Assignment 3 - UML & Sequence Diagrams
MA Lab11 Team 1

## Group Members:

1. How Yu Chern
2. Eugene Kah Chun Fan

## Work Breakdown Agreement (WBA)

Assignment 3 - Version 1
Last Updated: 8/5/2022

Group Members Signature:
1) How Yu Chern - I accept this WBA
2) Eugene Kah Chun Fan - I accept this WBA

Task Breakdown and Summary:

- Each person in charge of a requirement (REQ) will be in charge of its respective UML Diagram and Design Rational components. Each member's components will be combined to create the complete version of the UML Diagram, and Design Rationale for all 4 Requirements. This will be done in each member's own time until the deadline. Each person's work will be reviewed by other members of the group, to ensure that it is complete and up to date.
- One interesting feature in one of the 4 requirements will be used to create a sequence diagram out of it.
- For Group Tasks, members will meet up and collaborate, working together to complete the groups tasks of the requirements.
- This WBA, and all 4 Requirements Diagrams and Design Rationale will combine into a single PDF for submission.
- Note that we are choosing **Creative Mode**

Tasks for How Yu Chern
Review By: Eugene Fan
Deadline: 15/5/2022

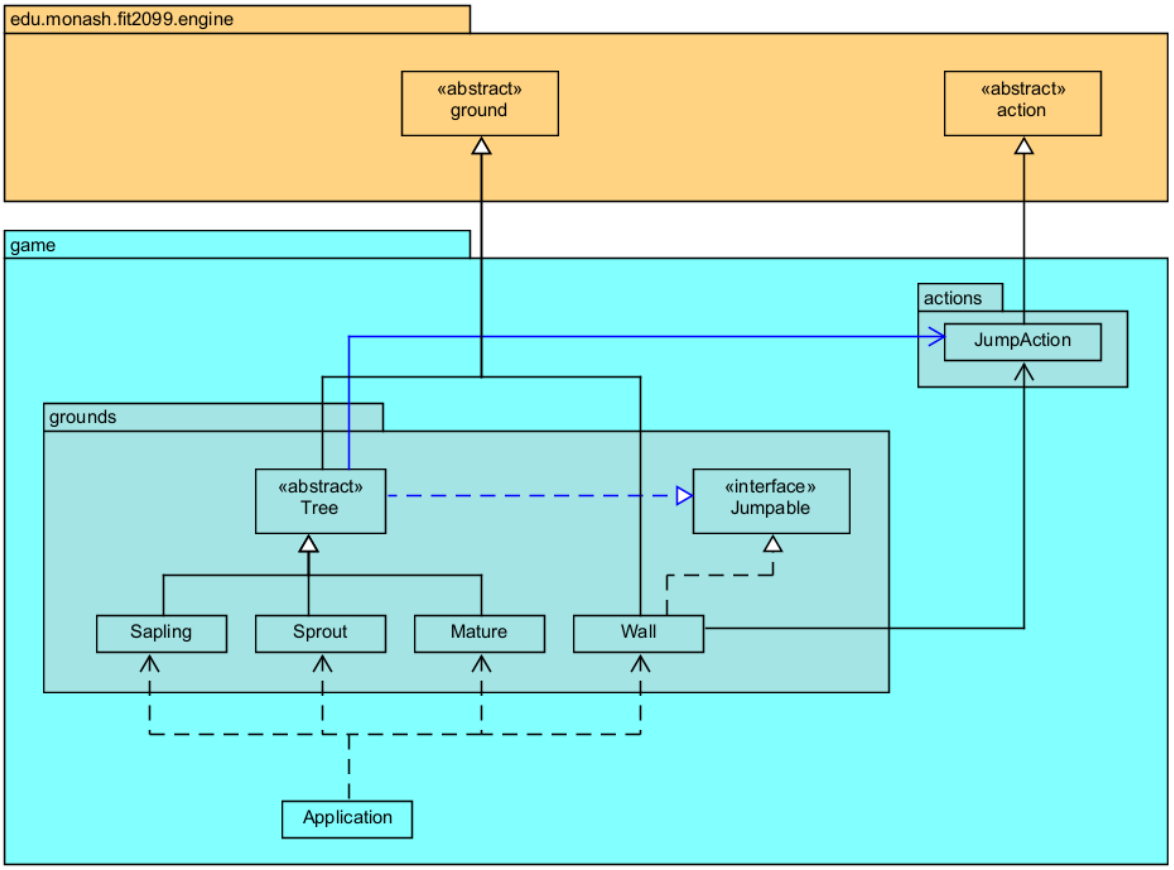
Tasks:
REQ1: Lava Zone
REQ4: Creative Task - Random Populator

