2022

FIT2099 Assignment 3

LAB 3 TEAM 2

SHANTANU THILLAI RAJ, CHAI LI GUANG, EDELYN SEAH

Work Breakdown Agreement for FIT2099 Assignment 3

Team 2 Lab 3:

1. Chai Li Guang (31858988)
2. Edelyn Seah (31107559)
3. Shantanu Thillai Raj (32141580)

	Tasks	Person-in-charge	Date to be completed
1	Implementing requirement 1	Chai Li Guang	15/5/2022
2	Implementing requirement 3	Shantanu Thillai Raj	15/5/2022
3	Implementing requirement 2	Edelyn Seah	15/5/2022
4	Implementing creative requirement 4	Edelyn Seah, Shantanu Thillai Raj	21/5/2022
5	Implementing creative requirement 5	Chai Li Guang, Shantanu Thillai Raj	21/5/2022
6	Testing implementation of requirement(s)	Chai Li Guang, Shantanu Thillai Raj, Edelyn Seah	22/5/2022

Chai Li Guang will be responsible for the above tasks. Reviewer: Edelyn Seah, Tester: Shantanu.

Signed by: ***Chai Li Guang***

Edelyn Seah will be responsible for the above tasks. Reviewer: Shantanu, Tester: Li Guang
Completion: ***Edelyn Seah***

Shantanu will be responsible for the above tasks. Reviewer: Chai Li Guang, Tester: Edelyn Seah

