class is actually a fertile ground which is a bad design.

Instead of using an interface, we can make FertileGround as another abstract class inherited from Ground. However, this would raise the multiple level inheritance and multiple inheritance problems. For example, if we want the FertileGround to be HighGround at the same time, we cannot make it since inheritance allows the child to have a single parent. Another possibility is that the function from Ground is overridden in FertileGround and inherited by Dirt. This may break the Liskov Substitution Principle and encapsulation (protected attributes) if extra care is not taken.

Using the above issue would allow maintainability and scalability since polymorphism is in use. Abstraction and Encapsulation would not be a problem without multiple level inheritance. However, we also have to make sure that the interface and abstract class used is small (contains only 1 responsibility) and easy to understand. This is because encapsulating every responsibility inside an abstract class or interface would make the program difficult to maintain and scale. For example, some classes may not need some of the responsibilities inside the abstract class and it would mess up the behaviour of the program if extra care is not taken. A better approach is to follow Interface Segregation principle as indicated in the above design where HighGround and FertileGround are 2 different interfaces instead of combining them into 1.

**3) Creating managers to manage multiple implementations of Interfaces.**

We have also decided to create Manager classes which allows us to manage multiple implementations of an Interface. The Manager stores objects of classes which implement methods from the particular Interface, and it handles accessing of these methods as well. This design is implemented to help reduce dependencies and to also not break the Separation of Concerns principle.

An example of this design being implemented is in Requirement 1, where the Tree should spawn enemies, trees and coins according to the set of rules defined in the specifications. Hence, we design a SpawnManager class to handle all the spawning behaviour of trees. This is an association relationship since the Tree should keep the manager as attribute for spawning behaviour.

All the spawn objects should implement Spawnable and SpawnManager should interact with the Spawnable instead of the objects respectively. This is aligned with the dependency inversion principle.

Without the use of Spawnable interface, it is highly likely that the SpawnManager has dependencies with every single spawnable object and no abstractions as the linkage between the manager and the objects. This is bad since we have to add in a new dependency relationship when we have a new spawnable object. Changes have to be made to both Manager and new classes. This would increase difficulty to maintain and scale the programme.

Imagine we do not have the spawn manager, then all the spawning behaviour would have to be handled by the Tree itself. This would make the codes inside the Tree long and multiple related functions have to be created inside the Tree which eventually require refactoring in accordance with code smells. Besides, it may violate the SRP (single responsibility principle) such that trees have multiple reasons to be changed, i.e. new behaviour is added and new objects to be spawned. To restrict the responsibilities to only behaviour, we have the spawn manager to handle the spawning behaviour for us, i.e. new objects to be spawned will not affect the Tree class but the SpawnManager.

Another example where this is used is in Requirement 6, where a Monologue Manager is created to handle monologues for Toad. An alternative approach is to let the Monologue Action handle the monologues for Toad. This is not a good approach as this breaks the Single Responsibility Principle (SRP) as one class is responsible for handling all the monologues for every Actor which implements it. Instead, the Interactable interface created allows any class which implements it to handle its own monologues. Then through the MonologueManager, we can access the methods from within classes which implement the Interactable interface. This design reduces a lot of dependencies and also makes the program much more scalable, as it is easier to add more objects which can Interact with the player without changing much of the existing code.

In Requirement 7, a Reset Manager is created to handle the resetting of the game. The alternative of this is to iterate through the whole GameMap to remove the Coin and convert the Trees to Dirt when Reset Action is called. This can be extremely inefficient especially if the game map is huge and many checks have to be made. Thus, the use of Reset Manager saves the hassle of iterating through the whole GameMap, by storing a list of Objects which are Resetabble, through implementing the Resetabble interface. In this way, each Object is responsible for the deletion/transformation of themselves, instead of relying on the Game Map class to do so.

Furthermore, the above design follows the abstraction since the Tree has no idea on how the spawning is done but it knows the manager can spawn the required objects. The hiding complexity allows the debugging process to be easier.

**4) Making use of Enums**

We have also made use of Enums in our design to easily keep track of the Status in which certain classes might have. Use of Enums could solve potential problems as explained below. However, maintaining a long list of Enums is also difficult when we have numerous status. Therefore, it is wise to separate the status into different Enum classes with different responsibilities to minimise the overlapping as well ease the use of the Enum classes.

According to the specification In Requirement 1, Tree has 3 stages which have respective behaviour. In order to keep track of the stages, we make use of Status (an Enum) to reduce the use of magic literals.

In Requirement 6, we have to check whether the player has certain capabilities in order to decide whether a monologue can be displayed or not. In our implementation we have decided to use Status (an enum class) to check whether a player has certain capabilities.

