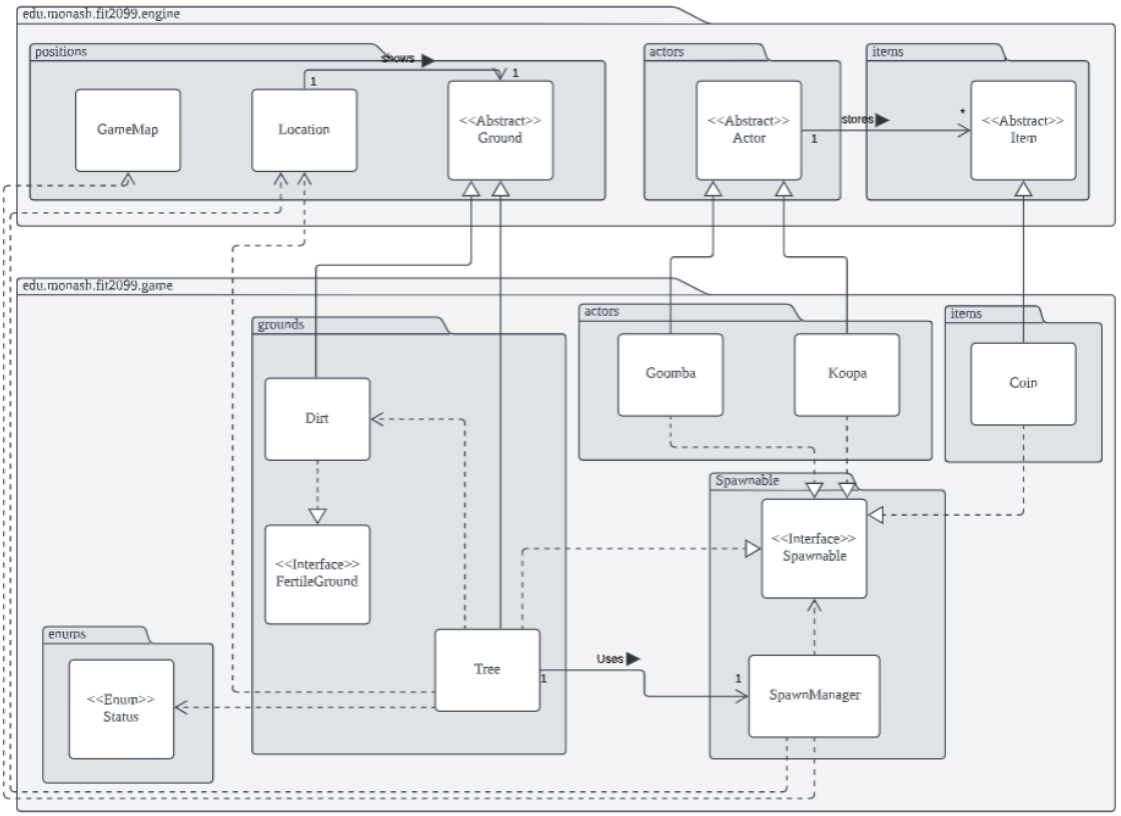
Design Rationale

Task 1



- Ground and Dirt / Ground and Tree / Goomba and Actor / Koopa and Actor / Coin and Item
  We have an inheritance relationship between Ground and Dirt classes. Note inheritance allows reusable code to be extended from another class so we only need to implement unique features/functionality inside a class and leave the common/general features to super class. This design improves both maintainability and scalability besides reducing the repeating codes.

  Compare this design to another design where inheritance is not used and both Dirt and Ground are independent of each other. We note that the DRY (Do not Repeat Yourself) principle is broken as we have to repeat the common functionality of both Ground and Dirt in their respective classes. Imagine we have multiple types of ground which would then raise issues on maintainability and scalability.
  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.

- Dirt and FertileGround
  We decide that Dirt and FertileGround should have an “implement” relationship. Note that such design always avoids multiple level inheritance from which encapsulation and polymorphism may fail, allow multiple inheritance which is forbidden using inheritance relationships and reduce the number of dependencies.

