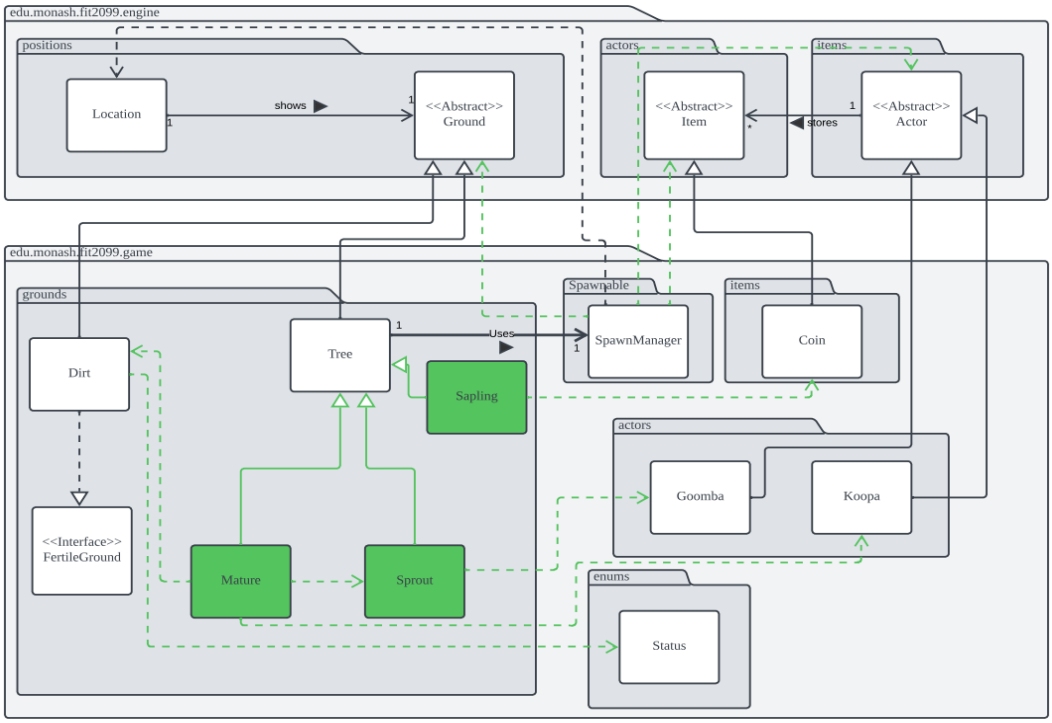
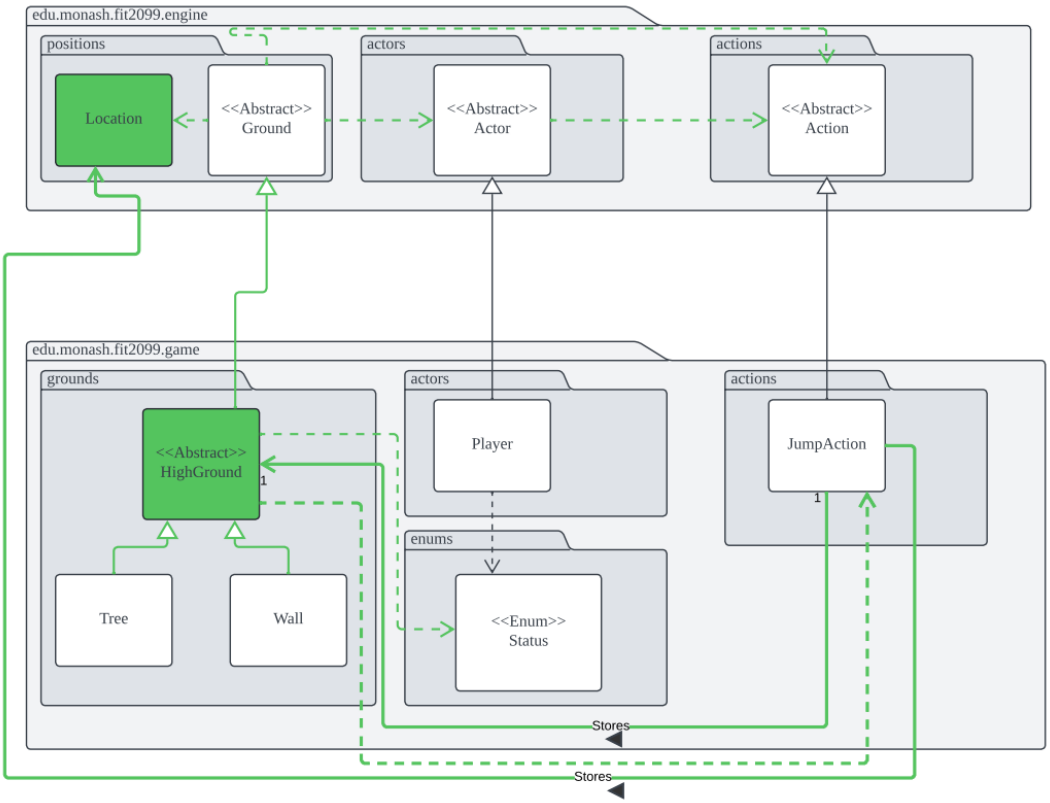
FIT2099 Assignment 2: Design Rationale

