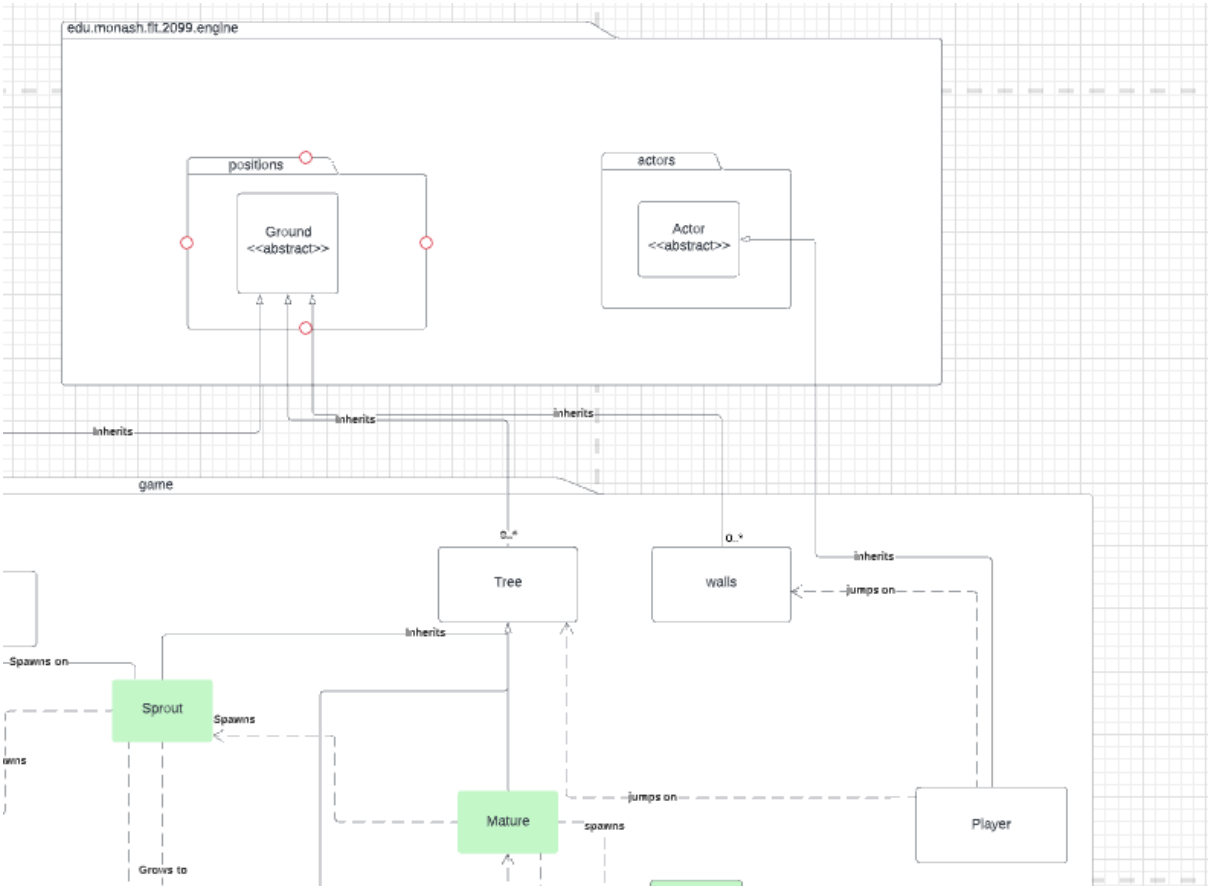
Assignment 1 FIT2099 Design Justification