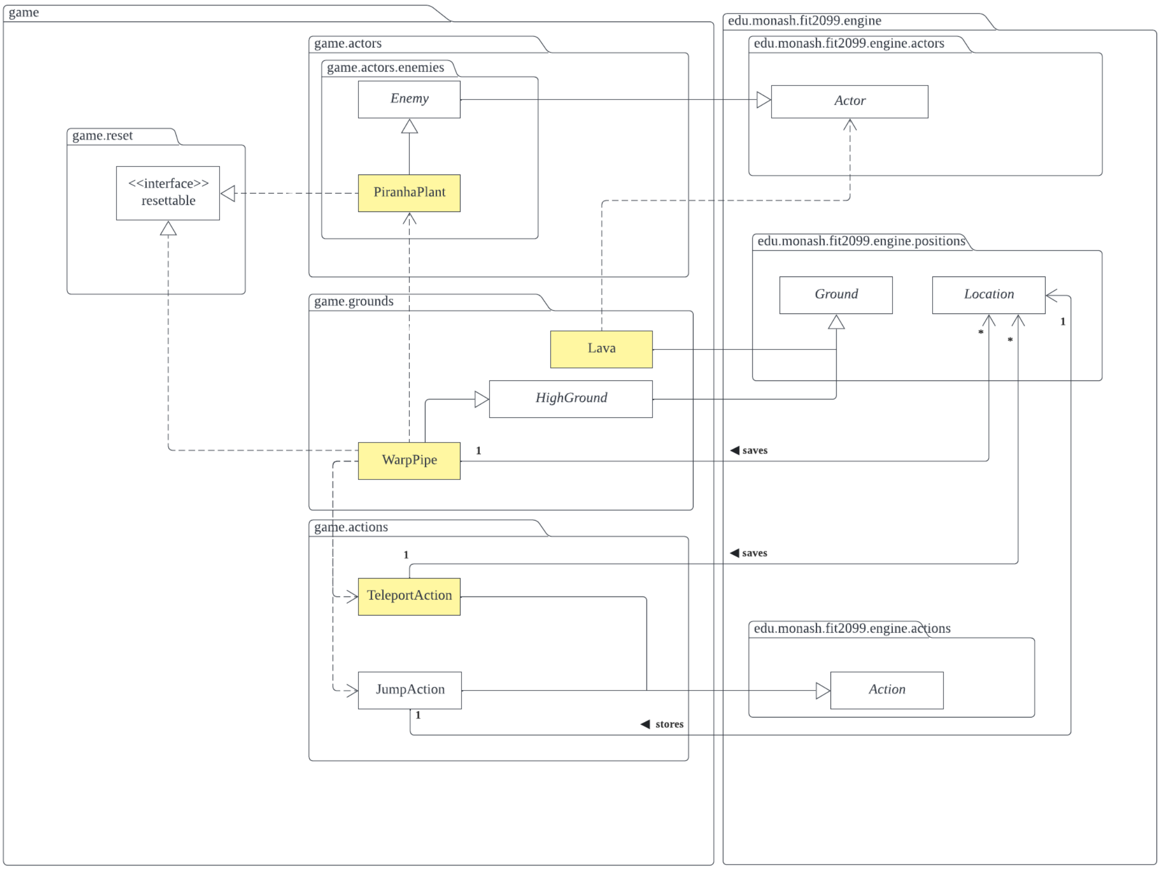
Completion: ***Shantanu Thillai Raj***

Changes from Assignment 2 to Assignment 3

- Changes based on feedback for implementing Requirement 4 in Assignment 2.
- Remove specific actions for pick up and consumption for power star and super mushroom.
- Made a general consume action for any consumable item.
- For any consumable item, it must implement the Consumable interface and use the action ConsumeAction.
- These changes can be seen properly in creative requirement 2, the consumable fire potion does the above.

Requirement 1: Lava Zone

UML Diagram



Design Rationale

General summary of operation

Lava Ground burning player

- Created a new Lava class extended from the Ground class in the game engine code
- In this requirement, the only feature it has is to damage the player(other actors like enemies are not allowed to enter).
- Implemented the hurting the player mechanism by overriding the tick() from Ground class. If the actor does not have the Status/Capability ENEMY, then the actor will reduce its hp by 15 by using the method hurt(), where we would pass in the damage(int) into the method.

Spawning Piranha Plant Enemy on WarpPipe

- All WarpPipe objects will spawn a Piranha Plant enemy on the second turn of the game, if player chooses to reset the game, it would check if the WarpPipe has the capability/status RESET and if it contains any actor on it, then only spawns a new Piranha Plant, that's why it implements the resettable interface so that it would get added into the ResetManager and get access to the resetInstance() which WarpPipe will have the capability/status RESET when the game is resetted.

Teleporting between the game maps

- Created WarpPipe class that is extended from the HighGround class and implements the resettable interface that was introduced in assignment 1&2. This class object sets as a teleportation platform for the game.
- There is an overloading constructor for WarpPipe to take in the target location as parameter when initializing it in the Application class.
- Only non-enemy actors can enter WarpPipe by jumping onto it, so the player would have a jump action shown in the menu just like jumping to any other high grounds.
- When player is standing on the WarpPipe it will only add the TeleportAction, which will take in some parameters such as (location of source pipe, location of target pipe, strings representing the name of the maps the target and source pipe is in)
- In the execute() of TeleportAction class, it will moves the actor to the target pipe location, and removes any Piranha Plant if the target pipe has any. It will also create a new Warp Pipe on the target location to change the target and source location of t he warp pipe so that it can teleport back and forth to the correct warp pipe at the same location.