Tasks for Eugene Fan
Review By: How Yu Chern
Deadline: 15/5/2022

Tasks:
REQ2: More Allies and Enemies
REQ3: Magical Fountain

## UML Diagrams

REQ2 of Assignment 1 & 2:




REQ2 of Assignment 1 & 2 Design Rationale:

Sprout, Sapling and Mature extends Tree abstract class. Walls and trees (which extend Ground and inherit its properties) are considered higher ground, and thus implement the Jumpable interface, which has methods that creates and gets JumpAction for the player if the player is near any of the 4 ground types. JumpAction allows the player to go to the higher ground by jumping onto them. This is because the four ground types are higher than normal ground and require the player to jump onto them.

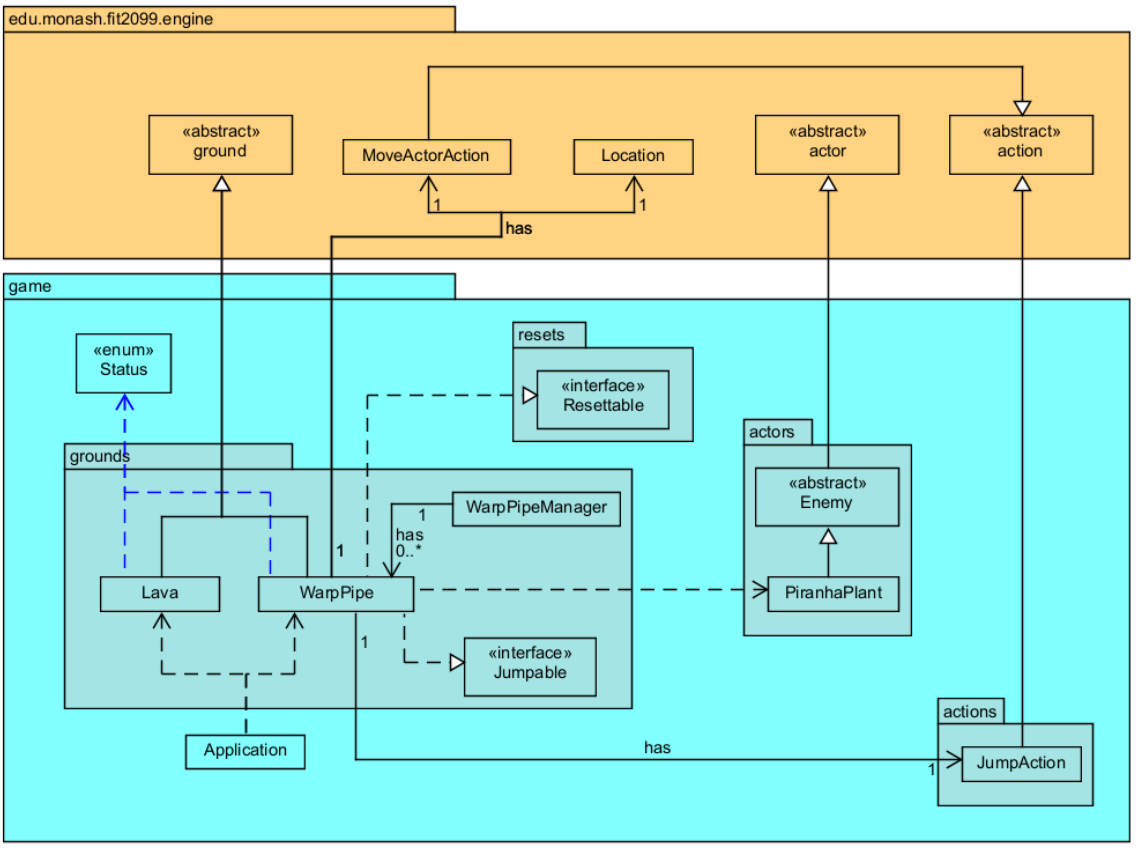
If a player chooses to jump onto the ground, it will call canActorEnter to check if the actor can enter it. The Jumpable interface requires those who implement it to create methods that specify a certain success rate for the player to jump onto, and inflicts a certain fall damage to the player for a failed jump. Each of the four ground types stores its own success rate and fall damage.