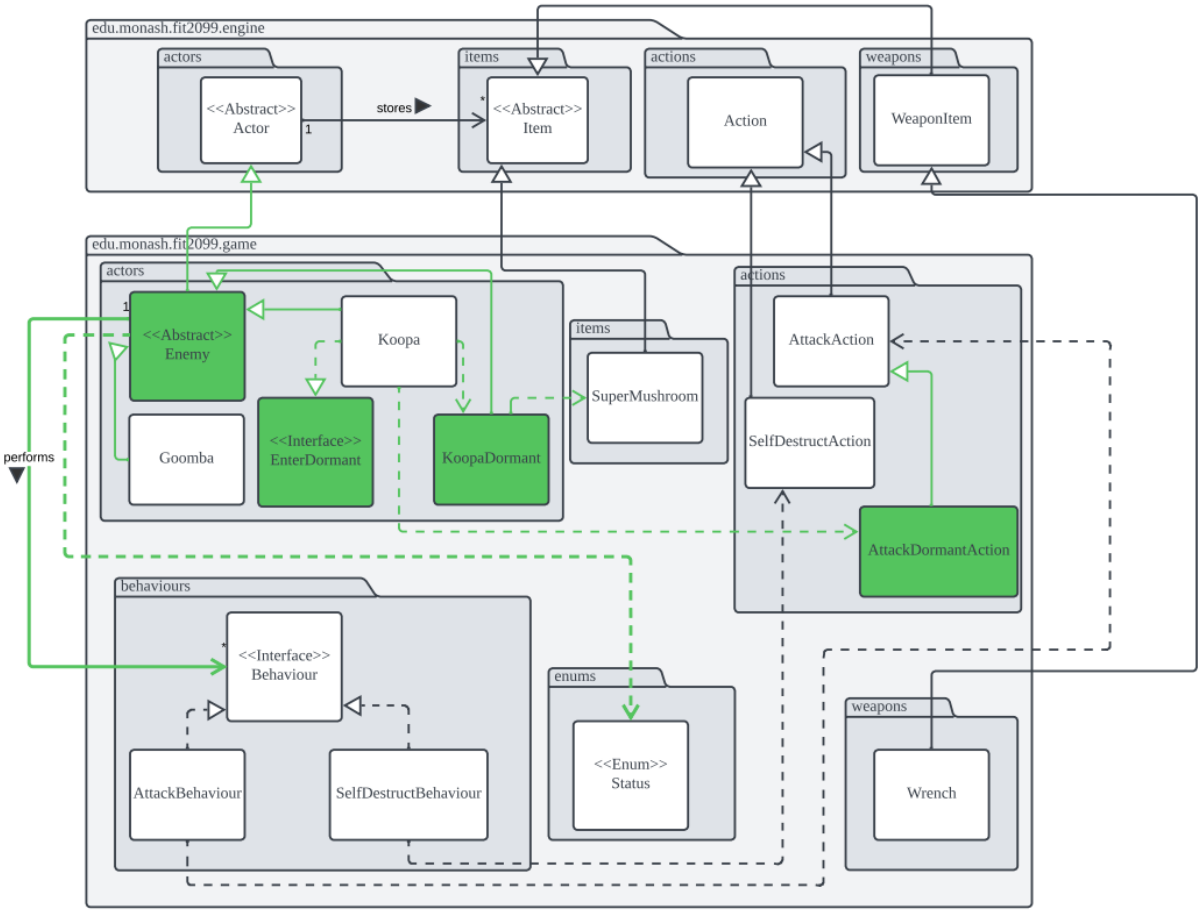
Task 1



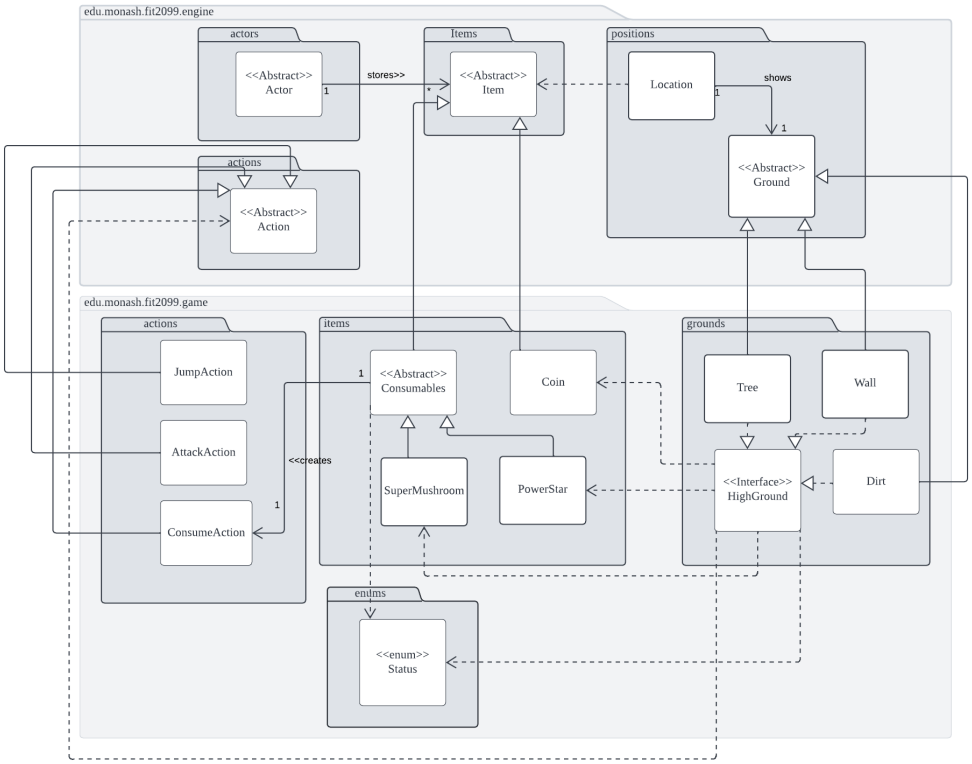
Task 2



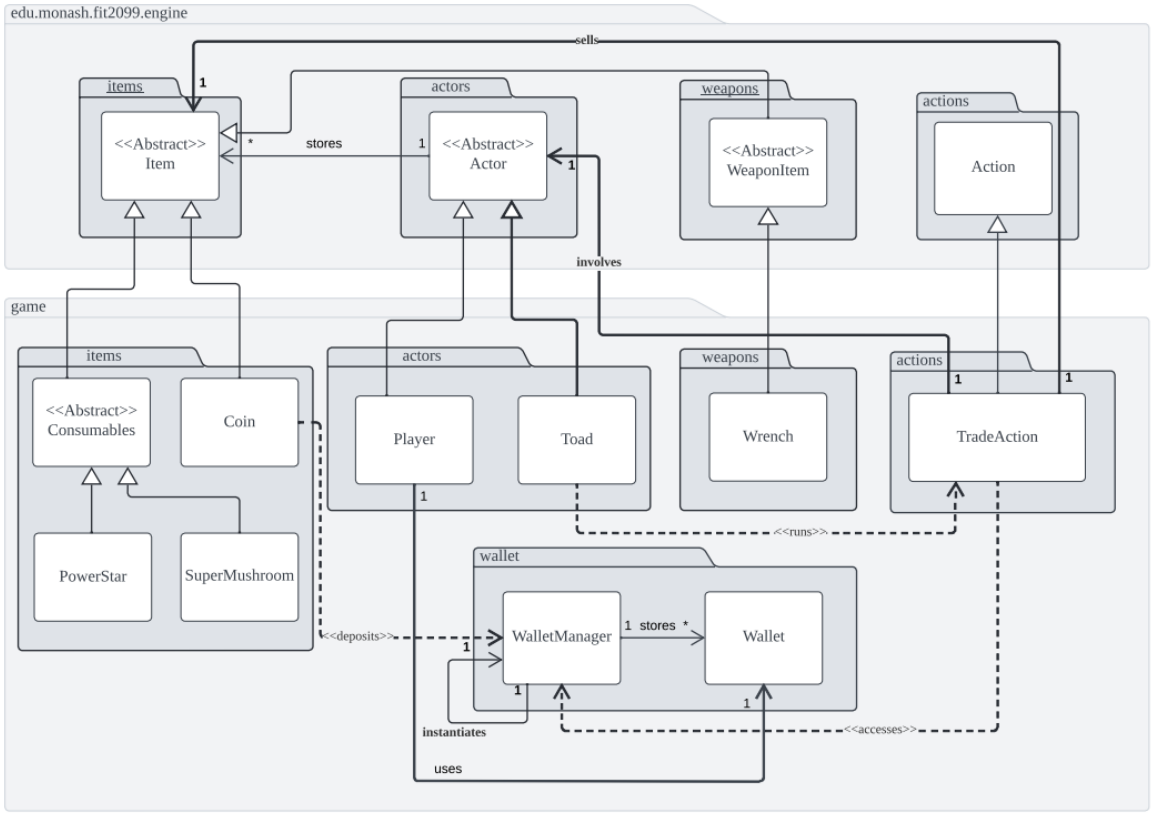
TASK 3


edu.monash.fit2099.engine
actors
<<Abstract>>
Actor
stores
items
<<Abstract>>
Item
actions
Action
weapons
WeaponItem
edu.monash.fit2099.game
actors
<<Abstract>>
Enemy
Koopa
Goomba
<<Interface>>
EnterDormant
KoopaDormant
items
SuperMushroom
actions
AttackAction
SelfDestructAction
AttackDormantAction
performs
behaviours
<<Interface>>
Behaviour
AttackBehaviour
SelfDestructBehaviour
enums
<<Enum>>
Status
weapons
Wrench