Principles Applied

Single Responsibility Principle

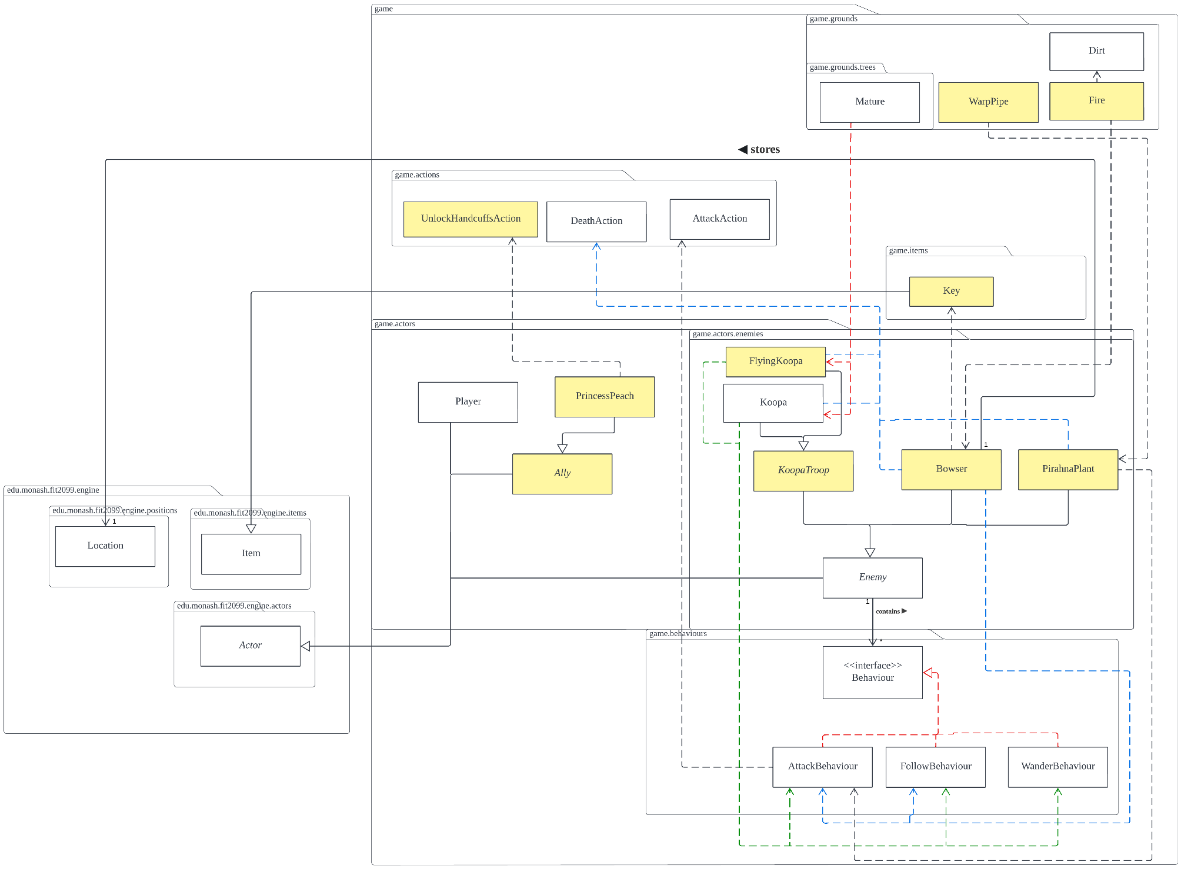
- TeleportAction only used to moves the actor from the source location to the target location and setting the target location with a new warppipe()
- Lava is a ground class is only responsible to deal a certain amount of damage to the player if they are stepping on it.

Interface Segregation Principle

- Since the Enemy and WarpPipe class implements the resettable interface, they will need to implement the methods like resetInstance(). And both the enemy and WarpPipe() requires all the methods in the resettable interface to make it work. That's why the resettable interface has applied the principle.

Requirement 2: More Allies and Enemies

UML Diagram



Design Rationale

General summary of operation

Implementation of Princess Peach Ally

- Princess Peach is categorized as an ally and extends from the ally class which extends from the Actor.
- To interact with the princess, the Key must be obtained by defeating Bowser and picking up the key using PickUpItemAction (in the engine class) and when the key is obtained, the player will own a Status called KEY.
- In the Princess class, if the actor that is interacting with the princess have a KEY status, then the UnlockHandcuffsAction will be available and a monologue (“Thank you for saving me!”) will be printed along with the actor being removed from the map using map.removeActor(actor) to end the game.

Implementation of Bowser Enemy

- Bowser stores the original location upon initialisation of the game on the BossMap, therefore it has an association with the Location engine class.
- Bowser extends from Enemy abstract class. Bowser does not wander, but it will attack and follow the actor when it is on one of its exits. Therefore it only has a dependency with the AttackBehaviour class and the FollowBehaviour class but not the WanderBehaviour class.
- Bowser will drop a Fire Ground (which causes damage to any actor) on where Mario is standing and it will add a new Dirt class into the ground after 3 ticks.
- Upon attacking (80 damage), Bowser will drop the fire on the ground where Mario is standing and deal additional 20 damage.
- If it dies, it will return the DeathAction.

Implementation of Piranha Plant Enemy

- Piranha Plant is spawned on the WarpPipe and extends from Enemy abstract class.
- It does not wander or follow the actor, therefore it only has a dependency with the AttackBehaviour class but not the FollowBehaviour class and the WanderBehaviour class.
- If it dies, it will return the DeathAction.

Implementation of Flying Koopa Enemy

- A Flying Koopa is spawned from the Mature class (which has a 50% possibility of spawning either a Flying Koopa or a Koopa)
- It flies above any ground, regardless of it being a high ground or not- other enemies are unable to go above certain grounds.
- Other implementations are the same as the Koopa (with the dormant capability etc).
- If it dies, it will return the DeathAction.

Principles Applied

Single Responsibility Principle

- PickUpKeyAction follows the Single Responsibility Principle because each of the actions are specific to performing one action.

- Each character class follows the single responsibility principle because it is specific to the character’s actions and behaviours only.