If the player meets requirements, the player can jump. If the player has Status.TALL capability, the interface will also allow the player to jump onto any of the four grounds due to super mushrooms giving players 100% jump success rate.

Creation of the Jumpable interface and having the trees and walls extend it aims to reduce excessive dependency of having 4 grounds use the default method of the interface to create jumpaction, rather than with repeated code. The Jumpable Interface also allows each of the

four ground types to implement similar behaviour, but in their own unique ways, allowing each ground type to have varying success rates and fall damage. Thus, this design aims to fulfil the DRY Principle, and the Principle of Classes Should be Responsible For their Own Properties.

REQ 1: Lava Zone




REQ1 Design Rationale:

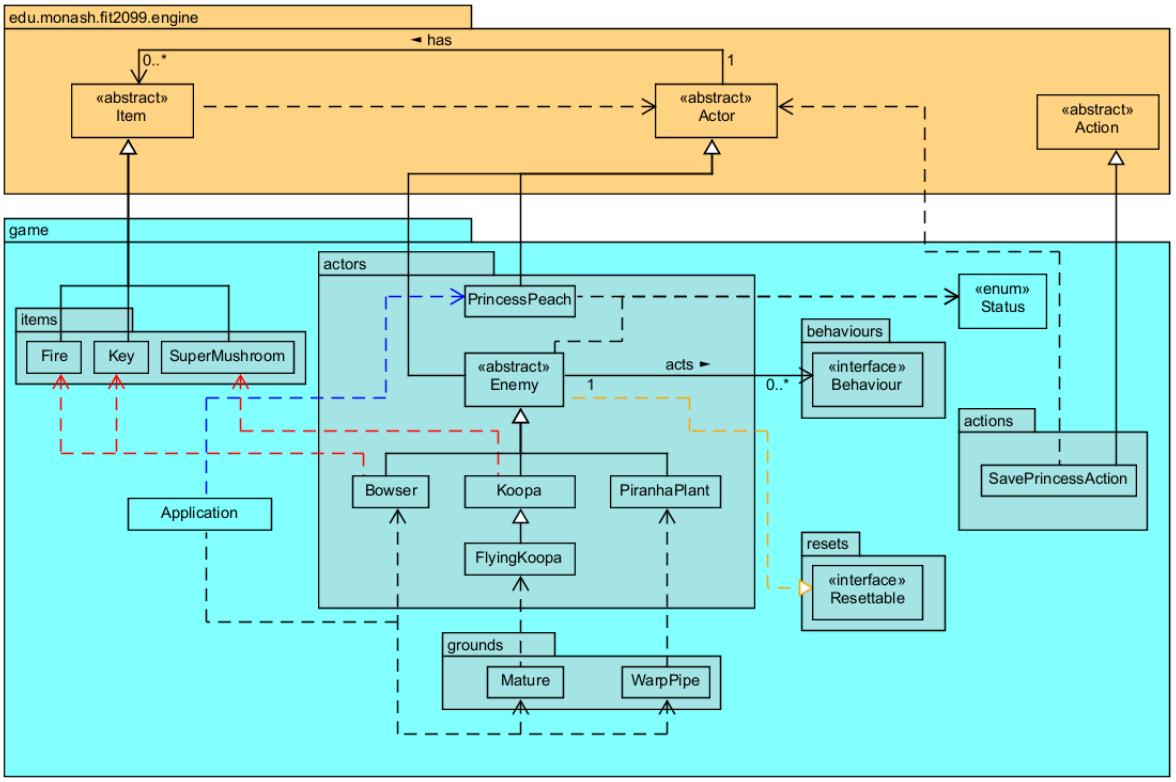
Lava is a class that extends ground, and has the functionality of a ground, with its own unique properties (such as damaging mario for 15 damage when he stands on it).