In Requirement 7,  at the start of the game, the player is given an Enum  status to be able to use the Reset Action function. This status is then removed from the player after the Reset Action has been performed to only allow the player to use it once.

An alternative approach would be to use magic literals like strings or numbers, various problems would arise. For example, what is the meaning of the strings/numbers? It is hard to maintain since other programmers may not understand the meaning without explicit explanation. Furthermore, checking conditions against literals is difficult since literals can take many forms. A simple example is lowerCase and capitalCase. Should we treat them differently or the same. This may cause the program to behave differently and unexpectedly without any compile-error. Lastly, it breaks the principle of not using too many literals when we have multiple stages.

In conclusion, the above design is the optimal solution to solve the above mentioned problems as well as keeping the program maintainable.

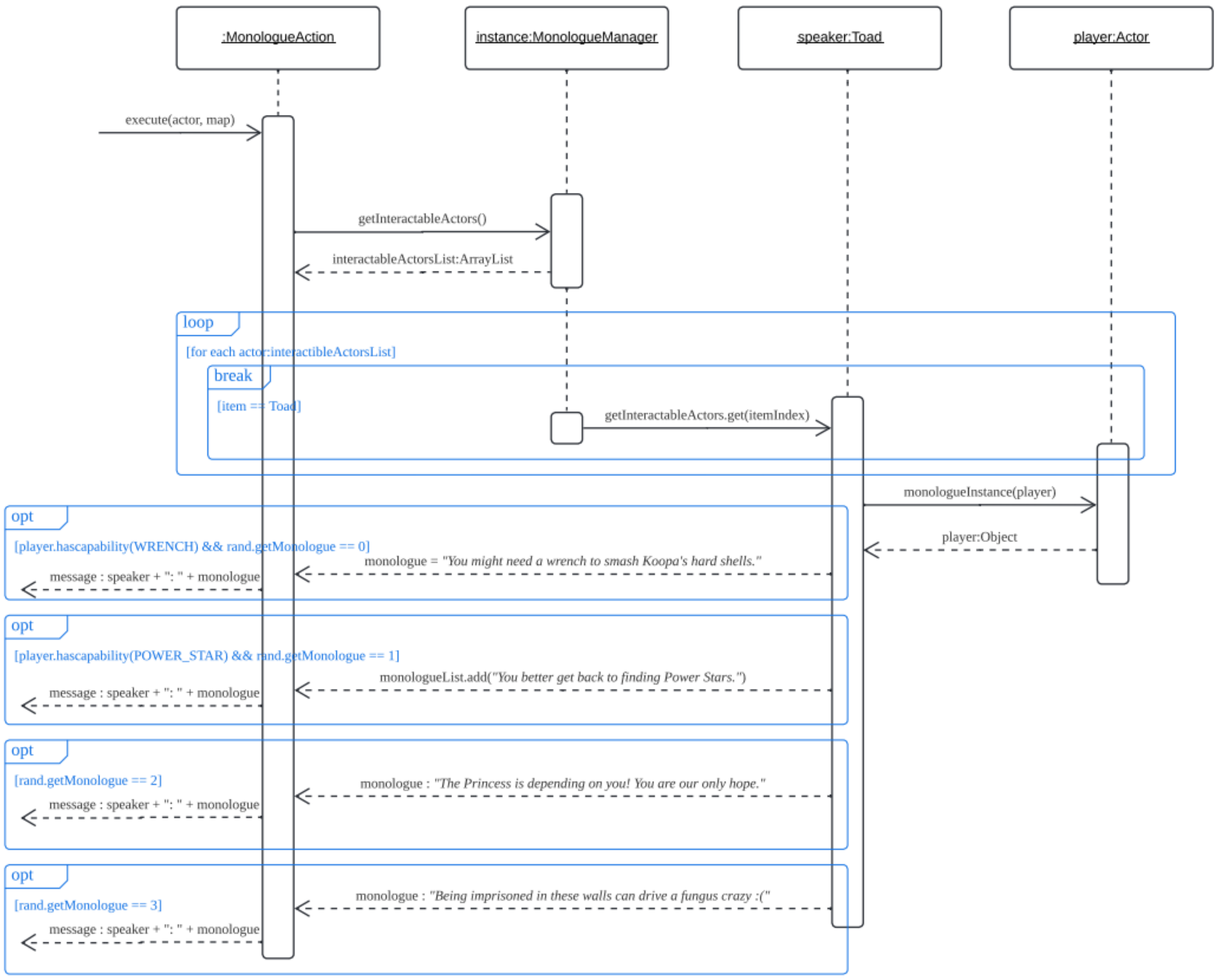
**5) Make use of Open-Closed Principle**

Note that open-closed principle is maintained in the association relationship in Actor/Item and Ground/Location. Note that no dependency relationship exists between the Actor and subclasses from Item. This is because Item is closed for modification and the Actor can complete the action on Item based solely on the Item common behaviour. This would help in maintaining the programme and improve scalability without much changes made. Other than that, every class implementing inheritance or interface also abides by the open-closed principle to allow maintainability and scalability.

