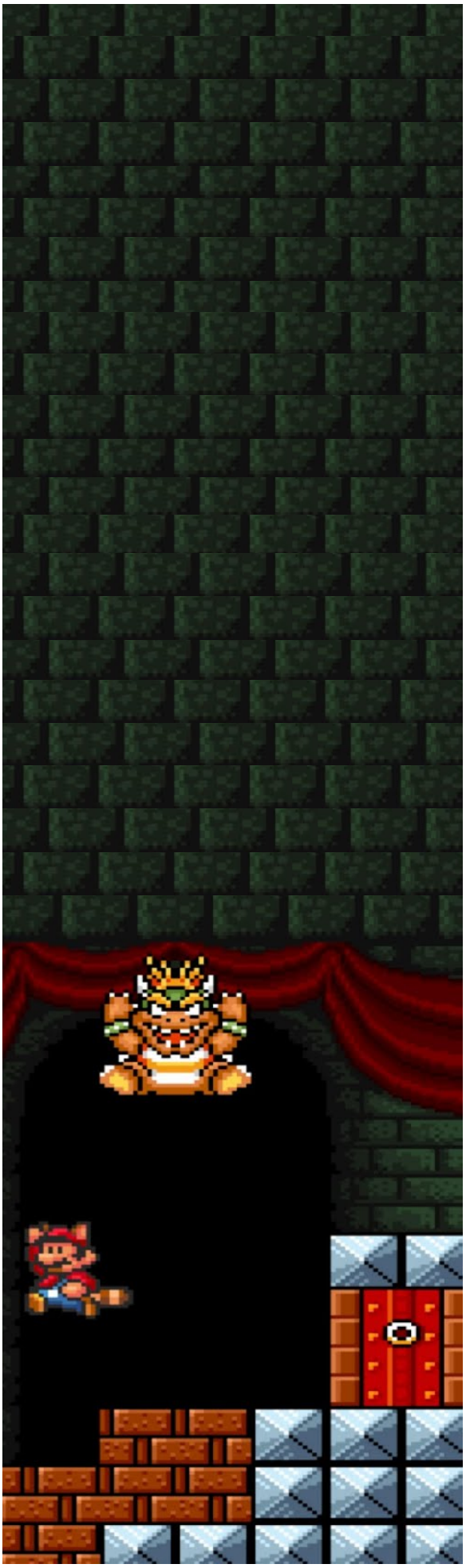


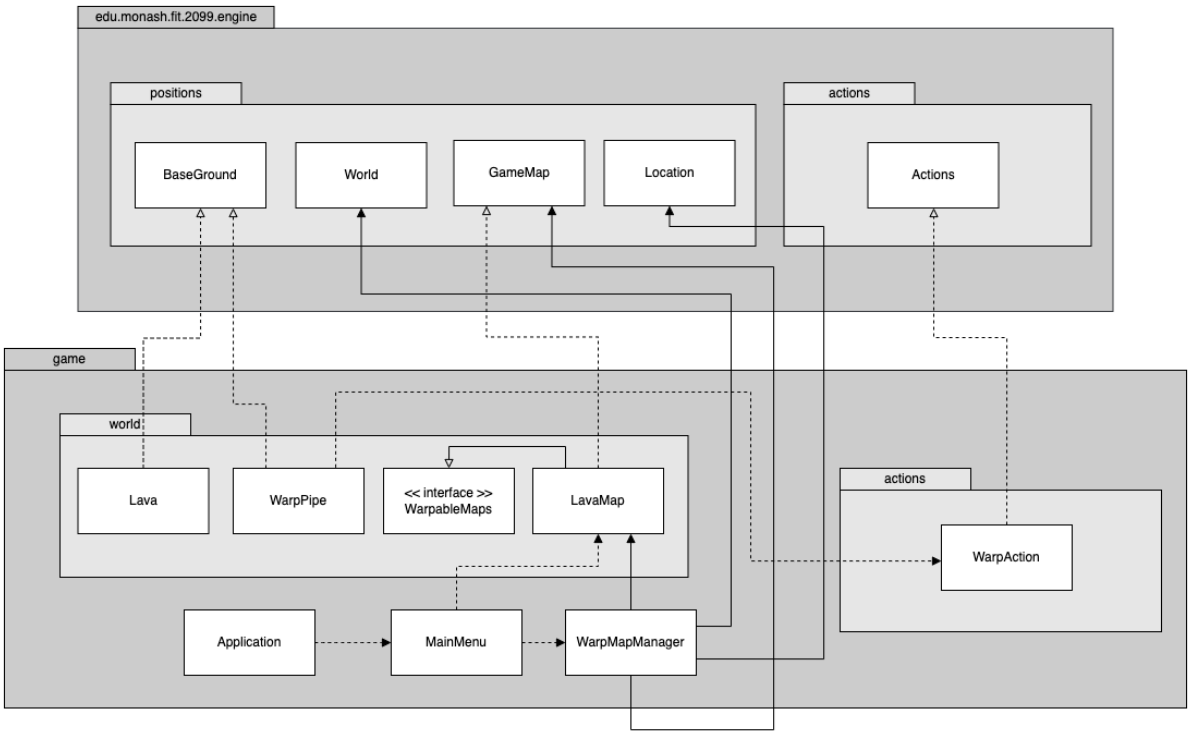
FIT2099 Assignment 3 Design Rationale

MONASH University

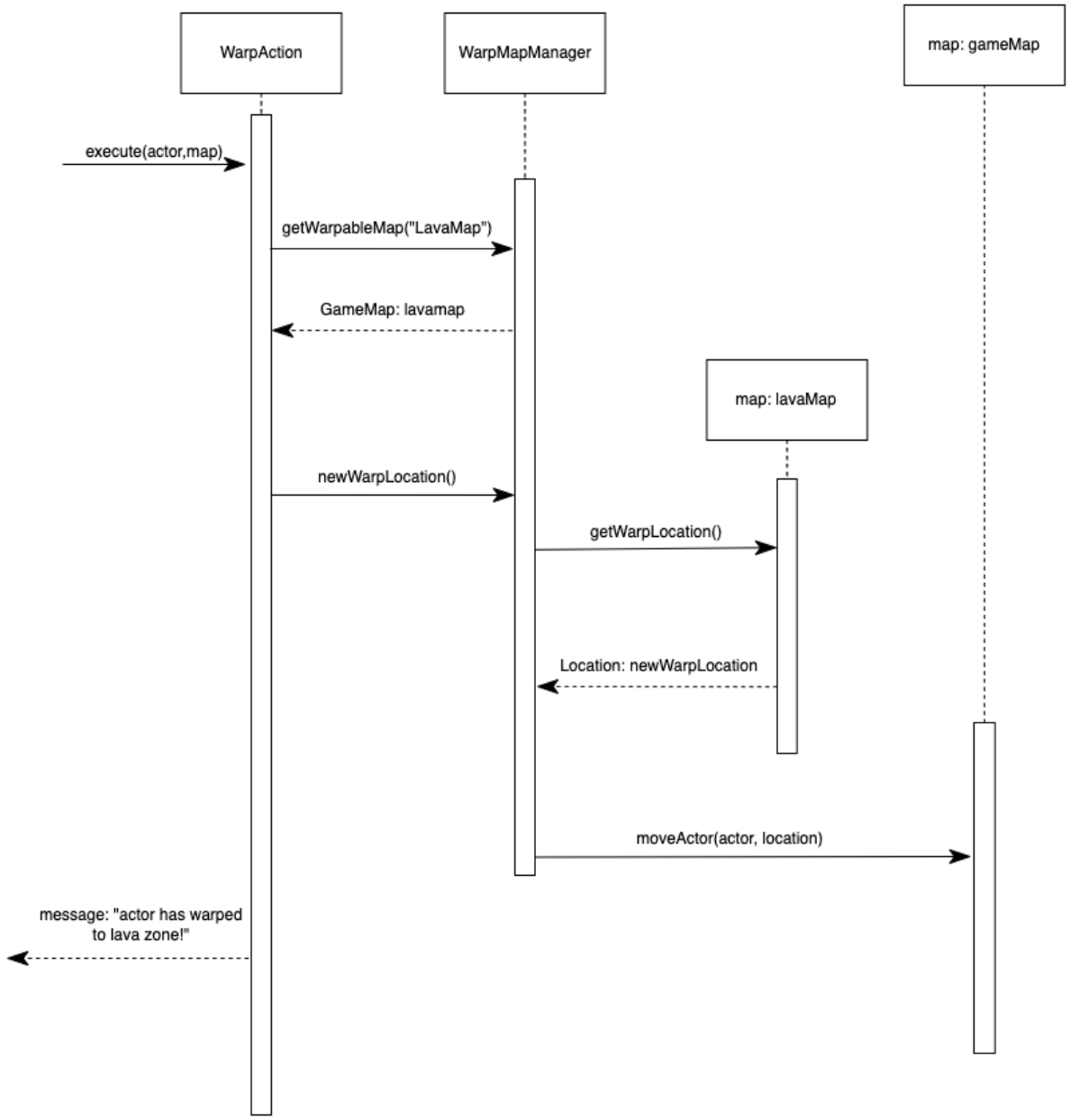
Nethmini Botheju, Yin Lam Lo, Jasper Martin
Lab 14 Team 7

Requirement 1

UML Diagram



Sequence Diagram



1. New Map 🗺️

- WarpMapManager

The implementation of Instance Class WarpMapManager allowed for the game to keep track of all the maps in one place. This class keeps a list attribute of the warpableMaps so that depending on the map needed, it is able to retrieve any. This allows future maps to be implemented if need be by simply using the addWarpMap().
Having WarpMapManager adheres to the **Single Responsibility Principle** as only this class has the ability to manage the maps and create attributes to hold the warp locations needed for other functions of the code.
Making this class static also allows it to be called from any other class without having to create a new instance that may cause the current attributes to overwrite itself

- LavaMap

This class implements the Warpable map interface as it allows this to follow the **DRY principle**. This interface makes it so that multiple warp maps can be created without having to repeat the same code over and over again. It also adheres to the **Liskov Substitution principle** as LavaMap isn't an extension of the main game map but rather its own inheritance of GameMap class with its own interface that allows us to replace main game map when the player warps without disrupting the behaviour of the game since they are two different maps altogether.

