FIT2099 A3 Work Breakdown Agreement

Team Number	Lab 16 Team 2
Names	1. Garret Yong Shern Min 31862616 2. Jastej Singh Gill 31107974 3. Low Lup Hoong 31167934
Date	21/5/2022

Our team will break down our work by assigning tasks to each member. For each requirement, the assigned member will be responsible for creating the respective UML Class Diagram and Design Rationale.

Task	Member	Date of completion
Requirement 1	Low Lup Hoong	18/5/2022
Requirement 2	Garret Yong Shern Min, Low Lup Hoong	18/5/2022
Requirement 3	Jastej Singh Gill	18/5/2022
Creative Requirement 1	Garret Yong Shern Min	20/5/2022
Creative Requirement 2	Jastej Singh Gill	19/5/2022
Reviewing	All members	20/5/2022

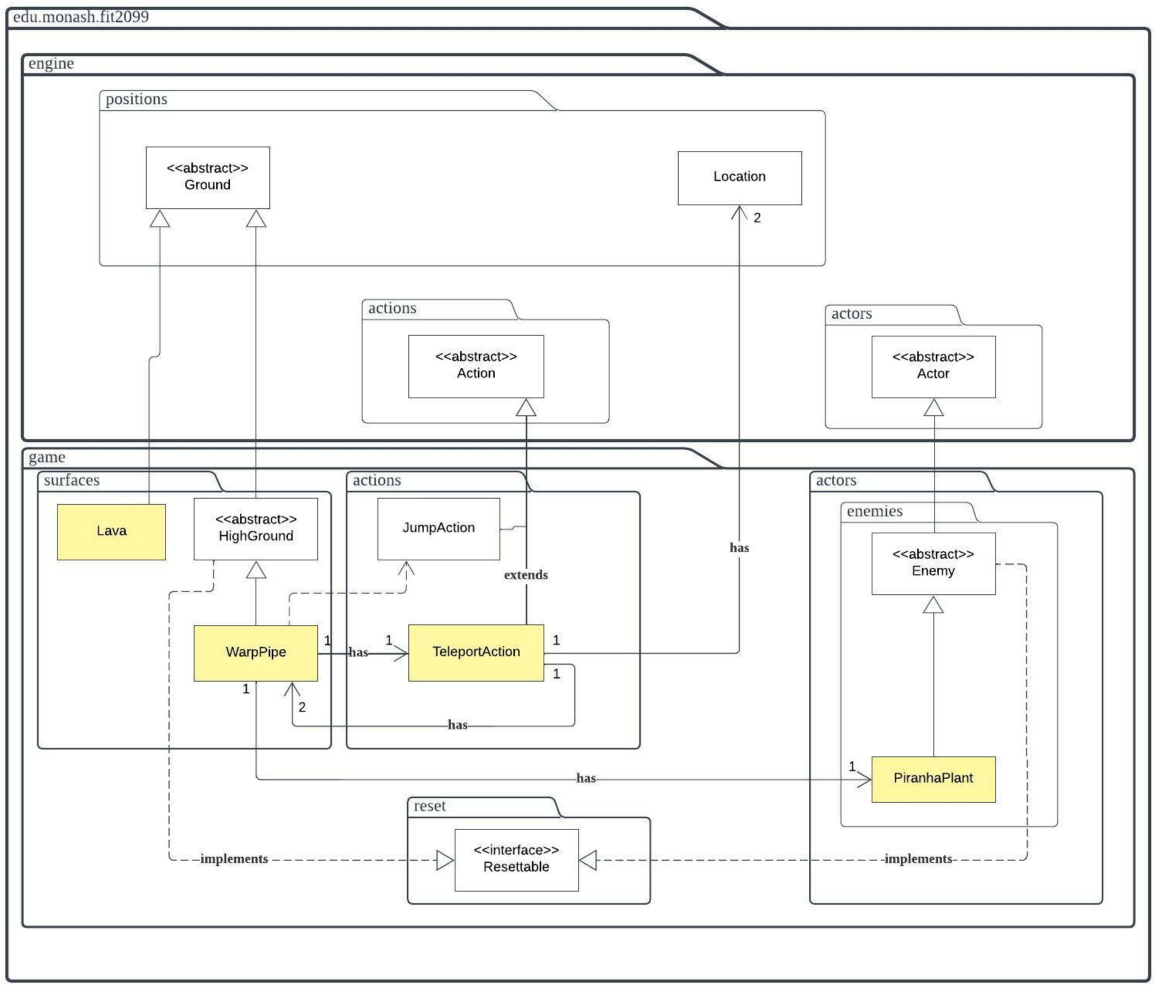
FIT 2099 Object-Oriented Design and Implementation