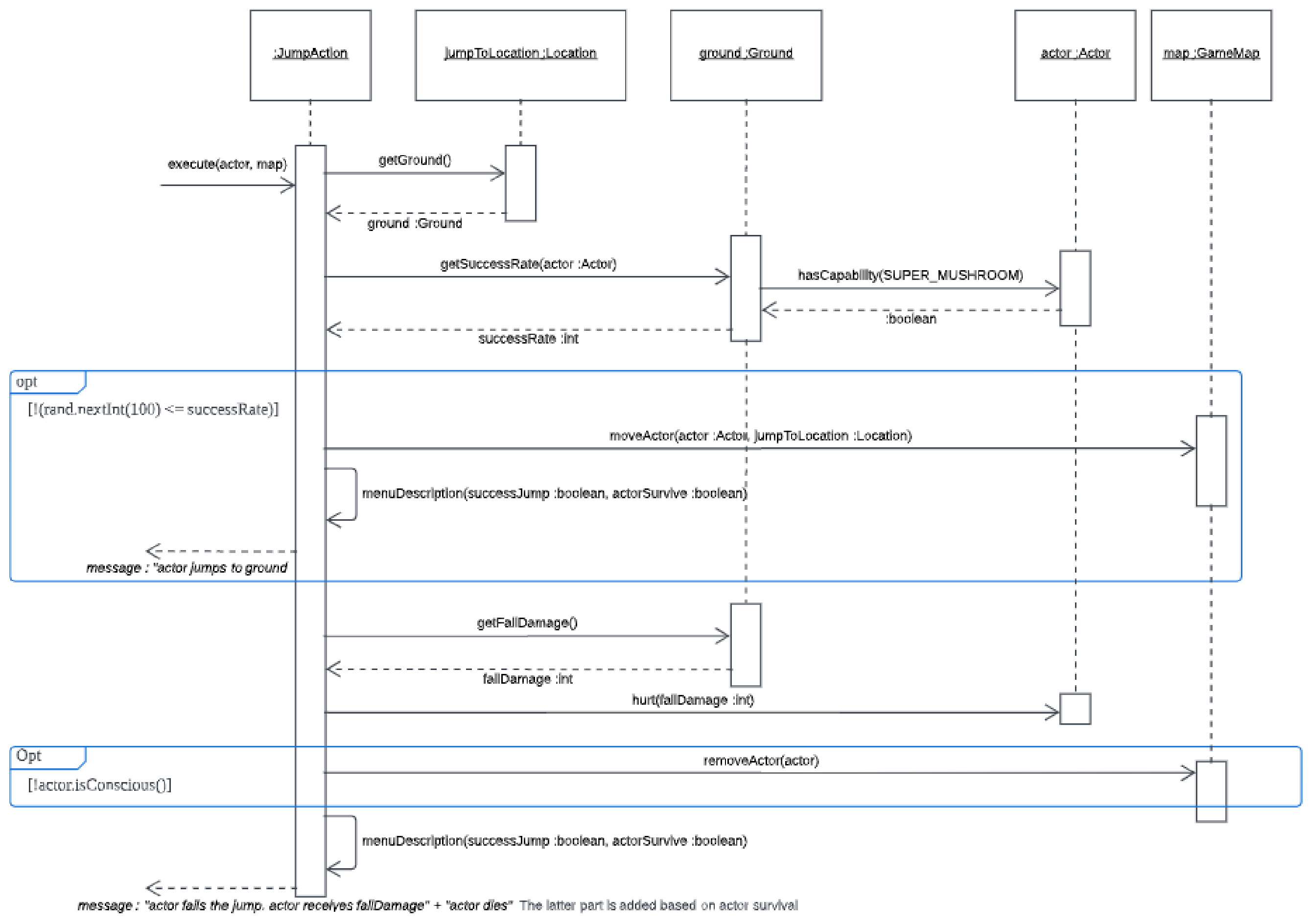
Imagine we have a Spawnable interface that does not maintain open-closed principle, then when we want to add a new class that implements the interface, we have to modify the existing interface which is prone to bugs and errors. Then, we have to modify the SpawnManager class as well to suit the new changes of the interface. This would, at the same time, break SRP and LSP. Hence, we ensure that every class implementing the relevant interface or inheriting from the relevant abstract class, open-closed principle does not break. As shown in the above design, the Spawnable and other classes implementing it do not have such problems.

