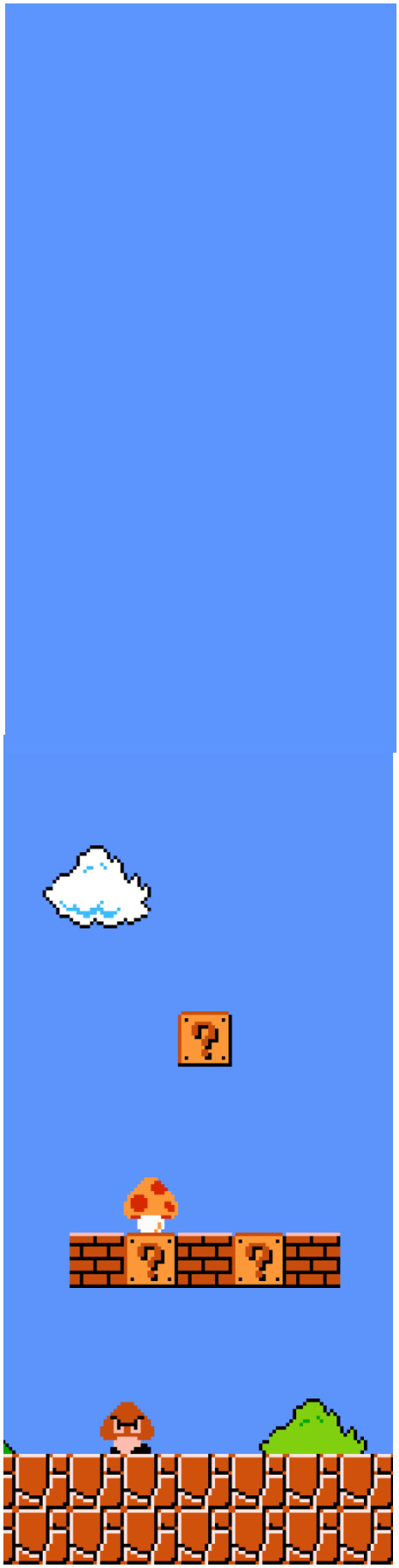


FIT2099 Assignment 2 Design Rationale

MONASH University

Nethmini Botheju, Yin Lam Lo, Jasper Martin
Lab 14 Team 7

Requirement 1

Tree Class
The tree class will be abstract. Each type of tree will be its own class, a child of the tree class. The actions specific to these types will be implemented in these child classes.

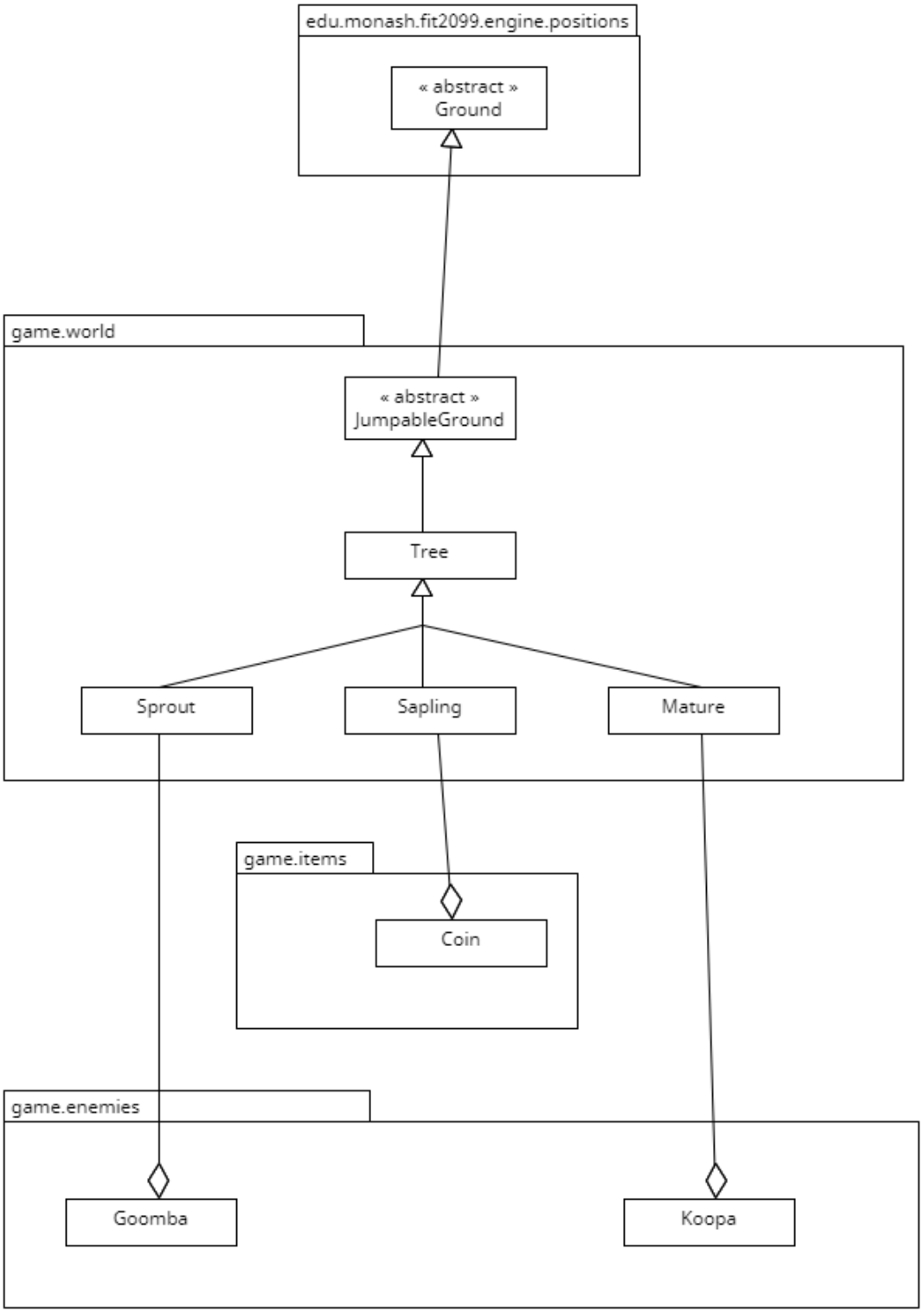
- Contains an age attribute
- Every 10 ticks of age, changes the object to the next type of tree. This will be overridden in the mature tree
- Has a 50% chance to delete the tree object and change the ground to dirt. (For resetting the game)