FutureProofing

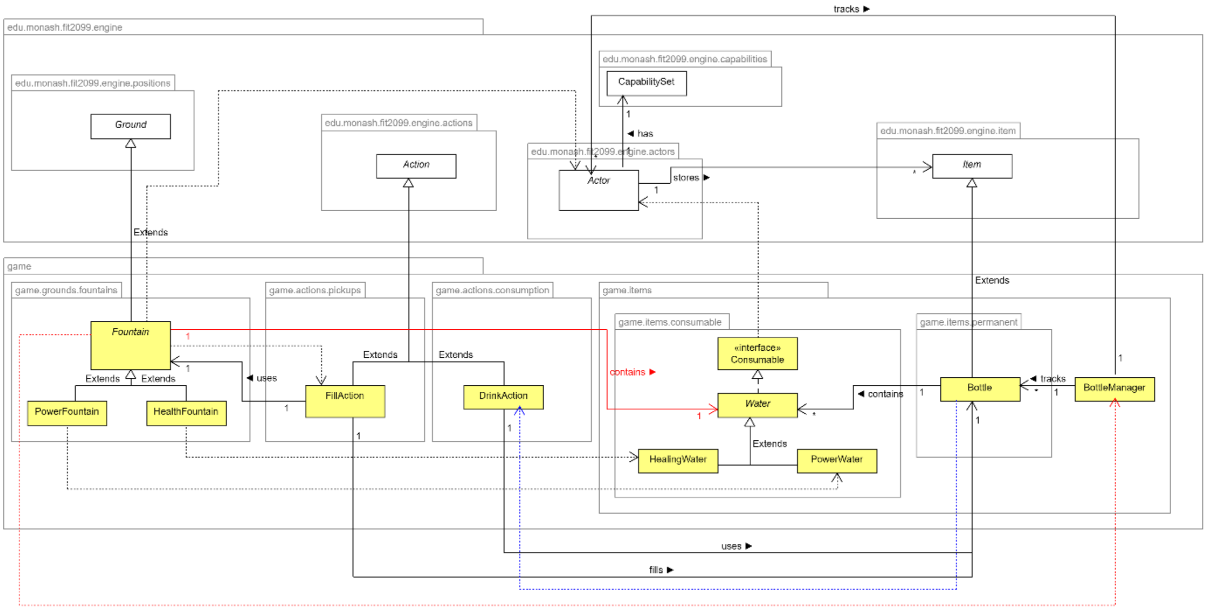
- Enemy and Ally abstract classes, we can easily add new enemies and allies in the future.
- KoopaTroop abstract classes also enable more koopa variants to be added.

Open-Closed Principle

- Enemy and Ally as well as KoopaTroop abstract classes adhere to this principle.
- Koopa and Flying Koopa extend the Koopa classes, without changing the methods in the abstract classes at all.

Requirement 3: Magical Fountain

UML Diagram



Design Rationale

General summary of operation

- *Did not implement optional requirements*

  ### Water Objects

  - Water abstract class implements Consumable interface.
  - Each water type (Power Water & Healing Water) extends from Water abstract class implementing the consume method (action that would occur on Drink Action execute)
  - When water is consumed (consume(actor) method called), the water does its specific buff for the actor.
  - For Power Water, it adds a capability to the actor and on the actors PlayTurn execute it increases their intrinsic attack value by 15 and removes the capability.
  - For Healing Water, it just directly uses the built-in heal method to heal the actor by 50.

  ### Fountain & Fill Action

  - Fountain is a new abstract class that extends from ground, it has instance variables: name (String) and water (Water)
  - The name is used for printing, the water instance variable is to set the water available for an actor to fill his bottle at the fountain with.
  - If an actor is standing on the fountain, using the Bottle Manager to get the actor's bottle, if the Bottle Manager returns the bottle and not null, create a new Fill Action for the actor to choose to execute. (Fill Action constructor takes in the bottle and fountain)
  - Fill Action gets the water from the fountain and adds it to the bottle.
  - Power Fountain and Health Fountain extend from the Fountain class to be set on the map. (Each holding its corresponding water)

  ### Bottle & Drink Action

  - Bottle extends from Item to allow it to be in the inventory of actors.
  - Bottle Manager used to keep track of bottles, whenever a bottle is added for an actor, the bottle manager saves it in a hashmap with the key being the actor (enforces 1 actor 1 bottle rule)
  - Drink Action is always available for actors if the bottle exists in inventory. (If bottle is empty, actor can still drink, will just print a message no buffs added, a wasted turn)
  - Bottle keeps track of water using a stack type, the Fill Action pushes water into the stack and on drink action execution, a water is popped from the stack and consumed by the actor.