2. Teleportation (C) 🌀

- WarpPipe

WarpPipe class adds a new capability to its constructor known as WARPABLE, this allows the game to recognise that this inheritance of BaseGround class lets the user warp to certain ground without throwing a null error. This avoids having to use instanceOf and violating **LIskov Substitution principle.**
Its sole responsibility is to call a warp action and decide if the actor is of player instance to be able to teleport. Which allows this class to also adhere to the **Single Responsibility Principle** since it is only responsible for calling the action class

- WarpAction

WarpAction's only responsibility is to get the warpMap from WarpMapManager and parse it back into another method. This adheres to the **Single Responsibility Principle** as it is only responsible for a single function of the game's overall functionality and nothing more.

3. Current code changes (If any):

- CreateWorld() method

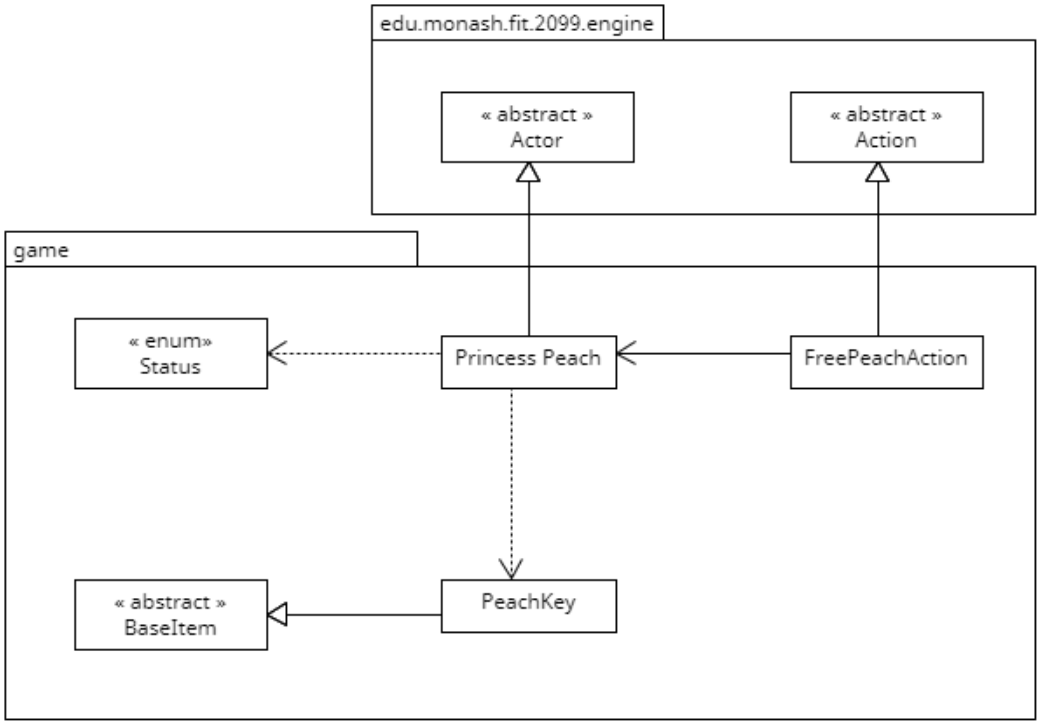
WarpMapManager() creates a new instance of World instead of having it run in the application, this allows for the class to be able to keep the world created as its own private static attribute which means it cannot be modified or accessed elsewhere. Having this makes it possible to add new maps through the class and have specific getters and setters for them to add and to access.