Sprout Class
- 10% chance to instantiate a Goomba object on the location if there is no other actor on the location, without affecting the tree object.

Sapling Class
- 10% chance to instantiate a $20 coin object on the location, without affecting the tree object.

Mature Class
- 15% chance to spawn Koopa object on the location if no other actor is on the location, without affecting tree object.
- Every 5 ticks past growing to be mature, will attempt to instantiate new sprout object on adjacent dirt. If no dirt is available, nothing will happen.
- 20% chance to delete the tree object and change location state back to dirt.

UML Class Diagram:



Changes during implementation:

- The choices in implementation of the tree classes were similar to how it in the original UML diagram. The only notable change was the addition of the JumpableGround abstract class between the Ground class and the Tree class, which was used to help implement the jumping functionality, as trees are not the only

ground objects that can be jumped up to. This helps follow **DRY rules** as each stage of the tree inherits the attributes of parent class Tree without having to rewrite the methods.

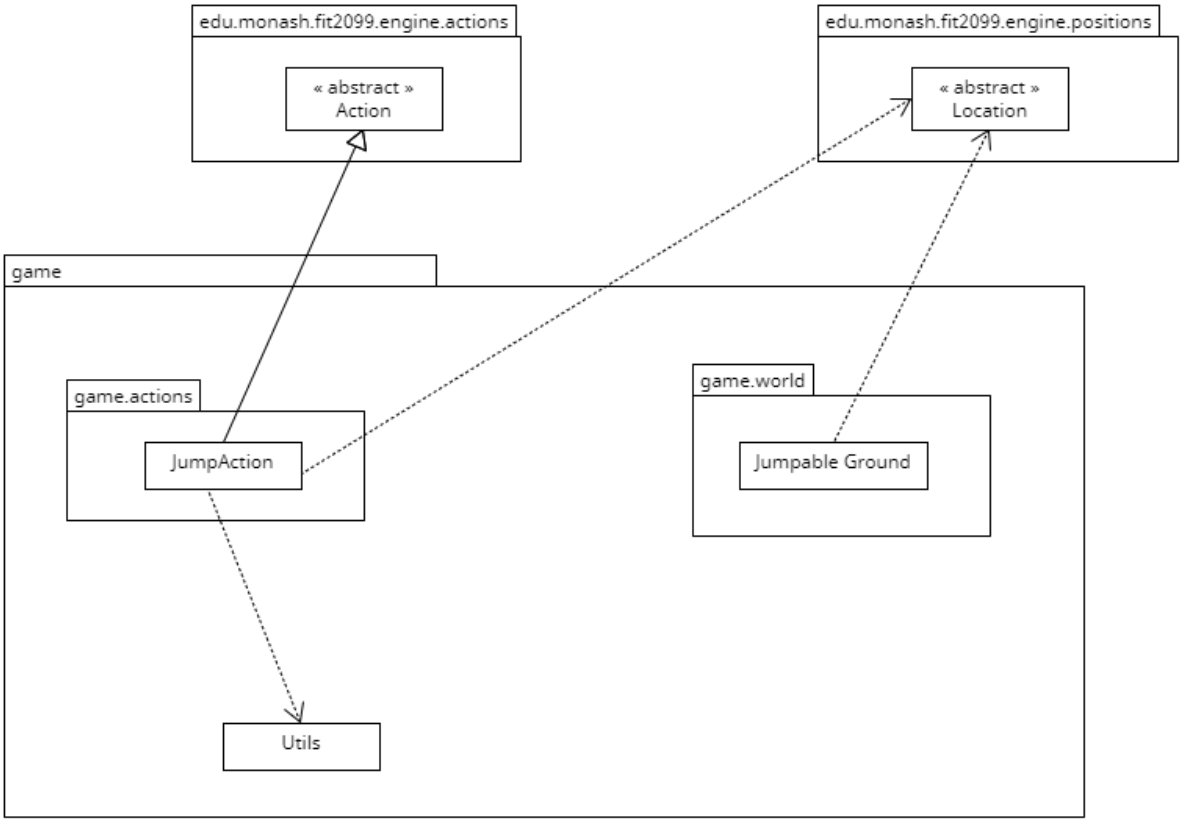
- The Tree class has become abstract and is being extended by Sprout, Sapling, and Mature Class to avoid implementing different states and becoming God class. The responsibility has been split into 3 subclasses which are Sprout, Sapling, and Mature Class and which class can only focus on their state, which avoids violating the **single responsibility principle**.
- Another minor change was that was planned was to have trees with continuous age, however it was simpler to give each new instance an age of 0, so whenever a tree grew, the age would reset to 0. This is better, as it means that even if different tree growth stages are added between the current ones, none of the current code needs to be changed in order to change the age of growth (10 turns).