Principles Applied

Futureproofing

- Using a bottle manager to keep track of the bottles instead of hard coding bottle to be the first item in inventory allows for new feature for example implementing the optional feature where player obtains bottle from toad, we would still be able to access the bottle via the Manager
- Fountain and Water abstract classes, we can easily add new fountains and water without changing any actions just by extending Fountain/Water.

Single Responsibility Principle

- Bottle Manager only used to keep track of the bottles (with their corresponding actor)
- Water and Fountain abstract class used to specifically implement all the different water and fountains via extending them.
- Fill Action is only to fill a bottle from a fountain.
- Drink Action is only for an actor to drink from a bottle (consuming a water from the bottle if it exists)

Open-Closed Principle

- Water and Fountain abstract classes adhere to this principle.
- Power Water/Healing Water & Power Fountain/Healing Fountain extend the Water and Fountain classes respectively, not changing the methods in the abstract classes at all.

Liskov Substitution Principle

- Water -> Power Water/Healing Water and Fountain -> Power Fountain/Healing Fountain. Both these cases adhere to this principle.
- Water is extended just to have the specific implementations of consume in each concrete water class.
- Fountain is extended just to have concrete classes of fountains with different water types.
- No useless unused methods in either implementation subclasses.

Dependency Inversion Principle

- Water and Fountain used to adhere to this principle.
- All the different waters and fountains do not depend on a concrete class instead it depends on the abstract classes.
- Consumable interface used for water to ensure it implements a method on what to do if an actor consumes that water.
- Bottle holds the abstract water type in stack instead of each specific water.

Creative Requirement(s) Tabulated Summary

Creative Requirement 1 : New Enemies with specialities

New Enemies : **Big Slime, Small Slime, Flame Ward**

Requirements	Features
Must use at least two (2) classes from the engine package	Item, Action, Actor
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	Re-used enemy behaviours (AttackBehaviour, WanderBehaviour and WanderBehaviour)
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	Created new SlimeTroop abstract class, new GroundChangeAction, new GroundDestroyBehaviour
Must use existing or create new capabilities	FIRE_IMMUNE, BIG_SLIME, BROKEN