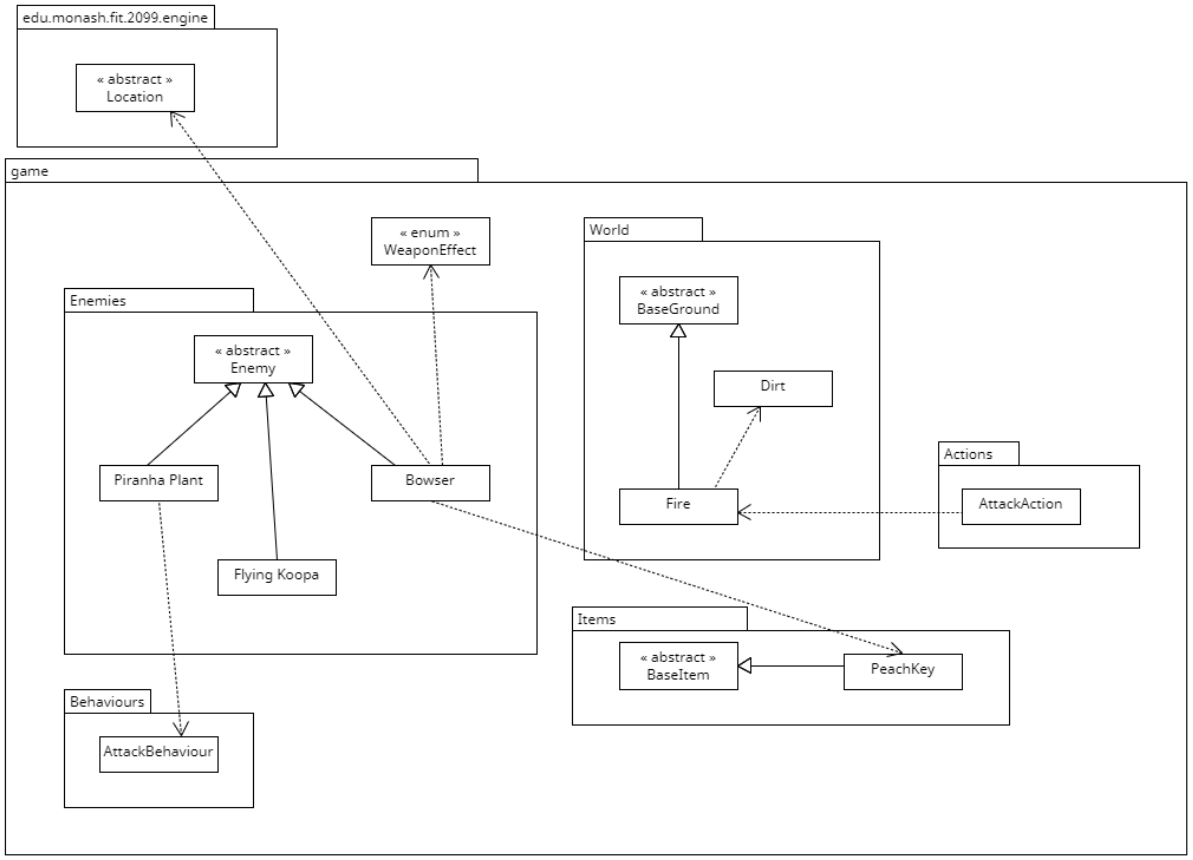
- Tradeable Interface

A tradeable interface was added so that all trading items like PowerStar, SuperMushroom and Wrench can implement it. This allows the code to adhere to the **Single Responsibility principle** as it takes the functionality of calculating the trade out of TradeAction making it easier to change the code and not alter the whole thing. Having this calculateTrade() method in all the items would be problematic hence having it as a single interface shows the code agrees to the **Don't Repeat Yourself principle**.

Requirement 2

UML Diagram





1. Princess Peach (P) 👸🍑

- Princess

Princess Peach is an actor that does nothing unless the player has the Key in their inventory, at which point she allows them to talk to her and finish the game. Freeing her is done using the FreePeachAction. Separating these things adheres to the **Single Responsibility Principle**, as the classes each have their own responsibilities, and none are infringing on each other.
The key that drops from Bowser and is used to free Peach is meant to be denoted by 'k', however I changed this to '¬' as I think it looks better.

2. Bowser (B) 🐢😈

- Bowser

Bowser is implemented using the pre existing Enemy parent. The special attributes of it are done within the Bowser class, such as implementing a spawn point for it, as when the game resets he goes back to the spawn point. Bowser is extended specifically from Enemy and not from Actor so that it can be made into a hostile actor using the Enemy classes behaviours rather than actions. Having this allows Bowser to adhere to the **Single Responsibility Principle** as his own functionality is to attack and follow which means any alterations to him will not affect the remaining part of the code. It also means there are no effects to the remaining enemies.

- Fire

In addition a new enum was added called WeaponEffect as Bowser's attack leaves fire on the floor. This enum is used to show if a weapon has an extra effect when it is used. This was done so that any weapon can have an effect, not just Bowser's.

3. Piranha Plant (Y) 🐟🥀

- Piranha

The Piranha Plant again extends Enemy. The only special attribute of the plant is that it cannot move. This was done by overriding the method that allows enemies to follow and attack the player, so that the plant can only attack, not follow. This follows **DRY** principles as the code from other enemies is not redundantly repeated, as well as the **open-closed principle** as this special attribute can be implemented within its own class without having to change the way in which all enemies work.

4. Flying Koopa (F) 💸🐢

- FlyingKoopa

The flying Koopa is very similar to the normal Koopa, however it does not extend the Koopa as that would violate the **Liskov Substitution Principle**. Having FlyingKoopa extend straight from Koopa would mean that if any methods from koopa were changed, it would

affect FlyingKoopa despite being two different actors. Instead FlyingKoopa is now a sibling of Koopa instead of a child of it so that it can have its own methods, like Flying which was implemented using a status, which is checked by the jumpable ground class. It also means that when it is spawned, we can be sure it is a FlyingKoopa and won't be confused for a regular Koopa.