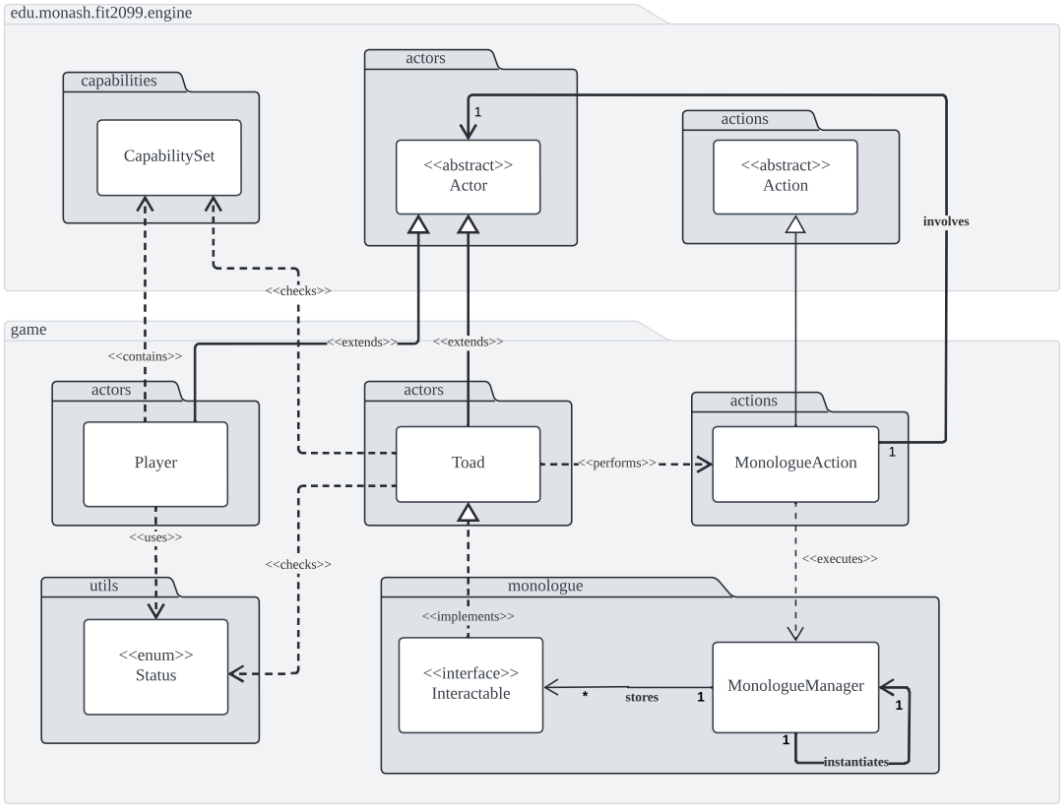
TASK 4


edu.monash.fit2099.engine
actors
<<Abstract>>
Actor
stores>>
Items
<<Abstract>>
Item
positions
Location
shows
<<Abstract>>
Ground
actions
<<Abstract>>
Action
edu.monash.fit2099.game
actions
JumpAction
AttackAction
ConsumeAction
<<creates
items
<<Abstract>>
Consumables
Coin
SuperMushroom
PowerStar
grounds
Tree
Wall
<<Interface>>
HighGround
Dirt
enums
<<enum>>
Status