Creative Requirement 2 : New Weapons/Consumables with specialities

New Weapons : **Dragon Scale Blade, Hammer**

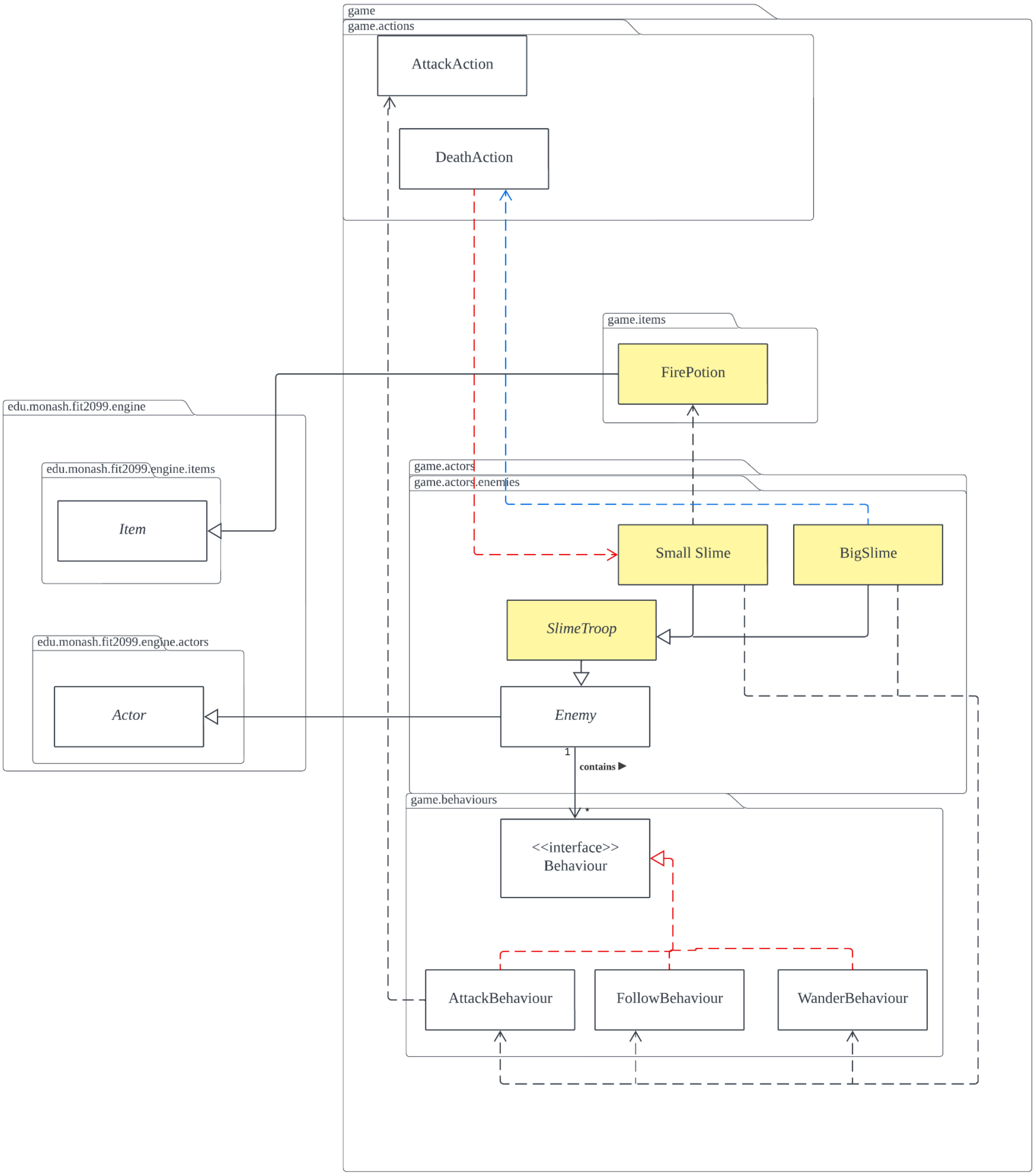
New Items: **Fire Potion**

Requirements	Features
Must use at least two (2) classes from the engine package	Item, Action, WeaponItem, Actor
Must use/re-use at least one(1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	The purchasing mechanism introduced in assignment 2 (buying items from Toad)
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	Used the Purchasable interface from assignment 2
Must use existing or create new capabilities	FIRE_IMMUNE, BROKEN