5. Current code changes (If any):

- AttackAction()

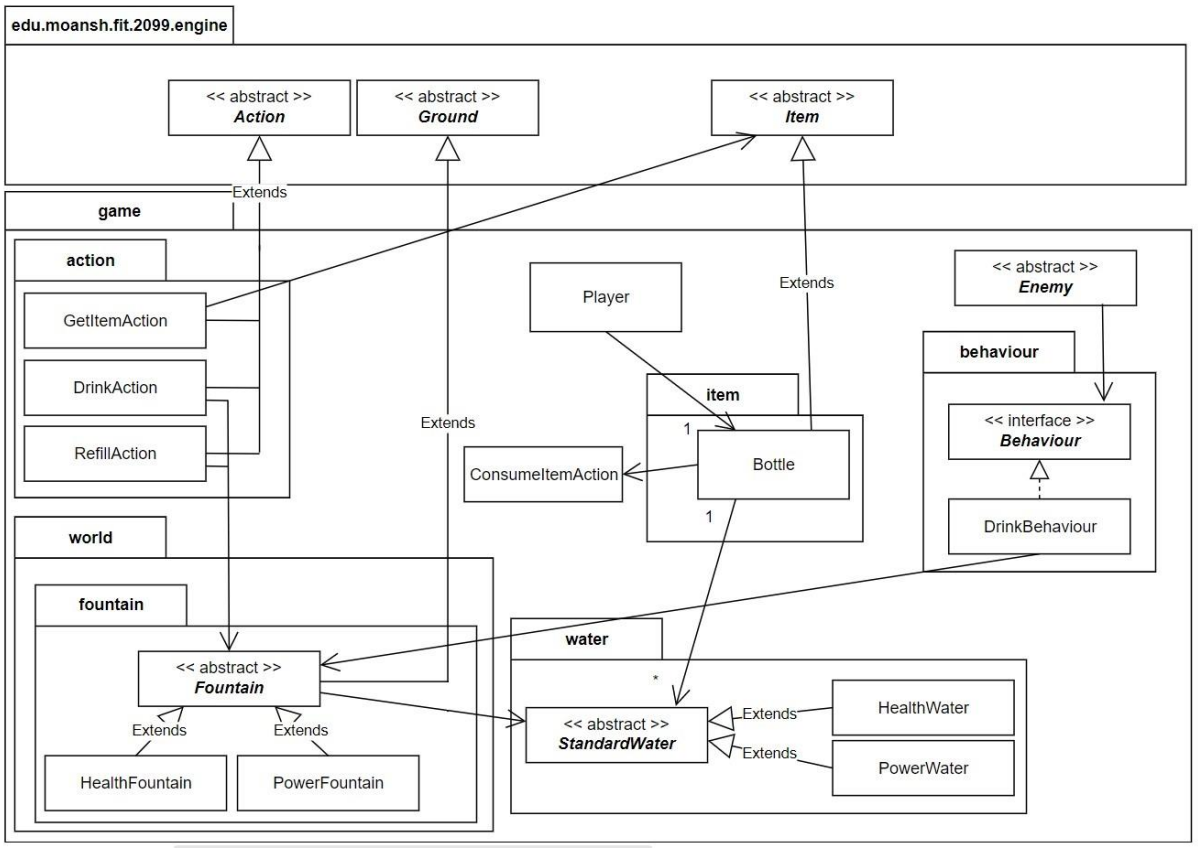
Added special case in AttackAction to allow for special weapon abilities. Specifically added the functionality that means that when Bowser attacks, he leaves fire on the ground. This was needed as otherwise the fire would be put on the ground using Bowsers class, which violates the single-responsibility principle.

- JumpableGround()

Changed logic for jumping and jumpable ground types to allow for flying actor types. This was needed as there was no check for something that was able to go over walls and other high grounds previously.

Requirement 3

UML Diagram



1. Bottle (b) 🥛

- extends from BaseItem implements Consumable

A new class Bottle extended from BaseItem which implements Consumable. It stores StandardWater by stack to achieve first in first out. Each tick it will detect does the bottle contain anything and add action to the player's action list. When a player drinks from the bottle (calls method consumeBy), it will pop the latest water from the bottle stack.

- StandardWater

The brand new abstract class StandardWater will be the base of different water types which stores its name. It is abstract since there will not be any clear water in the game. The status it holds if drinking water will give players any status (for Req5, more on that later). It will be stored inside the stack of the Bottle. Different water types extend from StandardWater which stores its own information about itself to achieve **polymorphism**, different water types have their own class.