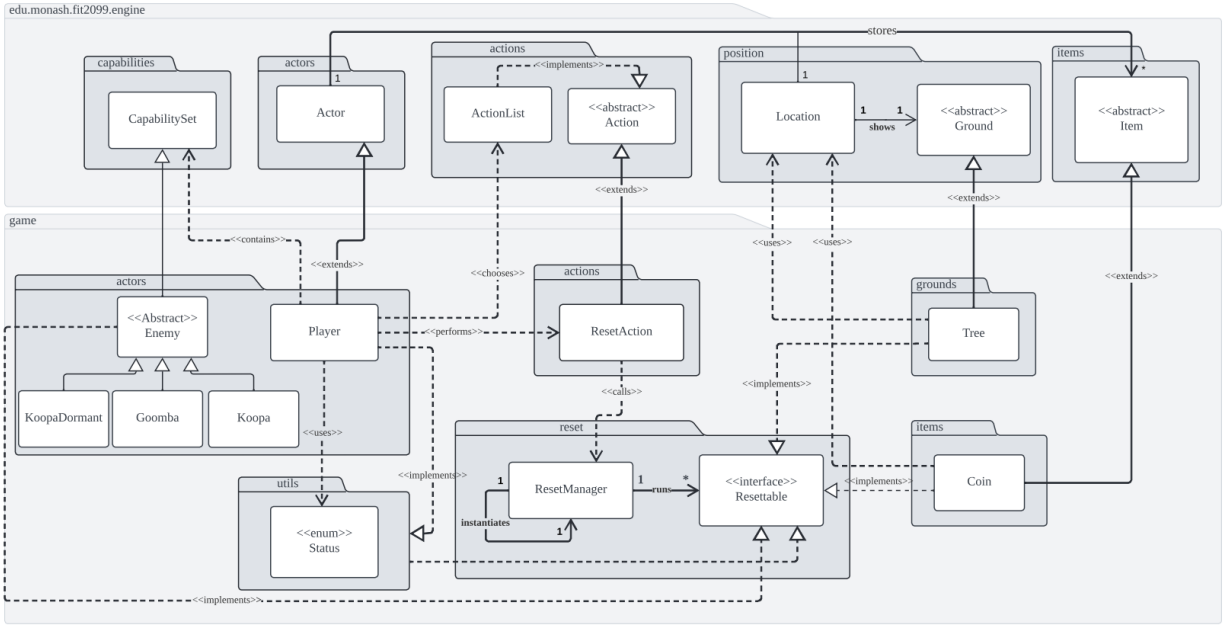
Task 5


edu.monash.fit2099.engine
sells
items
actors
weapons
actions
<<Abstract>>
Item
stores
<<Abstract>>
Actor
<<Abstract>>
WeaponItem
Action
involves
game
items
actors
weapons
actions
<<Abstract>>
Consumables
Coin
Player
Toad
Wrench
TradeAction
PowerStar
SuperMushroom
<<runs>>
wallet
<<deposits>>
WalletManager
stores
Wallet
instantiates
<<accesses>>
uses

Task 6


edu.monash.fit2099.engine
capabilities
actors
actions
CapabilitySet
<<abstract>>
Actor
<<abstract>>
Action
involves
<<checks>>
game
<<contains>>
<<extends>>
<<extends>>
actors
actors
actions
Player
Toad
<<performs>>
MonologueAction
<<uses>>
<<checks>>
<<executes>>
utils
monologue
<<implements>>
<<enum>>
Status
<<interface>>
Interactable
stores
MonologueManager
instantiates

Task 7



Inheritance and Polymorphism and Single Responsibility Principle

Note that we created 3 new classes, inherited from Tree, in Task 1. Those classes represent the stage of the tree respectively. This new design is better suited to OOP principles since the 3 stages have different behaviours (different responsibilities) and only share some common attributes. This helps us make use of polymorphism to evaluate the behaviours without violating Single Responsibility Principle. Compared to the previous design in Assignment 1 where we use Enum type to differentiate the different stages, it is a better design since we maintain relatively small classes each with their own responsibilities instead of having multiple if-else statements to handle different responsibilities (different behaviours at different stages). The HighGround is changed from Interface to abstract class since all the high grounds would share common attributes like fallDamage and jumpSuccessRate as well as the behaviours like allowableActions. Using an interface, we cannot make such generalisation since no private attributes / functions are inherited. Furthermore, repeated codes would show up if an interface is used instead of an abstract class, violating DRY principle. Other than that, we create a new abstract class, Enemy, in Task 3 which allows other enemies classes like Goomba and Koopa to inherit. This new class generalises the common attributes between the enemies entities, for example behaviours and some common actions under the same scenario. Without this generalisation as in previous design in Assignment 1, the principle of “Do Not Repeat Yourself” (DRY) would be violated. Lastly, AttackDormantAction is created to handle any attacks involving actors that can enter a dormant stage. This would ensure Single Responsibility Principle is followed where AttackAction is changed when the behaviour of normal attack is changed and AttackDormantAction is changed when the behaviour against enemies that can enter dormant stage is changed. This would also reduce the overflooded use