Title	FIT2099_S1_2022 **Assignment 3: Further Design and Implementation**
Date	21/5/2022, Saturday
Team Number	Lab16Team2
Team	1. Garret Yong Shern Min 31862616 2. Jastej Singh Gill 31107974 3. Low Lup Hoong 31167934

Fixed Requirements

Requirement 1

UML Diagram



Design Rationale

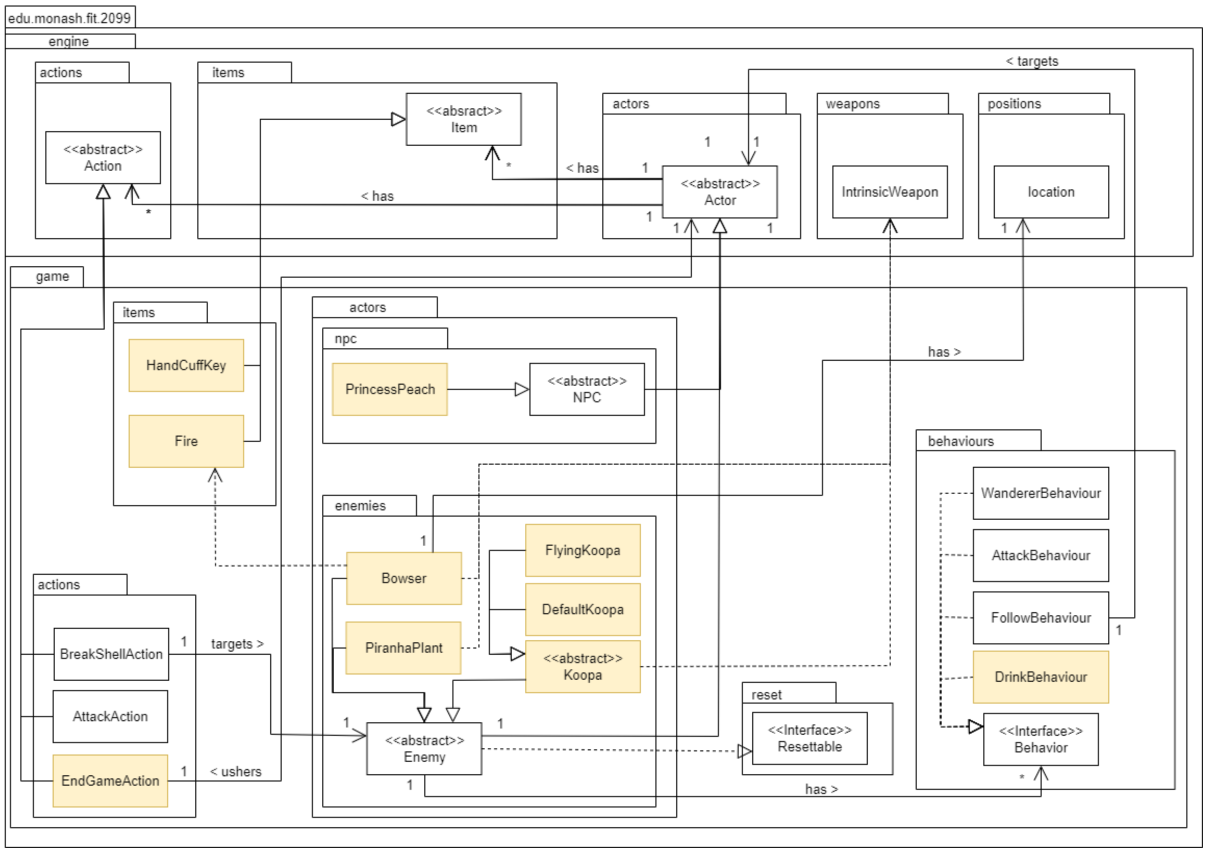
To ensure we adhere to DRY, Do Not Repeat Yourself principle, we made sure to utilise abstract classes from existing engine code and created in our Assignment 2. Lava class extends Ground class which has an integer attribute, *damage* to reduce magic numbers in our code. WarpPipe extends High Ground abstract class we created previously to inherit relevant attributes and behaviours such as ensuring only actors that have CAN_JUMP_ONTO_HIGH_GROUND status can jump onto it.