  Imagine that we have another design such that we do not need the interface FertileGround but 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.

- Tree and SpawnManager
  Note that 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.

  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 being spawned will not affect the Tree class but the SpawnManager. 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.

- Tree and Status
  According to the specification, 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.

  Instead, if we 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.

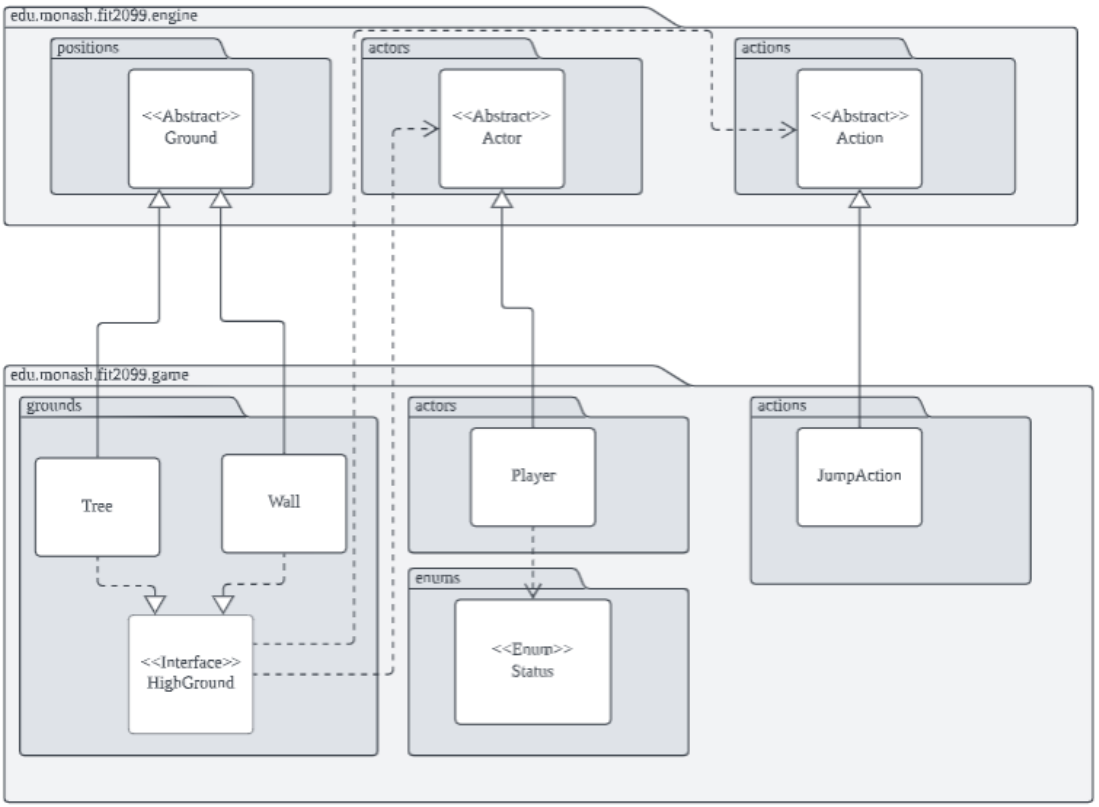
- Spawnable and SpawnManager
  All the spawn objects should implement Spawnable and SpawnManager should interact with the Spawnable instead of the objects respectively. This is aligned with the dependency inversion principle.

  Without the use of Spawnable interface, it is highly likely that the SpawnManager has dependencies with every single spawnable objectand 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.

- Actor and Item / Ground and Location
  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.