Task 1 and Task 2: Philip

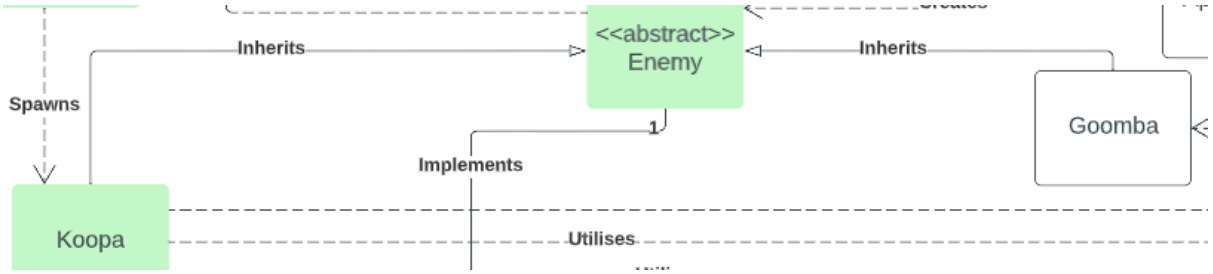


During the first task of our assignment, the player has a dependency of tree instead of each sprout, sapling and mature. This is used to implement the solid principle of dependency inversion when coding the damage and jump chance which will be calculated through an overridden method in each class which is then inherited from the tree class. This is all then inherited through a singular class called Tree. This allows the function to work through abstractions. The same will be implemented for the wall class as it will implement an abstract method from ground. This will allow us to implement different damages and chances depending on the method and not the different types of classes. This same method of inheritance applies when creating the sprout, sapling and mature.

Task 3: Amogha

This task implements the enemies in the game (currently Koopa and Goomba), and determines how they interact with the player and reside in the world.

The first design decision was to implement an abstract class Enemy, as both Koopa and Goomba share some functionality, which they could then inherit from this class.



Additionally, there is the potential for future enemies to be added later on, which could also inherit from Enemy. There is a set of behaviours that all enemies implement, such as wandering, following and attacking. As a result, Enemy implements the interface Behaviour, which has AttackBehaviour, WandererBehaviour and FollowBehaviour that also implement it. AttackBehaviour also uses AttackAction to carry out, which inherits from Action.

Importantly, as said prior, we can see that attacking, wandering and following are behaviours all enemies will implement. However, Koopa will sometimes lay dormant, and Goomba will sometimes commit suicide, but these are behaviours not shared between enemies. Therefore, these behaviours will be implemented as individual methods within the required Koopa and Goomba classes. Koopa also implements the DoNothingAction class, which will be within this method to enable it to be dormant when required.

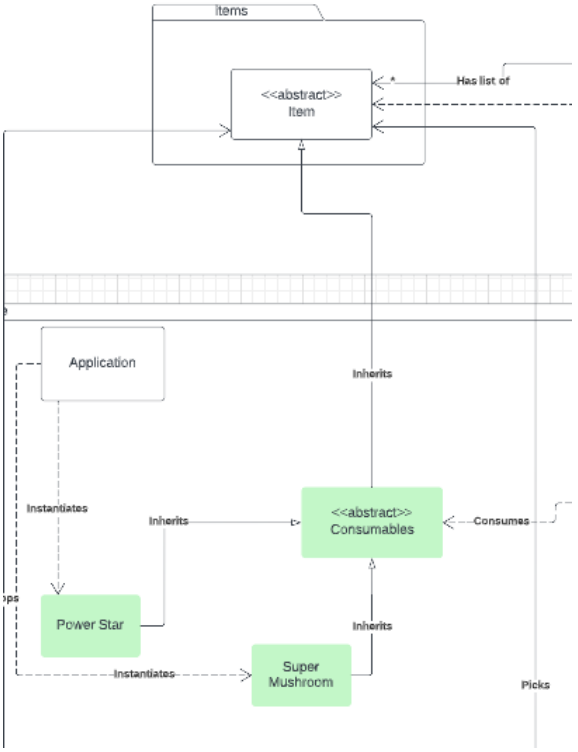
Koopa also drops a super mushroom once its shell is destroyed, so it also uses DropItemAction. However, another implementation of this could be to just remove the Koopa from the map and then spawn in a super mushroom in its location, but this would be redundant. Sprout has a 10% chance to spawn Goomba in every turn, and Mature has a 15% chance to spawn Koopa in every turn, hence this is also depicted in the UML diagram.