To practice good encapsulation, each WarpPipe will have one Piranha Plant and TeleportAction as its attribute. One WarpPipe will have one TeleportAction to ensure that whenever the Player teleports, it sets the destination WarpPipe's destination to where it came from. TeleportAction extends the engine code's Action abstract class to reduce code repetition.

TeleportAction has its own class instead of embedding it as WarpPipe’s method as we want to adhere to Single Responsibility Principle. This ensures that one class has only one responsibility. By creating a new class for TeleportAction, we also adhere to Open to Extension, Closed for Modification. We could easily reuse TeleportAction if we were to implement more grounds who supports teleportation action.

Requirement 2

UML Diagram



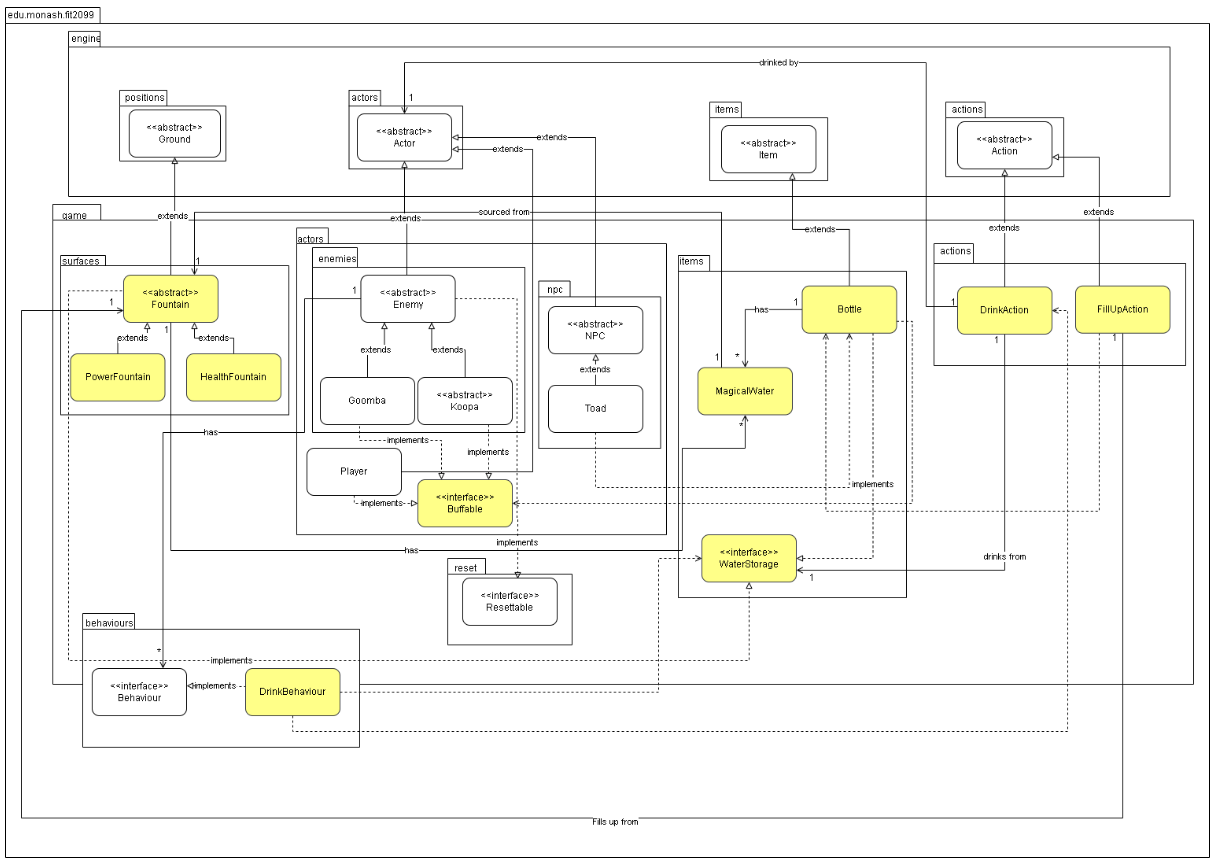
Design Rationale

When completing this tasks, we ensured that the DRY principle was adhered to by reimplementing several classes designed during Assignment 2. This includes the Enemy abstract class and the NPC abstract class which we utilise as all new child classes share similar attributes and methods as those abstract classes.