Extends from StandardWater: **HealthWater, PowerWater** (More water type at Req5)

2. Fountains ⛲

- extends from BaseGround

New abstract Fountain class will be the base of different types of fountain, and stores the mechanism for water limited serves and replenishes after 5 turns when emptied. It is abstract since a fountain must water with some effects, it will not exist in the game. Different fountain types extend from Fountain which store its own information about itself to achieve **polymorphism**, different fountain types have their own class.
Extends from Fountain: **HealthFountain**, **PowerFountain**, **MiracleFountain** (for Req5)

- RefillAction

RefillAction (extends from Action) allows the player to get water from the fountain and add to the bottle standing on top of it only. It uses which fountain the player is standing on and determines what type of water that fountain has.

3. Optional Challenges Implementations

- DrinkBehaviour

DrinkBehaviour (extends from Action implements Behaviour) determines should enemies drink from the fountain or not, it checks the Fountain.drinkable() and only drinks from it by a chance to keep the game balanced.

- DrinkAction

DrinkAction (extends from Action) allows the enemies to drink water from the fountain while being next to it.

- GetItemAction

GetItemAction (extends from Action) allows the player to obtain an item for free, in this case a bottle. Which should not use TradeAction. Since GetItemAction code could give player items, TradeAction new extends from GetItemAction to follow **Don't Repeat Yourself** principle.

- Limited serves for Fountain

Further code is added to Fountain class so every type of fountain could have the same serving system. And following **Single Responsibility Principle** (SRP) while keeping the mechanism in Fountain only. Each refill and drink will take one serve from the fountain.

4. Current code changes (If any):

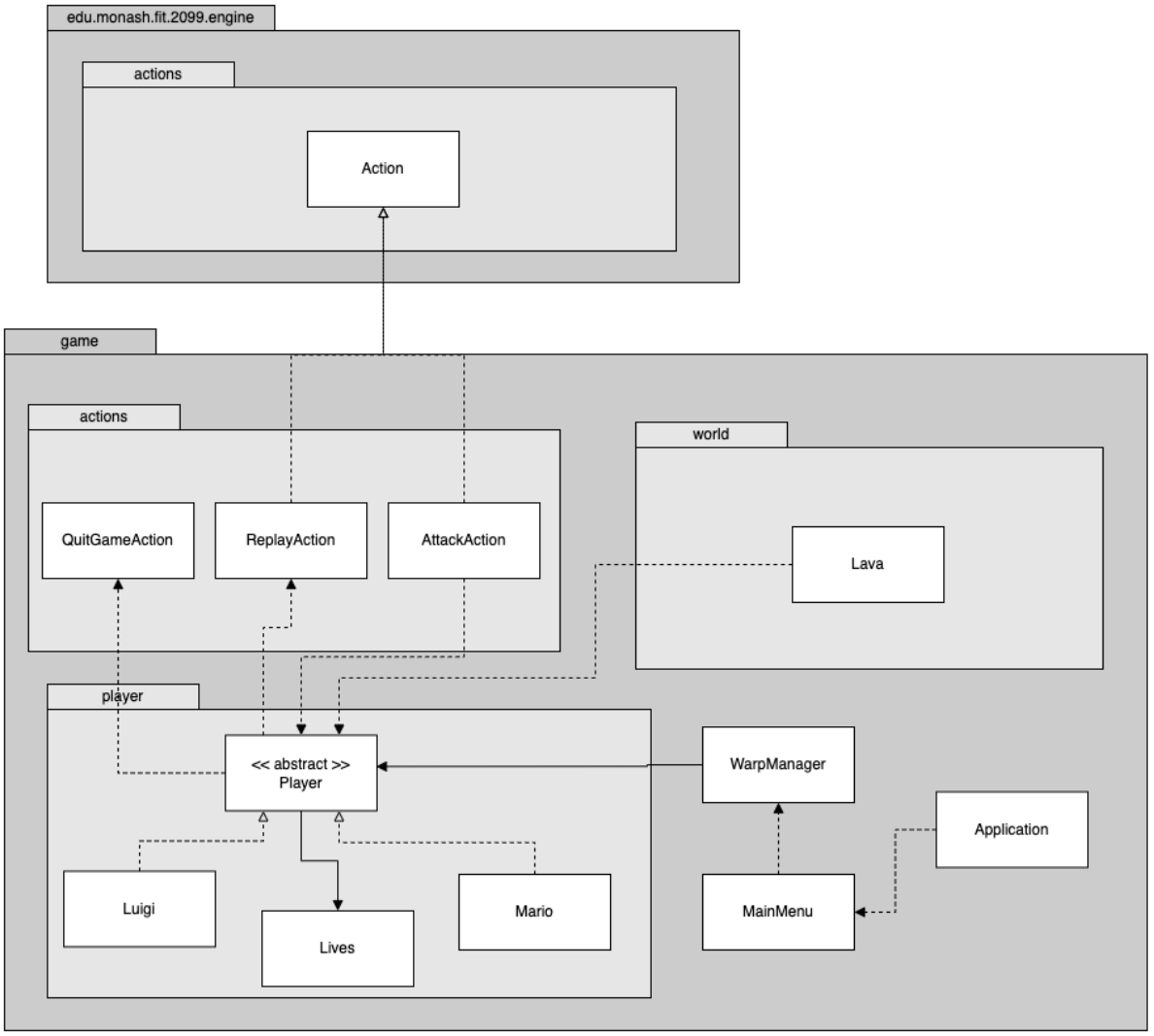
- Consumable

New interface that applies method comsumeBy(Player). Refactored the item that is only consumable: Mushroom, PowerStar, Bottle. Before refactored the ConsumeItemAction would allow any item to be consumed. With the Consumable interface implemented, ConsumeItemAction would only accept Consumable to limit what kind of Item the player could consume. With Consumable interface, we achieved **Interface Segregation Principle**