Task 2

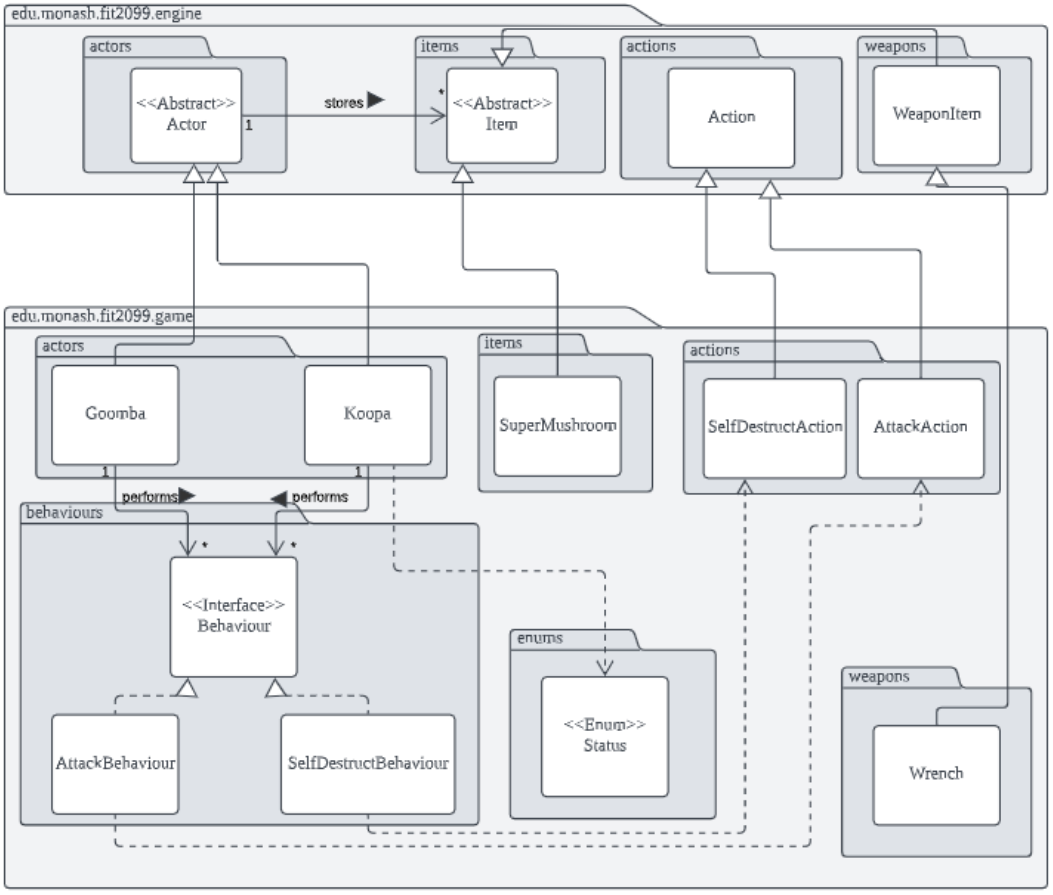


- Tree and Ground / Wall and Ground / Player and Actor / JumpAction and Action
  Refer to Tree and Ground in Task 1. They share the same explanation.

- Tree and HigherGround / Wall and HigherGround
  Refer to Dirt and FertileGround in Task 1. They share the same explanation. However, add-ons are included here since the design does follow Interface Segregation Principle.Imagine we have a single interface that handles all types of ground, i.e. FertileGround and HighGround. This would violate the principle since not every ground is both fertile and high. Then, those fertile ground would not need capabilities from high ground and vice versa. Extra handlers have to be added to make sure that. This would violate the SRP principle as well since a single interface has to take care of different responsibilities, i.e. fertile and high ground. As a result, the above design is chosen, i.e. every interface should be small and only responsible for what they are intended to.

- Player and Status
  Refer to Tree and Status in Task 1. They share the same explanation.

Task 3



- Goomba and Actor / Koopa and Actor / SuperMushroom and Item
  Refer to Tree and Ground in Task 1. They share same explanation.

- Goomba and Behaviour / Koopa and Behaviour
  Refer to Spawnable and SpawnManager in Task 1. They share a similar explanation, i.e. behaviours of the enemies should interact/defined via the abstraction and not direct dependency. Note Behaviour and specific Behaviour are similar with Spawnable and other spawnable objects, i.e. coin and enemies. The use of interfaces to create “implement” relationships is mainly about maintainability and scalability for better control of the objects.

- Koopa and Status
  Refer to Tree and Status. They share the same explanation.

Overall
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.

Class Responsibility

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.