Requirement 2

Jump Class

The JumpAction class will handle the player jump. This will contain the probability of success for each ground type, as well as the fall damage if the player does not succeed. It will also contain the method to carry out the jump action, as well as to then either move or hurt the player. The class will also check if the player has the SuperMushroom super jump capability and will act accordingly if the player does have this.

UML Class Diagram:



Changes during implementation:

- The choices in this requirement were also similar to the original plan. There is a JumpAction class that handles the logic of the actual jump, including different success rates, damage values, etc. However a new class JumpableGround was added to help add the JumpAction to available actions for each type of ground that can be jumped to. This was the only real change to the plan.
- JumpAction calls the abstract class Utils to gain a percentage of the player's ability to land a successful jump.
- JumpableGround deals with the player's ability to jump onto higher ground if PowerStar is active while JumpAction deals the action and it's implementation. This allow the individual classes to not violate the Single Responsibility principle as each class tend to their own responsibilities and changing one class will not affect the other.

- My choices in implementation of the tree classes were similar to how I envisioned it at the start. The only notable change was the addition of the JumpableGround abstract class between the Ground class and the Tree class, which I used to help implement the jumping functionality, as trees are not the only ground objects that can be jumped up to.
- This helps follow DRY rules. Another minor change was that I planned to have trees with continuous age, however it was simpler to give each new instance an age of 0, so whenever a tree grew, the age would reset to 0. This is better in my opinion, as it means that even if different tree growth stages are added between the current ones, none of the current code needs to be changed in order to change the age of growth (10 turns).

Requirement 3

Enemies Class

Enemies are like “Goomba” are originally extended from the class “Actor”. Enemies have different characteristics compare to ”Player”. And we also have another enemy “Koopa”. Instead of making another class called ”Koopa” extends from the class “Actor”, a new abstract class “Enemies” are created as the parent of “Goomba” and “Koopa”. Because they are enemies that shared a lot of similarity. Making “Goomba” and “Koopa” directly extend from “Actor” will repeat lots of code and violated “Don’t repeat yourself” principal.

The “Enemies” class groups the common point of “Goomba” and “Koopa”:

- Shared Attribute: hp, damage, hitRate, disappearRate, displayChar
- Enemy cannot walk through the ground type “Floor”
- Spawn from the ground type “Tree”
    - GameMap.addActor(Actor actor, Location location)
- If the target attacks them, or within their radius
    - Enemy should start following the target and attack if within range
    - this.behaviours.put(priority, new AttackBehaviour());
    - Actor.getIntrinsicWeapon()
- Else
    - Enemy will wander around until they were killed/removed
    - this.behaviours.put(priority, new WanderBehaviour());

Different value and condition for implementing “Goomba” and “Koopa”:

- Spawn Condition:
    - Goomba: 10% from ground type “Tree” in Sprouts state
    - Koopa: 15% from ground type “Tree” in Mature state

- HP:
    - Goomba: 20
    - Koopa: 100

- Attack Damage:
    - Goomba: 10
    - Koopa: 30

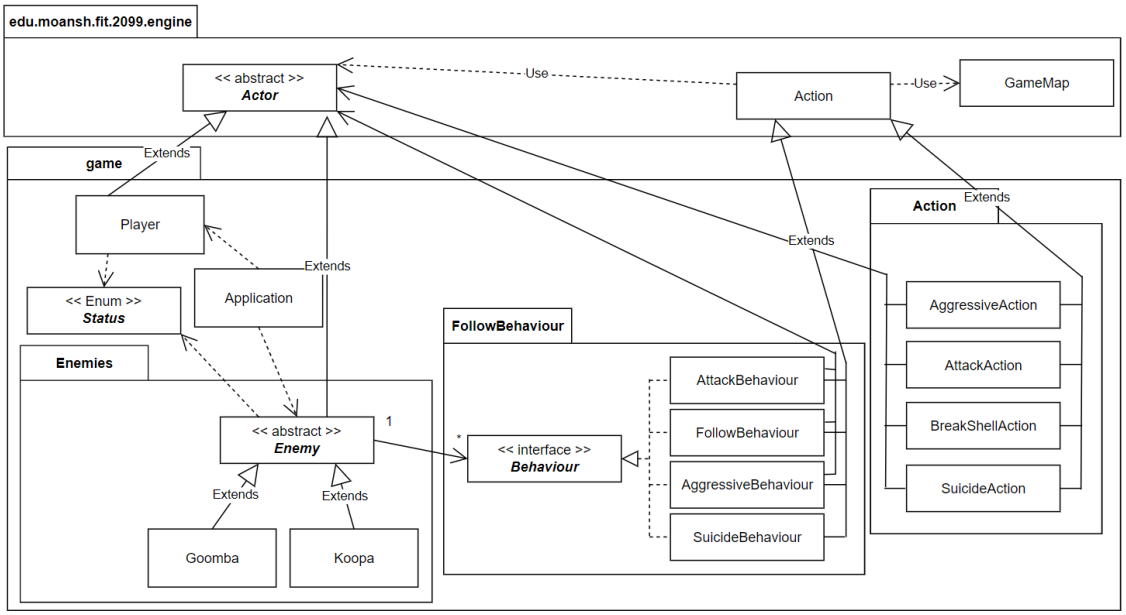
- Hit Rate:
    - Both: 50% (Default)