(ISP), which uses interface to apply features, and allows different consumables to have different behaviour when consumed.

Requirement 4

UML Diagram



1. Stayin' Alive 🕹️❤️

- ReplayAction

This class is the main one that is used to run the ability for the player to respawn once killed. It follows a **Single Responsibility principle** by only having to set the lives the player loses and call it reset manager afterwards. Replay action also checks if the actor using it is either of final boss or player enum constant. This allows both Bowser and Player to be put back into their spawn points when user chooses to replay the game

- Lives

Lives class exists to allow the Player class to adhere to the **Single Responsibility principle.** The methods were originally placed in Player class, but was later moved into a

Lives class that implements the same functionality but doesn't violate the SRP rule as player's only job is to create a player object and not calculate the lives.
Lives Class's only responsibility is to keep track of the lives the player has using its loseLife() and checksLives() methods and to set up the display for the player's status.

2. The Brothers 🔴🟢

- Mario and Luigi

The Mario and Luigi class are extending from an Abstract class known as Player. This allows the classes to follow the **Open-Closed method** because the code for player can be extended without having to modify the actual Player class itself which holds all the vital methods.
This would benefit in the future if the game were to have more players added with their own unique traits then an Interface can be made to further follow this principle.

- IntrinsicWeapon()

Mario class has an override IntrinsicWeapon feature added to him which allows him to deal more damage than the standard. This was mentioned in the design for Requirement 4 as both characters have their own unique traits. Luigi's is the ability to have more hp and more lives while mario, despite being weaker in hp, is able to deal more damage intrinsically and with weapons. This idea adheres to the **Liskov Substitution Principle** because mario is able to have his own separate ability from luigi and changing this will not affect Luigi or the Player class that it was extended from.

3. Current code changes (If any):

- MainMenu

The application class does not run the game anymore. Instead it simply calls a new instance of the MainMenu class which has the responsibility of creating the main menu Input/Output display and calling WarpMapManager to create and store the maps using the input from the user of which player is chosen.
This adheres to the **Single Responsibility Principle** as Application is not solely responsible for running the whole game but is now only responsible for calling a class that has two methods to do small parts of it.

- Spawn Point

Both Player class and Bowser class have added a spawn point attribute to themselves to keep a copy of where they were originally added to the map as they are both wandering actors. Since the game consists of a ReplayAction that goes a step beyond resetting the map by actually restarting the whole game, having this Location saved for both actors allows them to move back to their original location when the user chooses to replay the game.

- hitPoints

The player class has an added int attribute called hitPoints that keeps track of the hitpoints that's been parsed for the Actor superclass. Because Luigi and Mario have different hp, during the replay game, having this getter allows resetting to maxhp for each player

separately. Since in the previous implementation, it was a hardcoded 100 integer for the resetMaxHp() which forced luigi's hp to be 100 too.

- Lava

Lava class has an added feature in its tick() method that checks if the player is conscious every time it inflicts damage. This uses the Lives classes checkLiives() method and immediately parses a boolean to the player's playturn method. Having this implemented allows the game to end in the turn that the player is unconscious rather than to wait another turn before executing. This helps the overall functionality of the game run smoother and reduce an unnecessary plays the game takes otherwise