This also brings up the consideration into Liskov's Substitution Principle (LSP). Whereby all the Child classes would follow the constraints of their respective parent classes and they would be easily substituted with one another. This was demonstrated most prominently with our newly converted Koopa abstract class. Initially the Koopa class was a concrete class which was mean to represent the default Koopa, but due to the introduction of the FlyingKoopa class the Koopa class was converted into an abstract class and DefaultKoopa (class that newly represents the default Koopa introduced in Assignment 2) and FlyingKoopa would inherit this abstract class. This was done in tandem to adhere the to DRY principle and since DefaultKoopa and FlyingKoopa added no new, unexpected constraints, it would be possible to swap out the child classes for the parent classes and this adheres to LSP.

In addition to that, AttackBehaviour and AttackAction has undergone some changes. By adding an additional constructor, we would enable for easy to be implemented but modification would not be required. The new constructors would accept a special effect; for instance Bowser would pass a new instance of Fire item into the constructor of his AttackBehaviour to enable him to drop fire when attacking. However, a Goomba could leave this parameter empty to indicate that this Koopa has no special effect to leave behind when attacking. In short, by making this small change we reduce the need for maintenance and modification while increasing the robustness of the code.

Requirement 3
UML Diagram



Design Rationale
A new Fountain abstract class has been created to reduce repeated code in the HealthFountain and MagicalFountain class to adhere to the Do Not Repeat Yourself principle. Method such as drinkedFrom and filledUpFrom are defined and implemented in the Fountain class instead of repeating it in the HealthFountain and PowerFountain classes.

The Fountain abstract class also adheres to Dependency Inversion Principle as the concrete classes HealthFountain and PowerFountain will not have an association with the other concrete classes MagicalWater and FillUpAction.

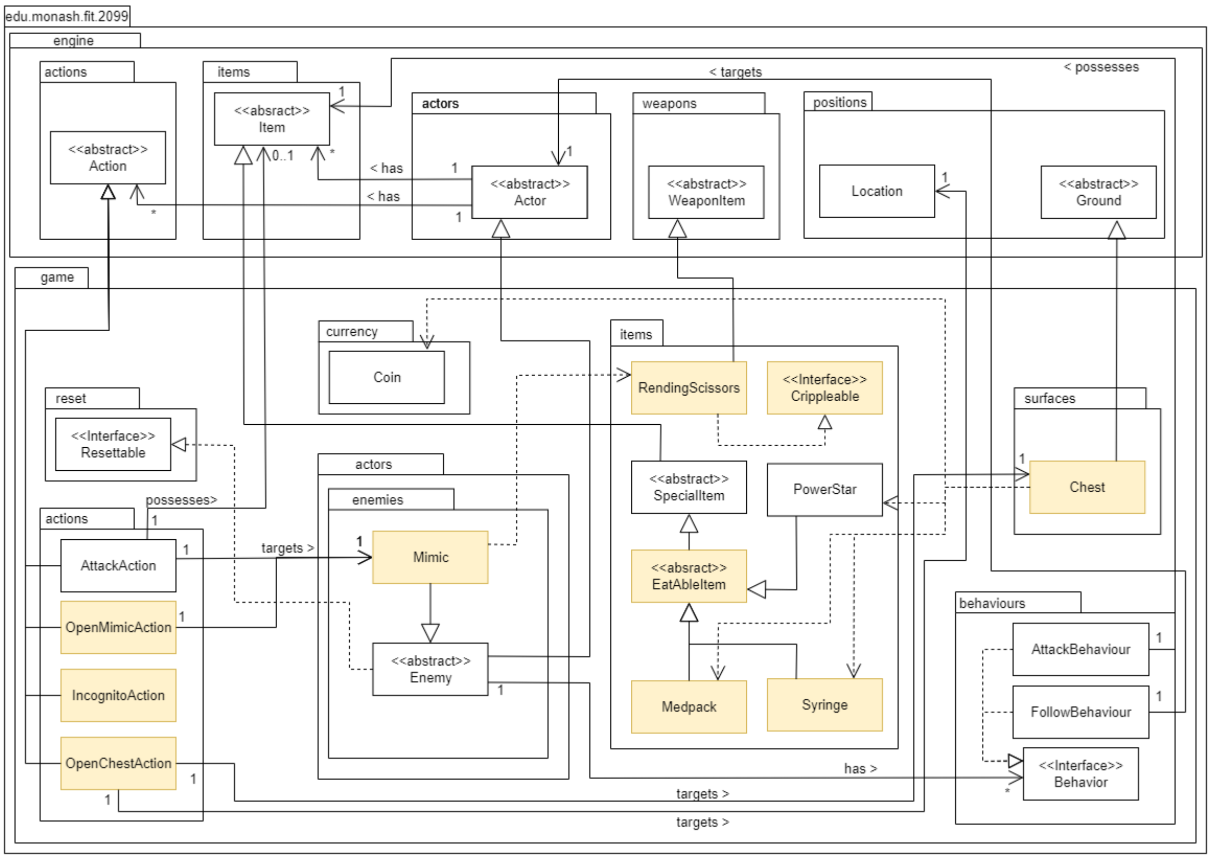
Since, Bottle and Fountain both behave similarly as both can be drinked from and store MagicalWater, they implement a WaterStorage interface to ensure they both behave similarly and adhere to the dependency inversion principle as the DrinkAction class no longer has an association with Fountain and Bottle.