- Condition to get remove from the map:
    - Goomba: HP <= 0
    - Koopa: Get hit by wrench in dormant state

- Item drop when killed:
    - Goomba: None
    - Koopa: SuperMushroom

- Dormant state (Koopa only):
    - When HP <= 0, stay on the ground cannot attack nor move)

UML Class Diagram:



Changes during implementation:

- By creating a new abstract class Enemy, it will achieve the **Open/closed principle**. If we want to add a new enemy, we just have to create a new enemy class that extends from Enemy. But the core of the enemy's behaviour is inside the Enemy Class, if we want to add new functionality, we could do it the Enemy Class and every enemy could have the same method from their parent class. So part of the code from the original Goomba class was moved to the new enemy class with some more methods added for more functionality for the requirement.
- The Goomba class extends from Enemy class and spawn with more Behaviour than before, so it could suicide at a chance and be aggressive to the player. The Koopa class is similar to Goomba but it will change DisplayChar according to whether koopa is dormant or not.
- Part of the implementation of this part goes according to the plan, except for the enemies actions. With more understanding of the code and notice the enemies actions are based on behaviour's actions priority list. So new Action and Behaviour was implemented to Achieving Goomba Suicide and enemies behaviour changing.

Requirement 4

Items Class
New classes “SuperMushroom”, “PowerStar”, and “Coin” will be created extending from parent class in engine “Item”. And “Wrench” will be extending from “WeaponItem”. Since Item have attribute of capability, “SuperMushroom” and “PowerStar” should store “TALL” and “STAR” capability individually. When player consume the item from the inventory, the player should get the capability from the item and add it to player’s capability list. The status are given from detecting capability on the player, not the player consumed item or not.

New attribute should be added to “Player”:
- Integer mushroomEaten; (to determine maximum health)
- Integer starTimer; (to keep track the star effect duration)

When Player consumed ”SuperMushroom”:
- Player get “TALL” status in its capabilitiesSet
    - Player.addCapabilityList(“TALL”)

- Change the display character to “M”
    - Actor.setDisplayChar(“M”)

- Depends on R2: JumpAction Class
    - override all jumpRate = 100%
    - override all fallDamage = 0

- Increase Max HP + Heal
    - Count how many mushrooms eaten: mushroomEaten++
    - Set the max HP: Actor.resetMaxHp(100 + 50 * mushroomEaten)

When Player got damaged:
- Remove tall status from player’s capabilitiesSet
    - Player.removeCapabilityList(“TALL”)

- Change the display character to “m”
    - Actor.setDisplayChar(“m”)

- Remove jumpRate and fallDamage override
- Keep Max HP (Nothing to implement)

When Player consume “PowerStar”:
- Player get “STAR” status in its capabilitiesSet
    - Player.addCapabilityList(“STAR”)

- Start starTimer countdown for 10 turns
    - starTimer++

- Makes player hittable with 0 damage
    - Disable hurt() execute using if

- R2: Bypass jump action
- Set ground to dirt when the ground is Wall/Tree
    - Location.setground(Dirt)
    - Location.addItem(Coin)

- Kill attack target if hit

When starTimer == 0:

- Disable abilities the star given

All the capability changes for player will be processed inside the Player class, to achieve single responsibility principle for managing the changes of the players.

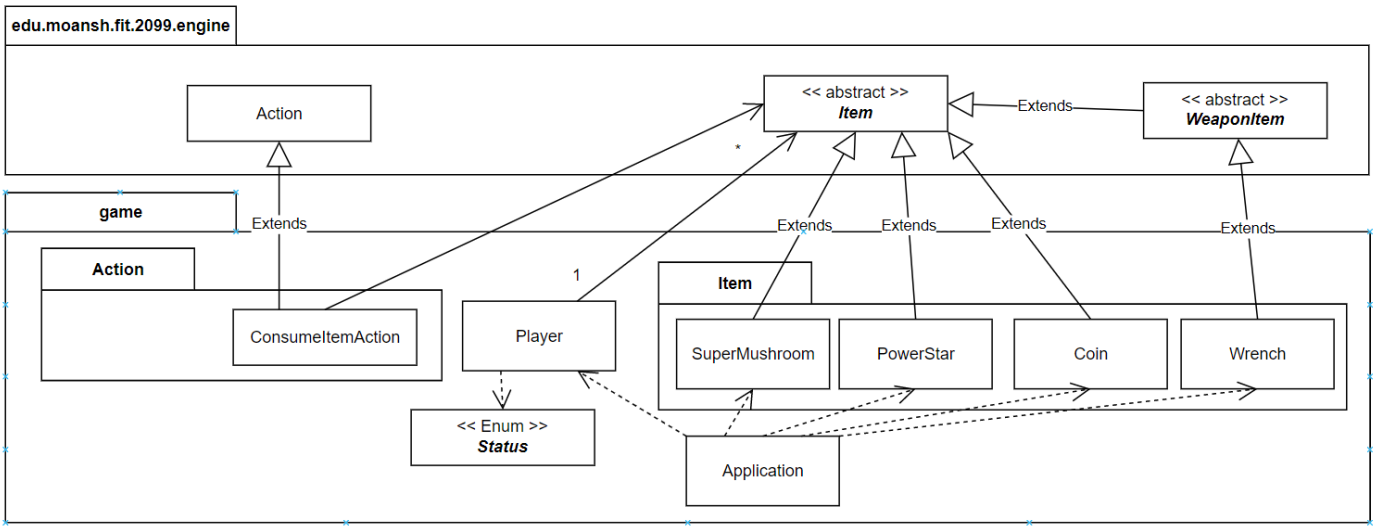
Wrench Class

Extends from ”WeaponItems” and has Capability, with 80% hit rate and 50 damage. It gives player ability to break the shell. Should show option to break the shell when near a shell with a wrench in the inventory.