Enemy inherits from Actor, and has a list of actions that it can carry out, so it is dependent on the ActionList class.

The sequence diagram covers an enemy attempting to detect the player. If it detects them, it will attempt to attack and the player potentially loses HP.

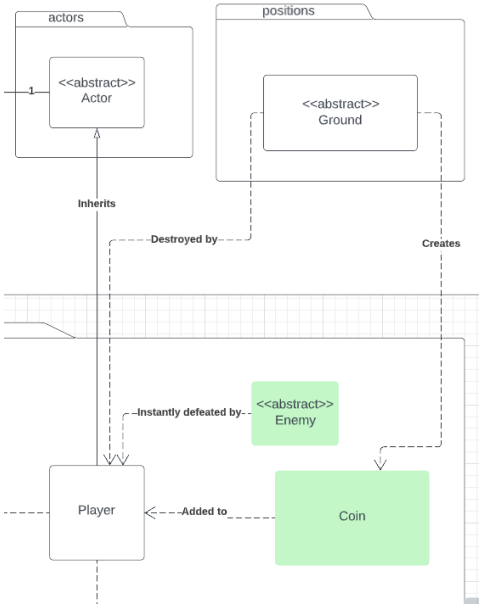
Task 4: Amogha

Task 4 implements the magical items in the game, specifically Power Star and Super Mushroom. To begin, the two classes for these items would inherit from an abstract class Consumables (as can be seen on the right), where similar methods could then be employed by both. Both magical items are also linked to Application, as they are instantiated along with the player at the beginning of the game. In addition, Consumables would inherit from Items, as there are some items that cannot be consumed (but all consumables are items). If we only used Items, we are likely to run into further issues down the road (code smell), and therefore, by having this intermediate class, we are following Liskov's Substitution Principle.



Items can be picked up or dropped using the DropItemAction and PickUpItemAction classes, both of which inherit from Action. This would be used when Mario picks up either of the magical items on the map, or when Super Mushroom is dropped from a Koopa.

As the player will consume these magical items throughout the course of the game, there is a dependency between Player and Consumables. The player will also buy magical items from Toad, an NPC in the game, which contains its own items.



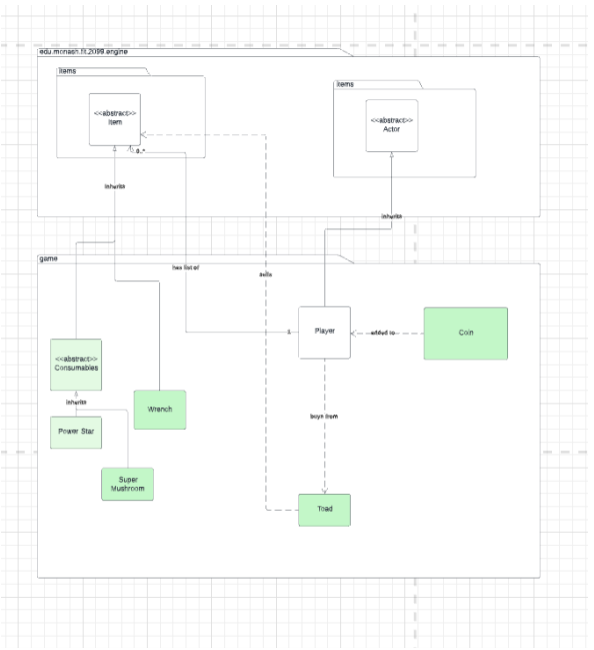
Note that, whilst the player has consumed Power Star and the effects have not worn off yet, the player will destroy high ground to dirt (so there is a dependency between Ground and player), destroyed ground is converted to coins (so there are dependencies between Ground and Coin, as well as Coin and Player) and lastly, enemies are instantly defeated (so there is a dependency between Player and Enemy).

Lastly, Player inherits from Actor, which has a list of actions that it can carry out, so it is dependent on the ActionList class.