The MagicalWater class does not extend the Item class in order to adhere to Liskov's Substitution Principle. MagicalWater behaves completely differently from a regular item as it can't be dropped to ground and can only be stored in WaterStorage. If MagicalWater did extend Item a lot of methods would have do-nothing implementations and even attributes like displayChar would be redundant as the item wouldn't be on the Ground.

Creative Requirements

Creative Requirement 1

UML Diagram



Description

This new feature adds numerous additional gameplay features to make the game more fun and continuous. The new extension introduces new interactables, weapons, combat features and many more.

The following lists the additions contributed in this new feature:

- Healing items
  - Medpack
    - This item heals its consumer by 70 hitpoints
    - Can be bought from Toad for $350
  - Syringe
    - This item heals its consumer by 10 hitpoints over the course of 10 turns
    - Can be bought from Toad for $300

- New Weapons and Effects
  - Rending Scissors
    - This weapon will only drop from mimics and has a chance to cripple an enemy
  - Cripple
    - This is a new status which will rob a character's mobility

Chests
  - Regular chest
    - Will drop a random amount of coins, Medpack, Syringe or a Power Star
  - Mimic
    - Will attack the player and has a very high amount of health but will drop a very valuable item (the new Rending Scissors)

Requirements

Must use least two (2) classes from the engine package	A new class called Chest will be created which will inherit the Ground class. Mimic class will extend the Enemy class which extends the Actor class Both healing items, Syringe and Medpack classes extends the EatAbleItem which extends SpecialItem which extends Item There are also the two action classes, OpenChestAction and OpenMimicAction, which both extend the Action classes
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	Mimic class will extend the Enemy class created during Assignment 2. EatAbleItem extends the SpecialItem created during Assignment 2
Must use existing or create new abstractions (eg., abstract or interface, apart from the engine code)	Mimic class would extend the Enemy abstract class The RendingScissor class implements the Crippleable interface and Crippleable is a new interface created to add methods for weapons capable of crippling
Must use existing or create new capabilities	New capabilities such as CRIPPLE_ATTACK, CRIPPLED, HEAL_OVER_TIME, CHEST_CLOSED were added to facilitate certain activities across the other features

Design Rationale

When completing this tasks, we ensured that the DRY principle was adhered to by reimplementing several classes designed during Assignment 2. This includes the Enemy abstract class and the EatAbleItem abstract class which we utilise as all new child classes share similar attributes and methods as those abstract classes.