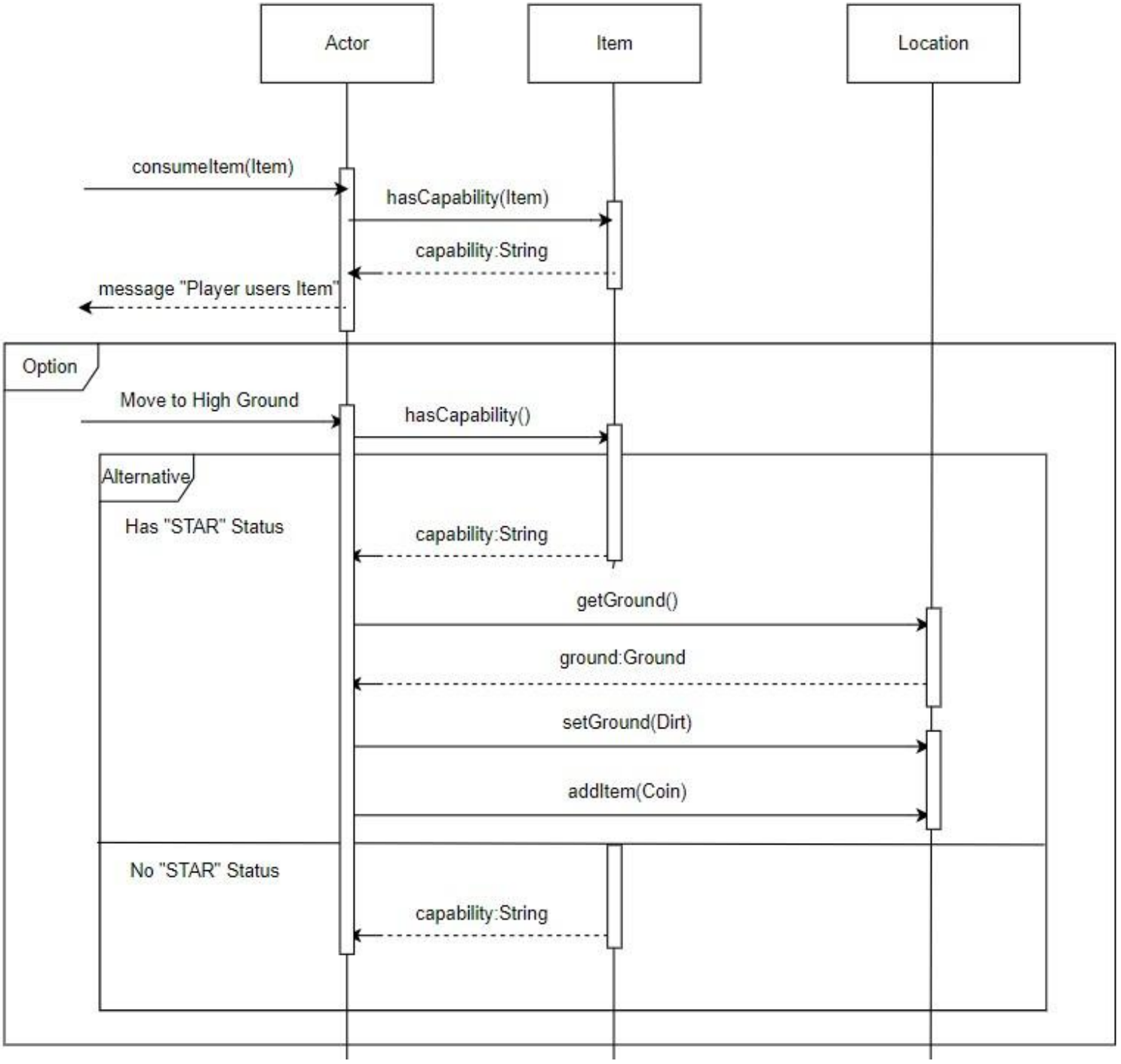
When Items are on the Map

When “PowerStar” is initialised on the map, an attribute in the star will count despawn time, while another item will not. Picking up “PowerStar” or obtaining it from Toad will consume instantly, but other items can be stored in the inventory.
For ConsumerItemAction, it will check the class of the instance to ensure which type of item is it and call the correct method in player to add different capabilities

UML Class Diagram:



Sequence Diagram:



Changes during implementation:

- All the capability changes for players will be processed inside the Player class, to achieve the **single responsibility principle** for managing the changes of the players. And ConsumeItemAction is managing which methods to call for corresponding capability methods.
- For ConsumerItemAction, it will check the class of the instance to ensure which type of item is it and call the correct method in the player to add different capabilities.