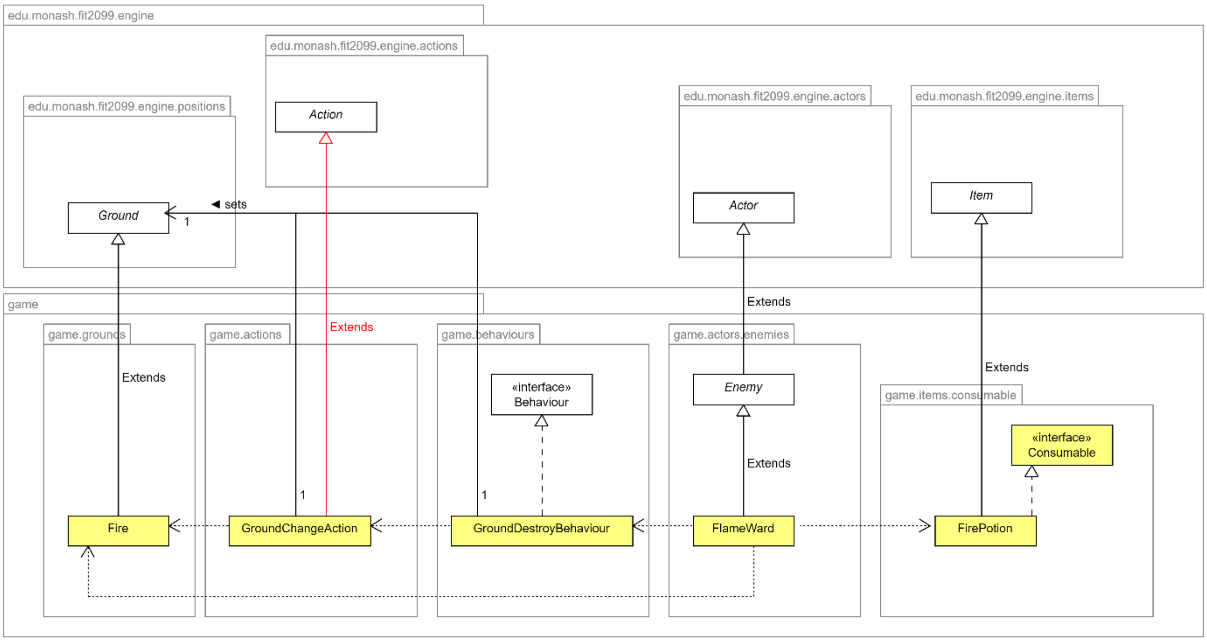
Creative Requirement 1: New Enemies

- 2 UML diagrams for better readability. (1 for the slimes, 1 for flame ward)
- New enemies : Big Slime, Small Slime, Flame Ward

UML Diagram for Big/Small Slime Enemy



UML Diagram for Flame Ward Enemy



Design Rationale

General summary of operation

Big Slime Enemy

- The Big Slime enemy extends from the SlimeTroop class. It has a dependency with the Wander, Follow and Attack Behaviours.
- It has Status FIRE_IMMUNE to ensure that the enemy is able to wander and follow the actor regardless of whether it is a Fire ground or not.
- It has a Status BIG_SLIME to be checked during the DeathAction. If the Slime is a BigSlime, it will spawn smaller slime enemies (SmallSlime) in the DeathAction on all the exit locations of the BigSlime enemy.

Small Slime Enemy

- The Small Slime enemy is almost identical to the BigSlime enemy, aside from the maximum hit points being half of the Big Slime, and it drops a FirePotion when it dies.

Flame Ward Enemy

- Non-moving enemy that every 5 turns sets its exits on fire.
- GroundDestroyBehaviour is added to flame ward for setting of surroundings on fire.
- When killed for the 1st time, becomes BROKEN. Stops setting things on fire and is healed to full. When killed again, dies fully and drops a Fire Potion item.

GroudDestroyBehaviour

- Behaviour that every 5 turns sets all surrounding ground of the actor (all exits) to be a different ground type.
- When created, it takes in a ground object. (That ground object will be what is created every 5 turns)

GroundChangeAction

- Action that takes in input of a ground type and on execute sets all exits of the actor to that ground type.

Principles Applied

Futureproofing

- GroundChangeAction takes in the ground object we would like to set the exits to. Therefore, it is usable in setting other ground objects not just fire (only setting fire used by the flame ward in current implementation)
- Behaviours are generic and can be used by any actor, enemy or ally alike.

Single Responsibility Principle

- Each specialized enemy class: Big Slime, Small Slime and Flame Ward only relate to their specific enemy and their specific actions/behaviours/specialities.
- GroundChangeAction is responsible for specifically setting all grounds around the input actor to the specified input ground type.

- GroundChangeBehaviour is only responsible for calling the GroundChangeAction every 5 turns.

Dependency Inversion Principle

- All generics done by all enemies are implemented in the Enemy abstract class. All enemies here extend the Enemy abstract class and implement their specific methods after.
- No full dependency on a concrete class. Only depends on the abstract Enemy class to work. (dependency here != dependency relationship)