While keeping the Single Responsibility Principle (SRP) in mind, we made adjustments to ensure there we no god classes in the program and each class would have one reason to change. For instance, there would be separate classes when opening a chest and a mimic. This is to reduce the usage of the instanceof command. In addition, while the OpenChestAction opens the chest, it is not responsible for the dropping of items on the floor as this would obviously be the responsibility of the Chest class itself and if left for the OpenChestAction to implement then it would violate the SRP. Besides that, another example of this would be the crippling of an enemy; in this case, when the enemy is attacked with a weapon that can cripple, the AttackAction would be responsible only for passing the Crippled status enum to the target's capability list but the actual logic for the crippling would be located within the Enemy class. In addition of adding it into the Enemy class, we are also adhering to the DRY principles as all enemies should handle the crippling the same.

As for the EatAbleItems class, it is a class that is open for extension but closed for modification. This adheres to the Open-Closed Principle as this class allows for new items such as Syringe and Medpack to extend but there was no need for modification.

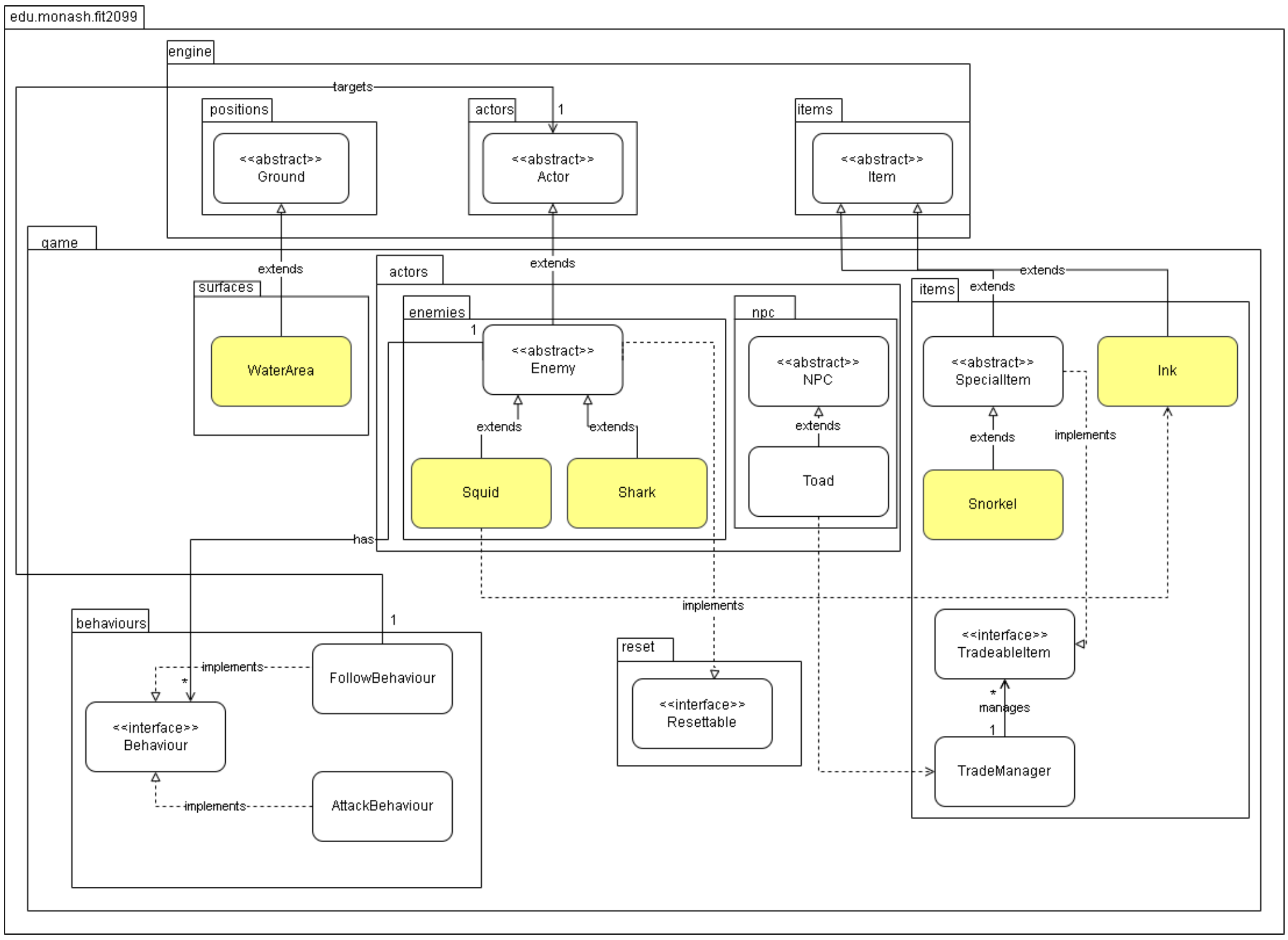
All classes such as Mimic, CrippleAttackAction, Chest and Healing Items: Syringe, MedPack extend their respective parent classes but all share the same constraints as the superclass and do not possess any unexpected constraints. Hence, adhering to the Liskov's Substitution Principle.