- The actual implementation of Items are very similar to what was planned. After further understanding of the mechanism of the program. It is implemented by a new Action - ConsumeItemAction for the player to obtain capability of the items for a new action when it is the player's turn. And the items no longer store the capabilities of what they give.
- Coin class uses the abstract class Utils's method randomIntCoin() which gives the coins that are dropped a random value between 500 to 1 which are divisible by 5.

Requirement 5

Toad Class
Toads primary purpose is to be able to interact with the player. Once the player interacts with toad, it will create a new Trade Action object and display the menu of possible trade options the player has. The player then selects the trade option needed and passes to Toad which will directly use the Trade Class to process this.

Relevant methods used by Toad:
- create new Trade object
- execute(actor, map): executes the selection to the trade class
- playTurn(): shows selection made by player
- menuDescription(): shows the player all the selections.

Trade Class
Trade class is responsible for taking in the selection made by the player through Toad and execute the relevant actions. It is extended from the Actions class and is dependent on the PlayerInventory, Wallet, SuperMushroom, PowerStar, and Wrench class. Trade class will call the getWallet() to show the amount available for trade. It will then subtract the relevant amount from the wallet int for the trade selected by the player. If the wallet < 0, Trade class will output an error message to Toad to display on the menu. If wallet > 0 then trade class will deduce the amount and update the wallet class. It will then call player inventory and create a new instance of the Item that the player wants, then append into the PlayerInventory. Once transaction complete, it will output a success message to Toad to display on the menu.

Relevant methods used by Trade:
- Wallet: getWalletAmount(): shows trade how much money is current in the wallet
- Wallet: removeAmount(): remove the amount of the item from wallet
- creates new instance of SuperMushroom, Wrench or PowerStar
- TransactError(): outputs an error message due to wallet
- SuccessMessage(): outputs a success message
- Player: addItemToInventory(): allows to append the item bought into the PlayerInventory

Player Inventory Class
Player Inventory class is extended from the Actor class which allows it to use the GetInventory() and RemoveInventory() methods. It also has a dependency on the items class to get the ticks() method as some requirements indicate the PowerStar object will disappear from the inventory.

Relevant methods used by PlayerInventory:
- Ticks(): allows to remove the PowerStar after a certain amount of ticks.

Wallet Class

Wallet class's primary goal is to keep track of the amount of coins collected by the player. It will store this as an integer and be able to have a setter which changes the amount depending on Trade Class's use and a getter to show the amount present in the class. Trade class is hence dependent on Wallet Class.

Relevant methods in Wallet:

- GetAmount(): shows the amount of money in the wallet
- removeAmount(): take money away from the wallet due to trade

Wrench, PowerStar and Mushroom Classes

Wrench, Power Star and SuperMushroom class have the interface of ItemPrice. This allows each item to have an int price, so that when called by the Trade Class (dependancy) for its getPrice() method it will return a price for the Trade Class to use.

Relevant methods in Wrench Power and SuperMushroom Class:

- getPrice(): shows the price of the relevant item called

UML Class Diagram:

Changes during implementation:

- Instead of Trade Class depending on the Wallet Class, this is now dealt directly In Player Class using association with the Wallet Class. This allows Trade not to violate the **Single Responsibility Principle** by having wallet subtract and its own prices. This lets Trade Class's only responsibility be to calculate the change for the trade and add the item to the player's inventory.
- Trade Class also calls Player Class's hasCapability() method to check if the player is a in fact mario and not another actor which will allow it to trade.
- PlayerInventory class has not been implemented and instead uses the abstract Actor class's Inventory List which integrates Engine's methods rather than creating our own.
- ItemPrice Interface has not been implemented and instead been added to Trade class's constructor to be used in Toad. This doesn't violate any **DRY principles** as having the same method implemented multiple times in each magical item class would be redundant. Instead, Trade does this for all the items and if need be, more, which shows its flexibility by not having to alter the entire class to implement another item

Requirement 6

Toad Class
Toad class is responsible for creating a new object of the monologue action in order to interact with the player. Player will execute an interaction with toad to which he will execute to the monologue class and output a string on the console.

Relevant methods used by Toad:

- execute(actor, map): Toad will perform the action of Monologue

Monologue Action Class
Monologue class extends the abstract Action class. It has a dependency on the Player, Utils and Wrench classes. Monologue does most of the work in terms of outputting and calling methods to create the sentences that Toad is to speak.

Relevant methods used by Monologue:

- getWeapon() from Player: will return the item as a weapon. This checks if the item that is returned.
- hasCapabilites(): checks if the capability of the player is of PowerStar. This returns a boolean of if the capability exists on the Player.
- Utilis.getNum(): this will call the static class Utils and have it generate a random number from 1-4 which will be returned.

Monologue will proceed to check the weapon in hand and power star active in order to check if the number generated by Utils is subitable to print. If not, it will ask Utils to regenerate. Monologue will then print to toad the relevant string in which Toad will output.