Task 5: Ollie

When designing the coin and money aspects of the game, the use of a wallet was decided rather than the player holding the coins. This will allow easier access to the coin value the player has for both purchases and collecting coins from the map.

Another abstract class for consumable items which extends the item abstract class was also added. As both of the power star and super mushroom can be consumed by the player, they will have some similar attributes that the wrench (another item) will not have such as the ability for mario to use them.



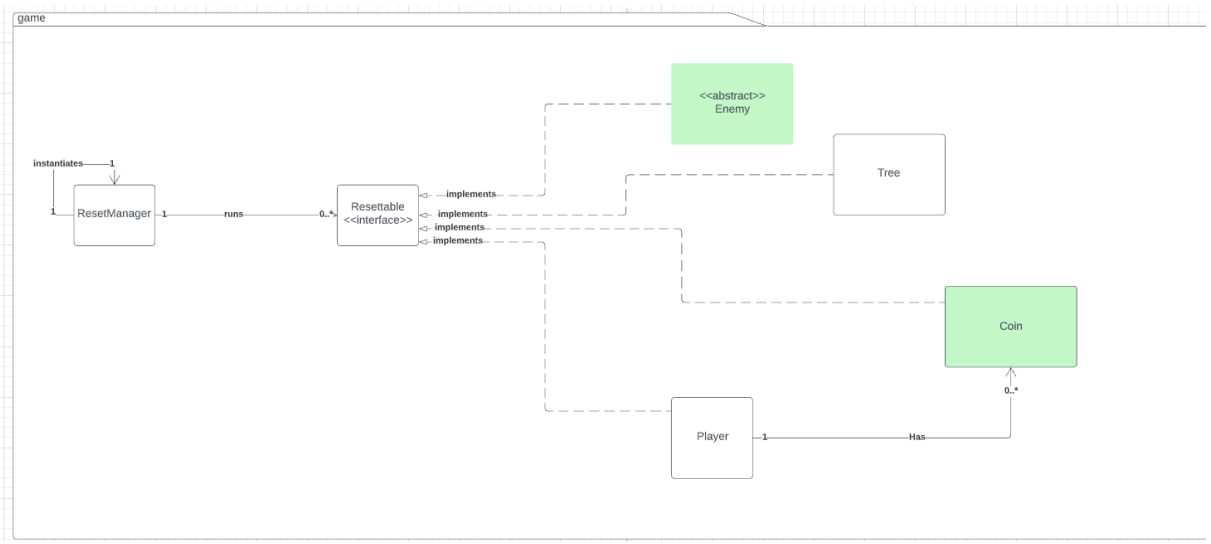
The player only has an association to the item class. This introduces the design principle of ReD (reduce dependencies). The inventory of the player is a list of the items the player has collected or purchased. This also uses the SOLID principle of Liskov substitution as the array list contains items however these items can be any item that extends the item class such as the power star, super mushroom or wrench.

Task 6: Ollie

When the player decides to interact with Toad, the code will check the player's item inventory to see whether a wrench is present and will check the player's status to see if they have an active power star. If one of these is true, the monologue line options for Toad will be reduced accordingly to avoid Toad using redundant lines in the situation. A random function will be used in order to generate a random monologue line as this will be the easiest way to do so. If the player has a wrench the first given line will be eliminated and if the player has an active power star the second line will be eliminated. The random function will be used in all situations regardless of the removed lines (used with the remaining lines).

Task 7: Joint contribution

Our reset functionality is built on dereferencing addresses. Our method of a reset game lies in clearing any pointers to the classes which allows java to clear them through garbage collection. Then new classes are instantiated, thus clearing the game.



This is shown in our UML diagram where the singular interface of reset is enacted upon all the different classes which allows them all to inherit the reset methods. This aligns with the DRY principle as no code will be duplicated over many different classes in order to reset each individual class. In addition this design implements the Liskov Substitution Principle as the reset manager class contains a list of resettable items and they can be replaced by any of the classes that implement the same interface.