There are classes in this new feature such as the new Weapon (RendingScissors) will implement interfaces such as Crippling however this interface only contains a very niche (specific) set of methods. Theoretically it would be possible to merge all the interfaces such as Resettable but a class should only be required to implement methods that it needs and not methods that is not needed. Hence, to ensure that we follow ISP be separating the interfaces, classes do not need to implement methods that they do not use.

The classes in this project pushes to implement the dependency inversions principle. The new classes introduced in this new feature also complies with this principle. For instance, the Player which is a high level module does not depend on low level implementation. For instance the player does not need to handle every single type of action there is (Attack, BreakShell, DoNothing, etc.). Instead, all actions share the same parent class and this leads to them sharing similar patterns and thus the player no longer depend on each action's methods but is instead streamlined to accept the Action abstraction which governs all other actions

Creative Requirement 2

UML Diagram



Description

This feature will add a new biome to the map in the form of a water area to diversify the map. This area will have new enemies waiting in the form of a squid and sharks that will relentlessly follow the player as soon as they have been detected. Player will drown in this area as their hp will continuously decrease. The player will be able to overcome this by purchasing a Snorkel from toad to prevent drowning in water.

- Water Area '~'
  - A type of ground that will drown a player without a snorkel (hurt by 5 hp every round)
  - Only swimmable enemies will be able to enter
  - Swimmable enemies are also not allowed to leave
- New Item: Snorkel '8'
  - Item used that will allow player to travel on water without taking damage from drowning
  - Can be purchased from Toad for 500$

- New enemies
    - Squid
        - Has an ink attack that drops ink into the water area
        - Standing in the ink will initial damage and will reduce the players chance of hitting by 50% for 5 turns
        - Actor can wash away ink by moving into clear water
    - Shark
        - Follows Player when detected(right beside)
        - Chomps player with 90 damage

- New Effects
    - INK_ATTACK
        - To indicate actor is in ink
    - AMPHIBIOUS
        - To indicate actor has a snorkel
    - DROWNABLE
        - To indicate actor can drown

Requirements

Must use least two (2) classes from the engine package	A new WaterArea class will be created that extends ground. A Snorkel class will be created that extends SpecialItem. Two enemies, Squid and Shark will be added that extends the Actor class. A new Ink class to implement effects of getting Inked by Squid that will extend Item.
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	The Snorkel will reuse the trade feature from Assignment 2 and can be purchased from Toad for 500$. Shark and Squid will use follow behaviour to actively hunt the Player as soon as the player is detected.
Must use existing or create new abstractions (eg., abstract or interface, apart from the engine code)	The enemy class created in assignment 2 will be reused to create Squid and Shark. Snorkel item will reuse the SpecialItem class from Assignment 2.
Must use existing or create new capabilities	Having a snorkel Item will give the player “AMPHIBIOUS” capability which prevents the player from drowning when in Water. A successful attack from the Squid will give the player an “INK_ATTACK” capability which will increase the Player’s likelihood of missing their target by 50% for the next 5 turns. Staying in water for more than 5 turns without a snorkel will give the player a “DROWNING” capability that will continue to decrease the player’s Hp by 5 until the player is no longer in water without a snorkel or until the player dies.

Design Rationale

Each class was created to ensure they only have one responsibility for instance:

1. WaterArea class for ground type water
2. Squid and Shark classes representing the enemies
3. Ink attack to add the effects of ink to an actor standing in it which will be released by the Squid
4. Snorkel class which will represent the Snorkel item used to trespass water without taking damage.

This will increase the locality of the classes and reduce connascence between classes.