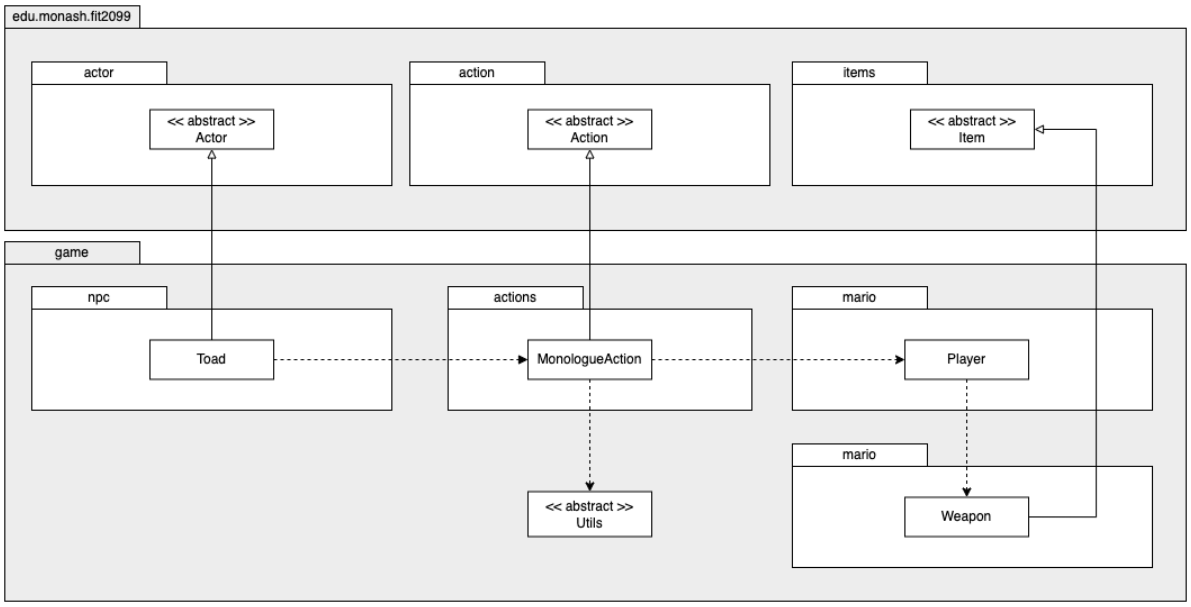
Wrench, PowerStar and Player
These classes exist primarily to use their methods for Monologue class.
Player is called by Monologue to check for its getWeapons() hence dependency. Player also has PowerStar and Wrench as it's attributes.

Power Star and Wrench both extend the abstract items class, since they are an Item held by the Player.

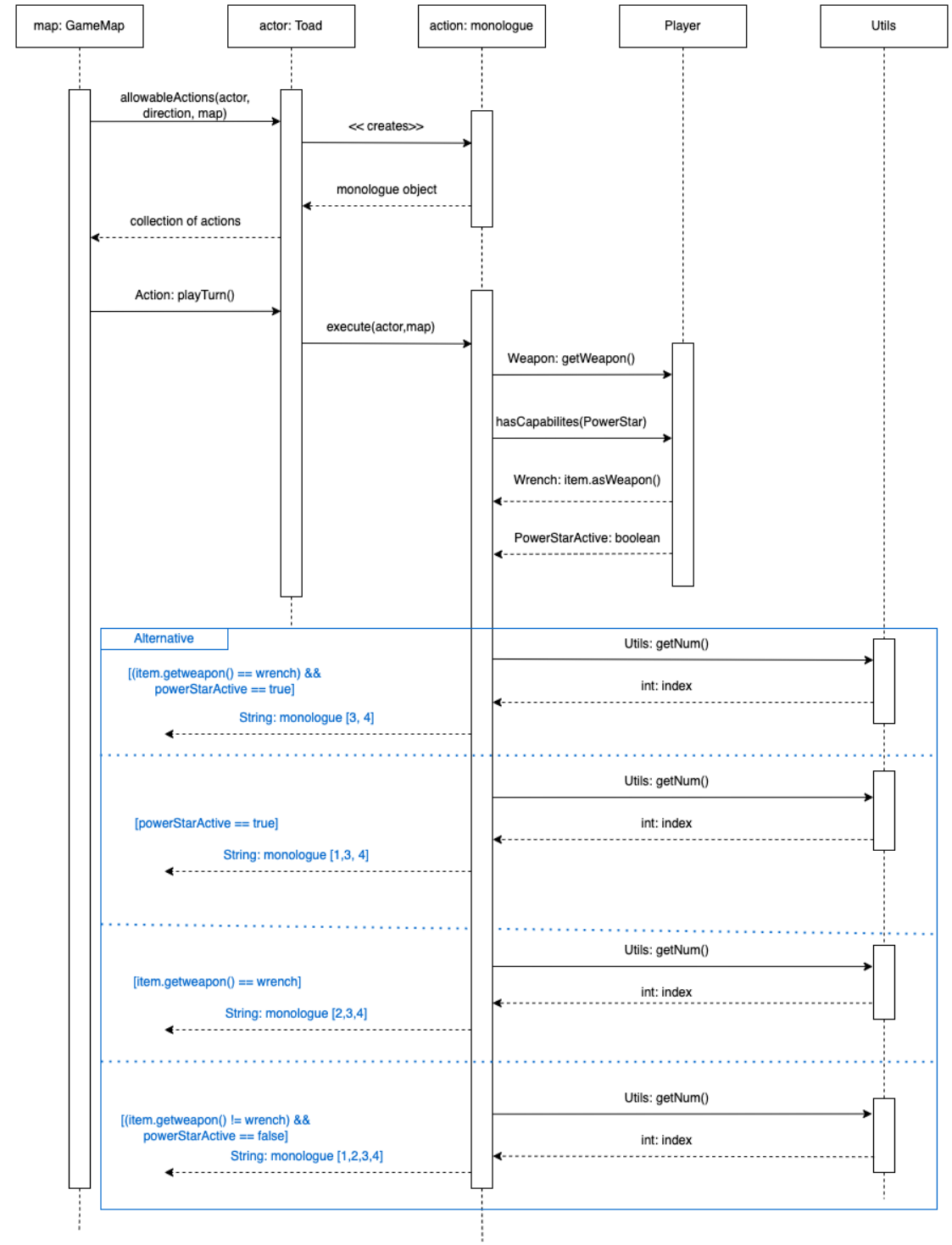
Utils Class
Its primary objective is to return an int from 1-4 for Monologue to randomly output a String.
Relevant methods in Utils:

- getNum(): returns a int of 1 to 4

UML Class Diagram:


edu.monash.fit2099
actor
<< abstract >>
Actor
action
<< abstract >>
Action
items
<< abstract >>
Item
game
npc
Toad
actions
MonologueAction
<< abstract >>
Utils
mario
Player
mario
Weapon

Sequence Diagram:



Changes during implementation:

- Having the Toad Class call the action for MonologueAction in his allowableActions() method lets the code not violate the **Single Responsibility principle** as Toad's only responsibility is to call a new object of that class.
- The association Player class has with PowerStar have not been implemented as it instead uses Player's hasCapability() method directly in MonologueAction to check if the player has the PowerStar active

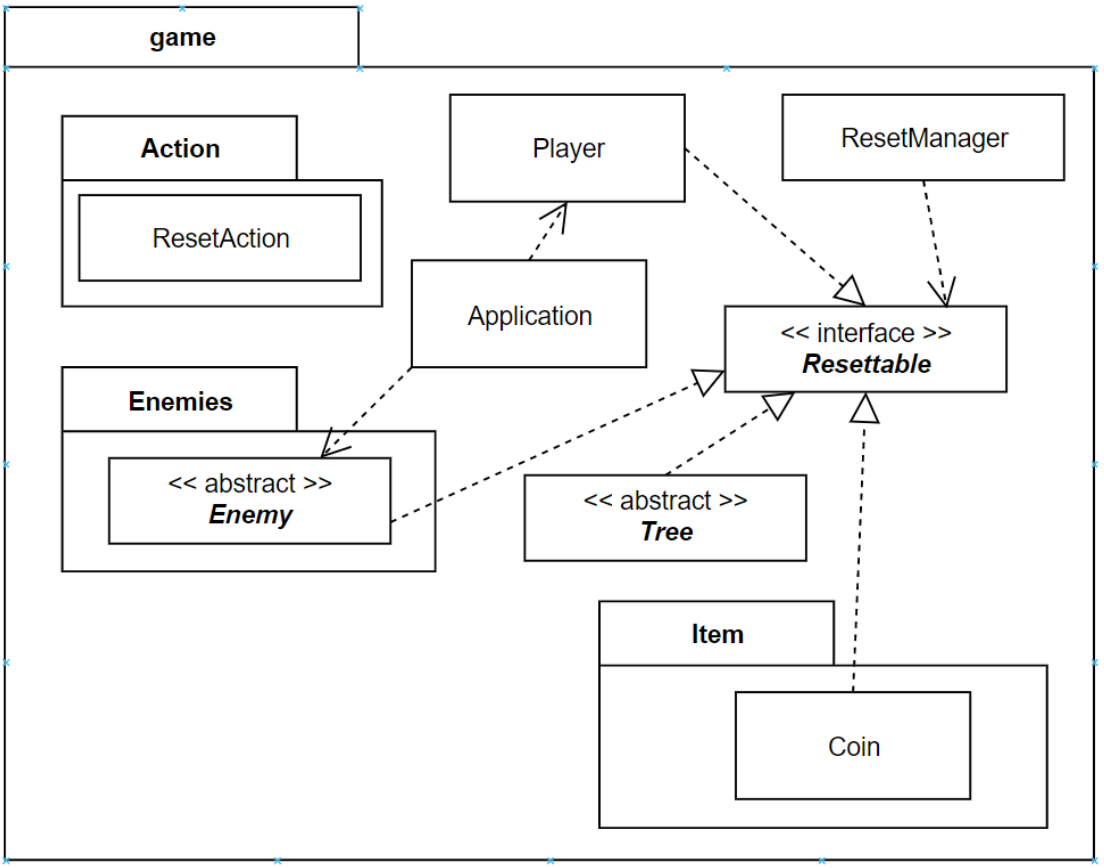
- MonologueAction class does not have a dependency to the Wrench class anymore and instead goes through the Player class to implicitly call the getWeapon() method that returns Wrench object.

Requirement 7

Resetting the game
Resettable will be implemented for processing the reset: To remove the enemies on the map, a method from resettable will reset the registered instance, any enemies class exist on the map and remove all of them.

In the class that implemented Resettable, it will call by the run method in ResetManager which each class should have their own reset code with will not interrupt other class, which achieved **single responsibility principle**.

UML Class Diagram:



Changes during implementation:

- RestManager has been implemented so that is has a dependency to interface Resettable, this helps maintain the **Single Responsibility Principle** as Reset managers only responsible for executing its run method.
- The resettable interface is implements by player, tree, coin and enemy abstract class which helps maintain the **Liskov Substitution Principle** when compared to the original UML diagram
- Application class depends on both player and enemy abstract class due to fact that it wipes the enemy off the map and resets the players HP.