Warp Pipe was decisively created as a ground type. It extends ground abstract class, and implements Jumpable interface due to it exhibiting properties of that of a ground type and behaves similar to that of a higher ground, such as the player needing to jump onto a warp pipe to be able to warp using it, and it cannot be picked up from the map or entered by actors (except for the piranha plant that spawns on top ot it). Warp Pipe adheres to the Interface Segregation Principle, as the interface that it implements is in charge of jump actions of the warp pipe only, and no other different behaviours.

Warp Pipe also has a Warp Pipe Manager, which was created as a static factory method, since only one instance of WarpPipeManager is required to manage many instances of Warp Pipes across multiple game maps in a world. Warp Pipe Manager manages the warp (Move Actor Action) of each warp pipe, as well as sets the return warp pipe for lava zone's unique warp pipe.

Both Lava class and Warp Pipe class adhere to the Single Responsibility Principle, as they are in charge of their own respective properties as respective grounds. They also inherit common properties from their parent class, thus following the Do Not Repeat Yourself (DRY), and Reducing Excessive Use of Literals principles, through abstraction and inheritance. All classes used in creating REQ1 were ensured to be well encapsulated, with local variables set to private with appropriate getters and setters. This is to prevent other classes from accessing the private variables unintentionally.

REQ2:



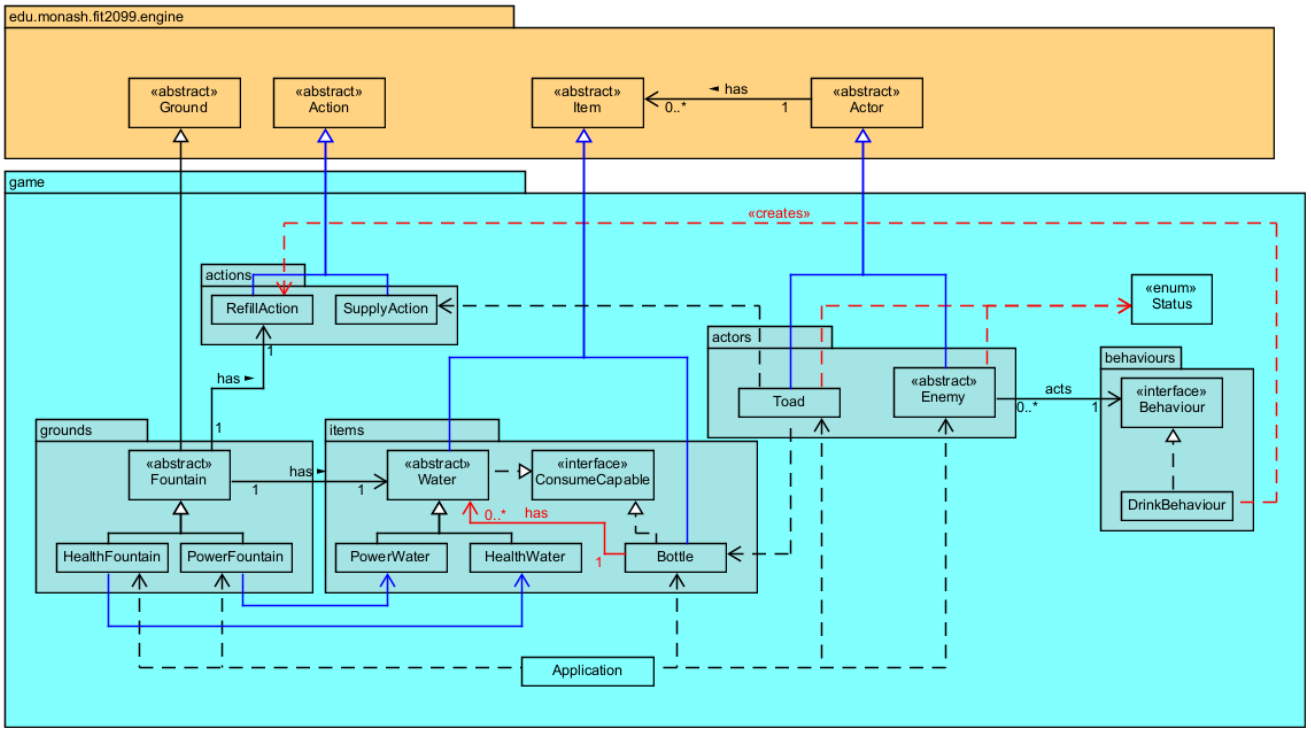

REQ2 Design Rationale:
There are four new actors. Princess Peach, Bowser, Flying Koopa and PiranhaPlant. Princess Peach inherits from the Actor class whereas Bowser and PiranhaPlant inherit from Enemy class. Flying Koopa inherits from Koopa class, which in turn inherits from Enemy class too. All these four actors inherit from other classes as the classes before them have similar methods that these new classes can use.

Do note that the inheritance from FlyingKoopa to Koopa is a display of the Liskov's Substitution Principle (LSP). This is because if a theoretical method takes in Koopa objects as a parameter, FlyingKoopa objects will also be acceptable, but it doesn't work if the method expects FlyingKoopa and Koopa is given. FlyingKoopa is spawned by Mature trees; PiranhaPlant is spawned by WarpPipes.

Bowser will have a Key item in its inventory. This Key allows the Player to save PrincessPeach and will be dropped by Bowser if it is defeated. While Bowser attacks, it will drop a Fire item onto the location of attack. This Fire item can hurt anyone standing on it and will fade in 3 turns.

Lastly, a SavePrincessAction is created. SavePrincessAction inherits from Action abstract class. This particular action is used by the Player to save PrincessPeach. By creating a specific class for a specific action, this adheres to the Single Responsibility Principle (SRP) as each responsibility will be split into class to avoid God Class.