# Sequence Diagrams:

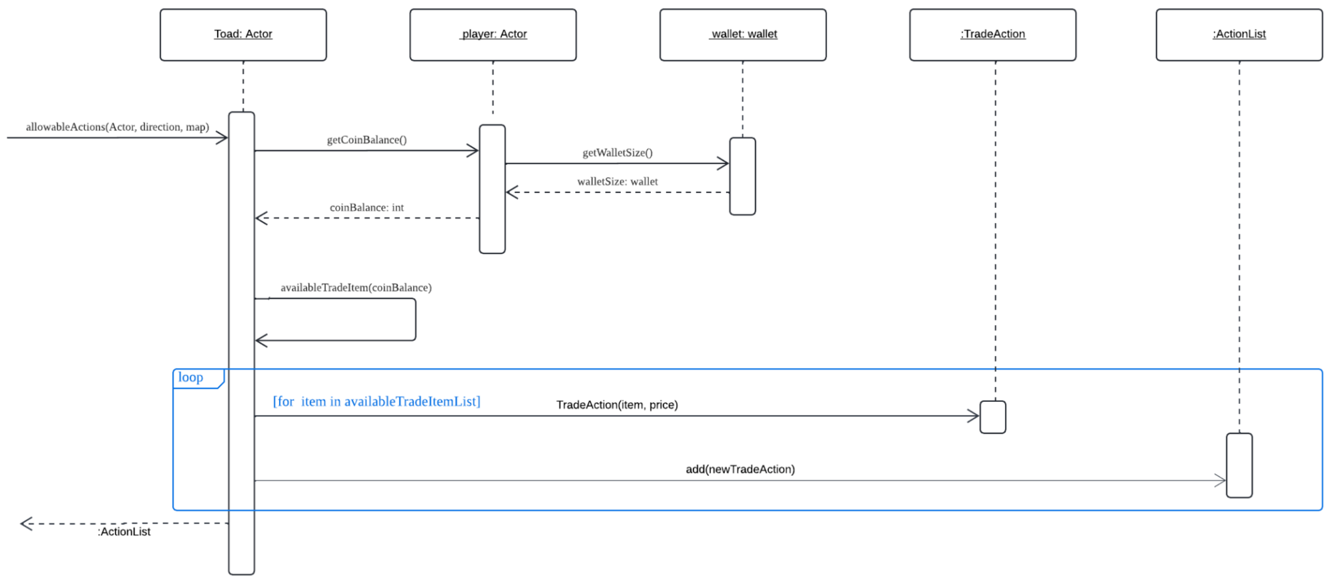
## Monologue Action (Interaction with Toad)


:MonologueAction
instance:MonologueManager
speaker:Toad
player:Actor
execute(actor, map)
getInteractableActors()
interactableActorsList:ArrayList
loop
[for each actor:interactibleActorsList]
break
[item == Toad]
getInteractableActors.get(itemIndex)
monologueInstance(player)
player:Object
opt
[player.hascapability(WRENCH) && rand.getMonologue == 0]
monologue = "You might need a wrench to smash Koopa's hard shells."
message : speaker + ": " + monologue
opt
[player.hascapability(POWER_STAR) && rand.getMonologue == 1]
monologueList.add("You better get back to finding Power Stars.")
message : speaker + ": " + monologue
opt
[rand.getMonologue == 2]
monologue : "The Princess is depending on you! You are our only hope."
message : speaker + ": " + monologue
opt
[rand.getMonologue == 3]
monologue : "Being imprisoned in these walls can drive a fungus crazy :("
message : speaker + ": " + monologue

## Jump Action (.execute() function)


:JumpAction
jumpToLocation :Location
ground :Ground
actor :Actor
map :GameMap
execute(actor, map)
getGround()
ground :Ground
getSuccessRate(actor :Actor)
hasCapability(SUPER_MUSHROOM)
:boolean
successRate :int
opt
[!(rand.nextInt(100) <= successRate)]
moveActor(actor :Actor, jumpToLocation :Location)
menuDescription(successJump :boolean, actorSurvive :boolean)
message : "actor jumps to ground
getFallDamage()
fallDamage :int
hurt(fallDamage :int)
Opt
[!actor.isConscious()]
removeActor(actor)
menuDescription(successJump :boolean, actorSurvive :boolean)
message : "actor falls the jump. actor receives fallDamage" + "actor dies" The latter part is added based on actor survival

## Trading sequence diagram


Toad: Actor
player: Actor
wallet: wallet
:TradeAction
:ActionList
allowableActions(Actor, direction, map)
getCoinBalance()
getWalletSize()
walletSize: wallet
coinBalance: int
availableTradeItem(coinBalance)
loop
[for item in availableTradeItemList]
TradeAction(item, price)
add(newTradeAction)
:ActionList