- AttackAction

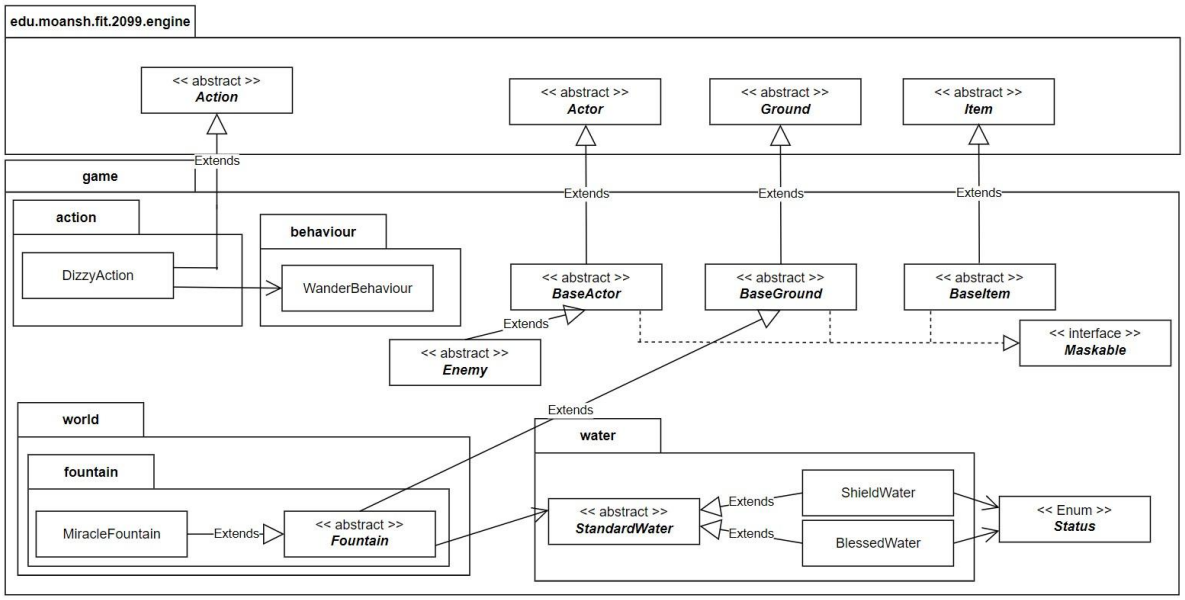
Similar to the Lava class, attackAction also uses the checksLives() method to see if the inflicted damage has rendered the player unconscious. The overall functionality is the same as the Lava class and is implemented to reduce the playturn redundancy.

- Bowser

Aside from the player itself, Bowser is the only other actor that reappears when the game starts. As Bowser follows the player around hence should go back to his original position when the player chooses to replay the game. This is why Bowser has an added enum constant called FINAL_BOSS that allows the ReplayAction to account for his reset to default position as well.

Requirement 5

UML Diagram



1. Fog of war 🌫️

- Maskable

Classes that are printable and maskable will call masking from Maskable to determine should it show the symbol on print out. masking() is a static method to call so all the masking mechanism will be kept inside Maskable to achieve **Single Responsibility Principle** (SRP) and implementing as an Interface could achieve **Interface Segregation Principle** (ISP), which uses interface to apply features, in this case different printables. Since we could not edit the engine code, a new Base class must be created to implement Maskable. Also the Fog of war effect can be made by a player having Status.BLIND and not hardcoded to be more flexible. Any new changes to the map will show for a turn and it will be masked. This gives the player a sense of surroundings but with unclear information.

2. Miracle Fountain 🚰✨

- MiracleFountain

The third type of fountain is MiracleFountain (extends fountain) which randomly changes between BlessedWater and ShieldWater. The serving mechanism is extended from its parent class.

- New Types of water

The below types of water are given according to their child class. And any new type of fountain is given according to their child class. This **Open-Close Principle** (OCP) allows

any closed modification in the parent class(Fountain and StandardWater), while open for extension to add more child classes for more features.

- ShieldWater

ShieldWater (extends from StandardWater) gives Status.SHIELD, the actor with this capability will have 90% chance to take no damage for a turn and loses the capability.

- BlessedWater

BlessedWater (extends from StandardWater) gIves Status.BLESSED, the actor with this capability will receive 50% less damage in 5 turns.

StandardWater and Fountain is an abstract concept and its subclass are the details. This achieved the **Dependency Inversion Principle** (DIP) since they are an abstract concept that will not be implemented and the details are in their subclasses. That achieved the **Liskov Substitution Principle** (LSP) since any subtype of them could be accepted even that are subclasses.

3. Lingering Effect 🔥😵

- Status.BURNT

The player will receive Status.BURNT and the longer he stays on Lava the longer the Burnt will keep after exiting the Lava area. The damage will be higher the longer the player stays on lava. The turn will tick down if the player leaves the lava area.

- Status.DIZZY

There will be a random chance that enemies will deal critical hits to players and apply Status.DIZZY. The only action that the player can do while having this status is DizzyAction.

- DizzyAction

DizzyAction (extends from Action) is the only action that the player can do when dizzy. Player could be awake again by a random chance, if he could not awake successfully, it will get a random movement from WanderBehaviour to retrieve a random direction, since Dizzy are originally planned to have random movement, making use of the current code could achieve **Don't Repeat Yourself** principle.

4. Current code changes (If any):

- New BaseCode for unifying code for **Don't Repeat Yourself** principle

Many of the printable elements have to adapt to the new fog of war effect, and are not allowed to change the base code, so a new base for each type of item is implemented. BaseActor extends from Actor, BaseGround extends from Ground, BaseItem extends from Item, BaseWeapon extends from WeaponItem, which all implements the new Maskable interface. All Base series classes will have a new attribute defaultChar for reversing the item when in range.

- Keep code unified

All random action uses Utils.randomChance() to unify and keep it clean. Some classes and methods that are long from previous assignment are refactored to get rid of code smells and for a better design.