of if-else statements to handle different cases besides reducing the maintainability of AttackAction class when new features related to attack action are introduced (new attributes may need to be added and more if-else statements used). In requirement 6, the ”ineteractable” interface is created to allow classes which can interact with the player to implement their own way of communicating with the player. This way, it abides with the Single Responsibility Principle (SRP). Besides that, for requirement 7, the “resettable” interface allows for classes which are affected by reset action to implement it’s own way of resetting itself, which also ensures that SRP is maintained

Open-Closed Principle

Using the above design where inheritance and interfaces are used, extensibility of the design is easily achieved. Taking an example, we can easily create a new stage of a tree just by creating a new class inheriting Tree. The Tree is open for extension but close for modification, i.e. we do not need to modify the Tree class to suit the new stage. Similarly, any new high ground would simply need to inherit the abstract class HighGround without any modification on the abstract class needed. Other than that, any enemies that can enter the dormant stage can be easily extended by implementing the interface EnterDormant and creating a new class for its dormant state. New enemies could be extended by inheriting the enemy class as well. This is better design compared to previous design in Assignment 1 since use of interface might require modification to suit every high ground and no generalisation on common private attributes or behaviours could be made. Furthermore, we note that AttackDormantAction is inheriting AttackAction instead of Action. This is because AttackDormantAction is exactly the same as AttackAction except that it can handle the enemy entering the dormant stage. This is another example of open-closed principle since every attack causing debuff (in the future) could be extended from AttackAction without modifying AttackAction. This allows high extensibility of the system and can accommodate new features easily.

Liskov Substitution Principle

Note that all the inheritance relationships above follow this principle. Looking at the engine code, we can see that methods in the engine code use the parent class as the parameter signature instead of the child class. However, all the child classes created in the above design have no problem in substituting their parent class in those methods. However, there is a limitation in the engine code which may cause some logical bugs not found during the compile time. For example, only an Actor can be added to the map using addActor method. This required that the variable passed into the method to be an Actor, resulting that Actors who are not of the type EnterDormant can be passed into the AttackDormantClass. Hence, any logical bugs have to be taken care of cautiously. Other than that, we can also reduce the dependency since polymorphism allows us to ignore the exact type of the child class but using the parent type.

Interface Segregation Principle

All the interfaces used in the above design remain small and only focus on 1 single capability instead of having more than 1 capabilities. For example, FertileGround is only used for the ground which is classified as FertileGround. Other classifications of the ground would have to rely on other new interfaces. The same thing applies to EnterDormant Interface. This is to make sure that our implementation also follows the SRP stated above. This maintains the extensibility and maintainability of the above system since we can easily extend a required capability using the interfaces.

Dependency Inversion Principle

An example in the above design is that Enemy access the low-level behaviours or actions using the Behaviours Interface. The interface acts as an abstraction for Enemy to access different actions and behaviours without having dependency on those low-level actions/behaviours implementations. This greatly reduces the number of dependencies in the above design and also maintains the extensibility.

General

Use of literals are greatly minimised with the use of Enums. This would greatly reduce the possible flaw in comparing literals and also overflooding the codes with if-else statements which is difficult to maintain. Codes implemented follow the above principles and maximise the use of polymorphism (overriding) to achieve the reusability of the methods. Furthermore, only necessary methods like getter and setter are defined where applicable in order to avoid new dependencies or leak of private information (incomplete encapsulation). For example, attributes related to only specific classes are set to be private while attributes related to child classes are set to be protected or accessed using getter/setter to ensure encapsulation.