Squid, Shark will extend Enemy. WaterArea and Snorkel extends Ground and Item classes respectively without modifying any of the aforementioned classes. This will ensure the classes adhere to the open-closed principle.

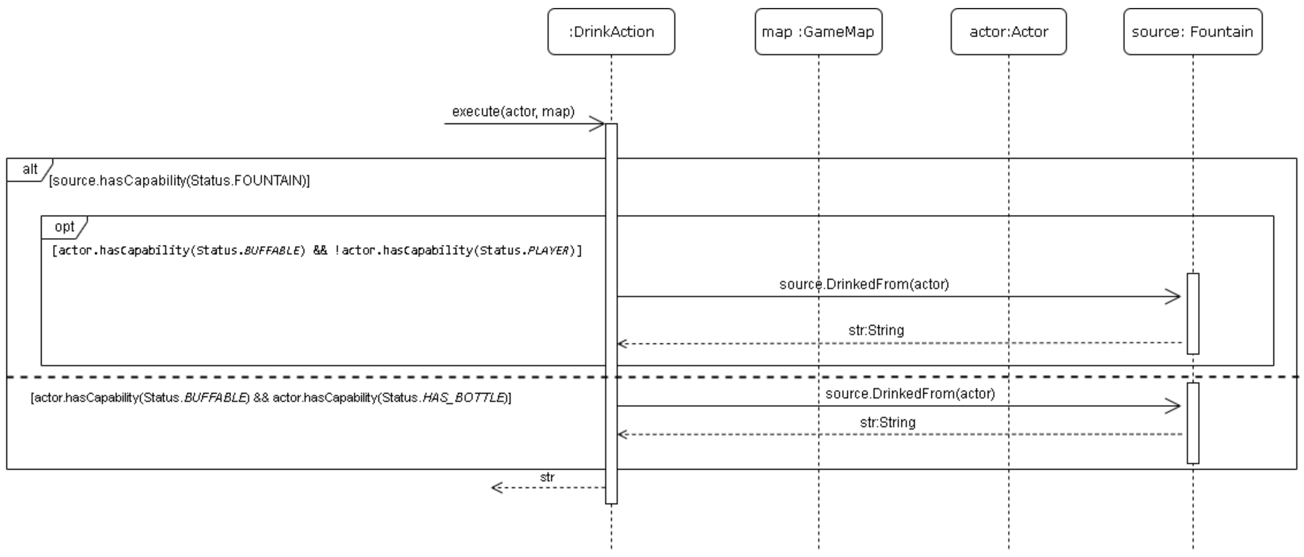
All extended classes (Squid, Snorkel, WaterArea, Shark and Ink) will not have unexpected behaviour by ensuring extended classes have similar behaviour to their parent class and any overridden methods will have similar results. For example, Snorkel will not have any methods from SpecialItem that have do-nothing implementations or any methods that behave differently from what is expected for SpecialItem subclasses.

The class Snorkel which extends SpecialItem will implement a unique Interface in the form of the TradeableItem interface which was introduced in Assignment 2 that only have methods which will all be implemented properly and not have do-nothing implementations. This is to adhere to Interface Segregation Principle by having Interfaces that are fully implemented instead of selectively implemented depending on the requirements of the class that implements the aforementioned interface.

The changes made to AttackAction and AttackBehaviour to accommodate Bowser's FireAttack have been reused for the Squid to drop Ink into the WaterArea. This is achieved using Statuses which prevent the classes from having needless dependencies which will also increase locality and reduce connascence.

The changes that have been made to AttackBehaviour and AttackAction for Bowser's fire attack have also been reapplied to Squid to add Ink to the water area which shows the extendability of the changes made to the classes to accommodate for new features and capabilities.

Sequence Diagram for DrinkAction.execute(actor, map)


:DrinkAction
map :GameMap
actor:Actor
source: Fountain
execute(actor, map)
alt
[source.hasCapability(Status.FOUNTAIN)]
opt
[actor.hasCapability(Status.BUFFABLE) && !actor.hasCapability(Status.PLAYER)]
source.DrinkedFrom(actor)
str:String
[actor.hasCapability(Status.BUFFABLE) && actor.hasCapability(Status.HAS_BOTTLE)]
source.DrinkedFrom(actor)
str:String
str