REQ 3



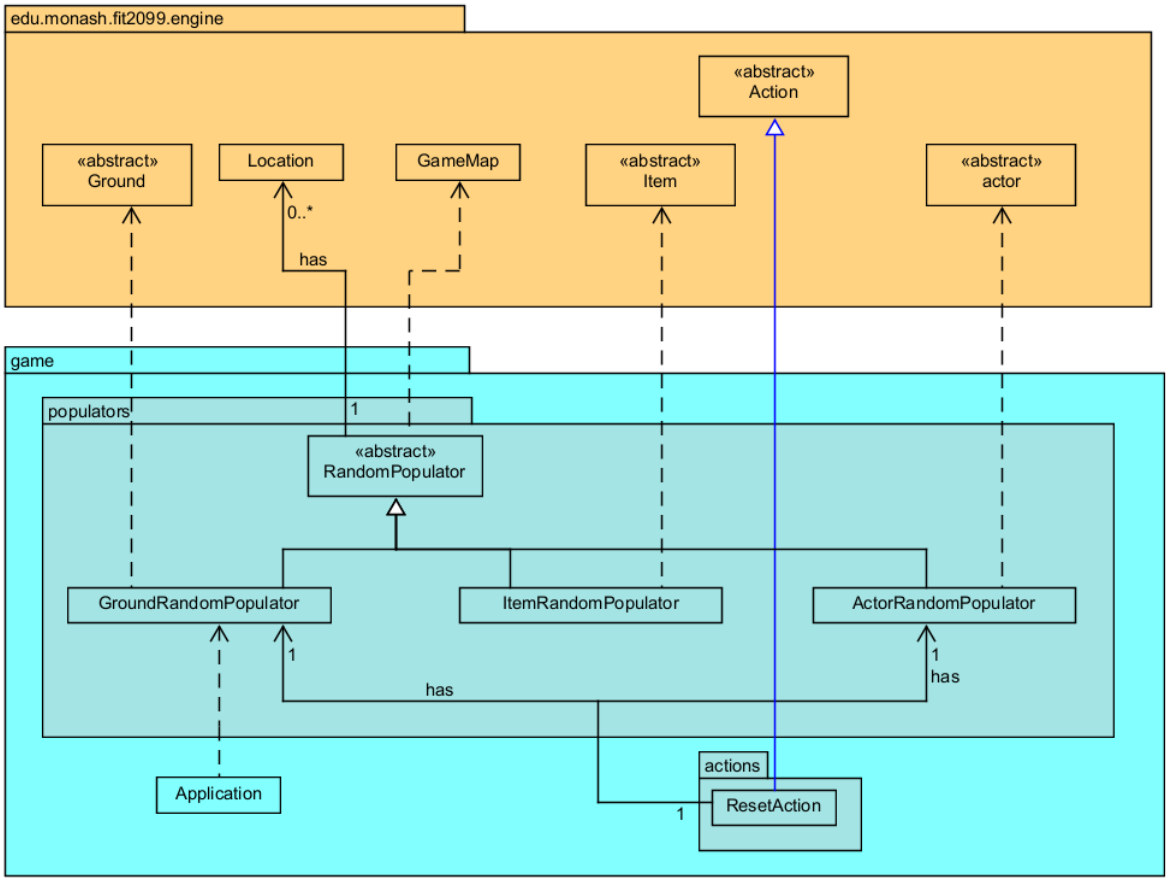

REQ3 Design Rationale:
Bottle is an Item object created. Bottle can be used by Player by serving as a stack capable of storing different types of Water. Bottle is supplied by Toad using the SupplyAction. Water (item) is an abstract class that is inherited by HealthWater and PowerWater.

Both Bottle and Water classes implement the ConsumeCapable interface as both of them are capable of being consumed by Actors. This adheres to the Interface Segregation Principle (ISP) as only items that can be consumed, will implement ConsumeCapable and get ConsumeCapable methods. Players can then consume the water from the bottle to gain abilities from the water.

Fountain is an abstract class which inherits Ground abstract class, and this Fountain class is also inherited by HealthFountain and PowerFountain. Again with the Single Responsibility Principle (SRP), HealthFountain supplies HealthWater, PowerFountain supplies PowerWater. Note that Actors can use RefillAction to refill from the Fountain.

Lastly, DrinkBehaviour is a Behaviour that is created to allow Non-Playable-Characters (NPCs) to drink from the Fountains. The DrinkBehaviour will call the RefillAction when it is executed by NPCs which are at the Fountains.

REQ4 (Creative Mode): Random Populator




REQ4 Design Rationale:
Random Populator is a feature that allows the game to randomly populate any gamemap with any of the three main game object types (i.e. Actors, Items, Grouds). It can be called to do so at the start of the game, or ay any time, such as when reset a reset occurs. The three variants (child classes) of the random populator class are GroundRandomPopulator, ActorRandomPopulator, and ItemRandomPopulator (This is mainly done due to the three game objects, which are actors, grounds, and items, extending different types in engine even though they all can be placed on the game map).

Firstly, Random Populator does adheres to the Single Responsibility Principle, as each variant (child class) of Random Populator is in charge of randomly populating the map with it's respective game object type, rather than random populator itself handling all 3. Secondly,it also adheres to the Open-close Principle, as the RandomPopulator parent class is open for extensions for other types of random populator. For Example, if there is a forth game object type, Heirlooms, that need to randomly appear on the map. A HeriloomRandomPopulator can be easily made by extending Random Populator parent class. Thirdly, it adheres to the Liskov Substitution Principle: An instance of the a random populator variant (child class) is called in place of the Random Populator parent class, which is abstract. Thus, the functionality of RandomPopulator is preserved by it's variants. Forthly, it also adheres to the Interface Segregation Principle, as the parent and child classes of random populator does not implement interfaces that are not required by the class. Random Populator also adheres to the Dependency Inversion Principle, as none of it's concrete class

depends on another concrete class. All concrete classes of Random Populator are child classes that extend an abstract class.

**Sequence Diagrams**

1. HealthFountain's refill()