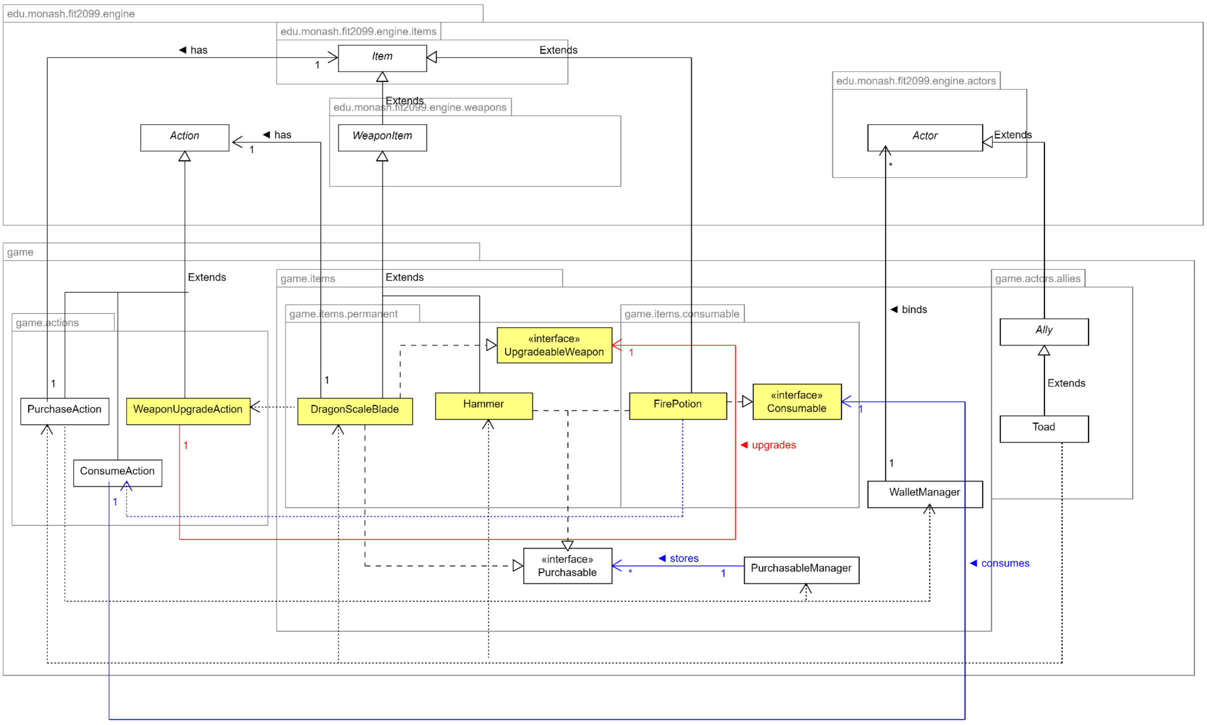
Liskov Substitution Principle

- All the new enemies can be replaced with just the generic abstract class enemy and it will not disrupt the system. (Only lose the specifics of those enemies)
- All methods used/implemented : no empty methods.
- No use of getClass()/instanceof to determine which enemy it is.

Creative Requirement 2: New items/weapons

- New items : Fire Potions (Consumable)
- New weapons : Dragon Scale Blade (Upgradeable), Hammer

UML Diagram



Design Rationale

General summary of operation

Dragon Scale Blade

- Dragon Scale Blade is an upgradeable weapon where an actor can sacrifice 50HP to increase the attack of the Dragon Scale Blade by 50.
- This weapon implements the UpgradeableWeaponInterface.
- If the blade is in inventory, it provides the option to upgrade in the menu via an action.
- If the actor dies when upgrading the weapon (does an upgrade with HP <= 50) the weapon breaks and the actor's health is set back to HP right before upgrade. (losing all previous upgrades and the ability to upgrade ever again)

UpgradeableWeaponInterface

- Interface that should be implemented by any weapon that can be upgraded.

WeaponUpgradeAction

- Takes in an UpgradeableWeapon to be upgraded.
- On execute, calls the upgrade method that the interface forces upgradeable weapons to implement.

Fire Potion

- Created a new FirePotion class extended from Item class in the game engine, and also implemented the Purchasable and Consumable interfaces.
- The potion would be sold by toad from 60 coins, it is also obtainable through killing small slimes (final form of big slimes) and Flame Ward enemies.
- Upon consuming the potion, the actor(player) will have a new capability added (FIRE_IMMUNE), which indicates that the player will take 0 damage from any kind of fire damage(from lava or fire ground) on the next encounter, then the buff/status will be removed.

Hammer

- Created a new Hammer class extended from WeaponItem class in the game engine, and also implemented the Purchasable interfaces.
- The purpose of creating this new weapon is to help the player to have an easier time to defeat the Bowser and other enemies as this weapon deals more damage than a wrench and of course the intrinsic weapon(unless the user focuses on drinking the power water).

Principles Applied

Futureproofing

- Implemented Consumable interface for all consumable items (changed power star and super mushroom in A2 to implement it as well)
- The Consumable interface allows us to use a single action ConsumeAction to consume all consumable items instead of creating a consume action for each item. (Removed the specialized actions to consume Power Star & Super Mushroom implemented in A2)

- The Consumable interface allows us to not need new consume actions for every consumable item. The item just needs to implement the Consumable interface.
- Similar to Consumable interface, Upgradeable weapon interface also allows any future weapons to implement it, making that weapon upgradable using the WeaponUpgradeAction.

Single Responsibility Principle

- Each item class is responsible for its specific item specification
- The WeaponUpgradeAction has a single responsibility of upgrading weapons on execute
- Weapon classes like Hammer and Dragon Scale Blade class will only be responsible to act as a weapon item which will be used by the player.

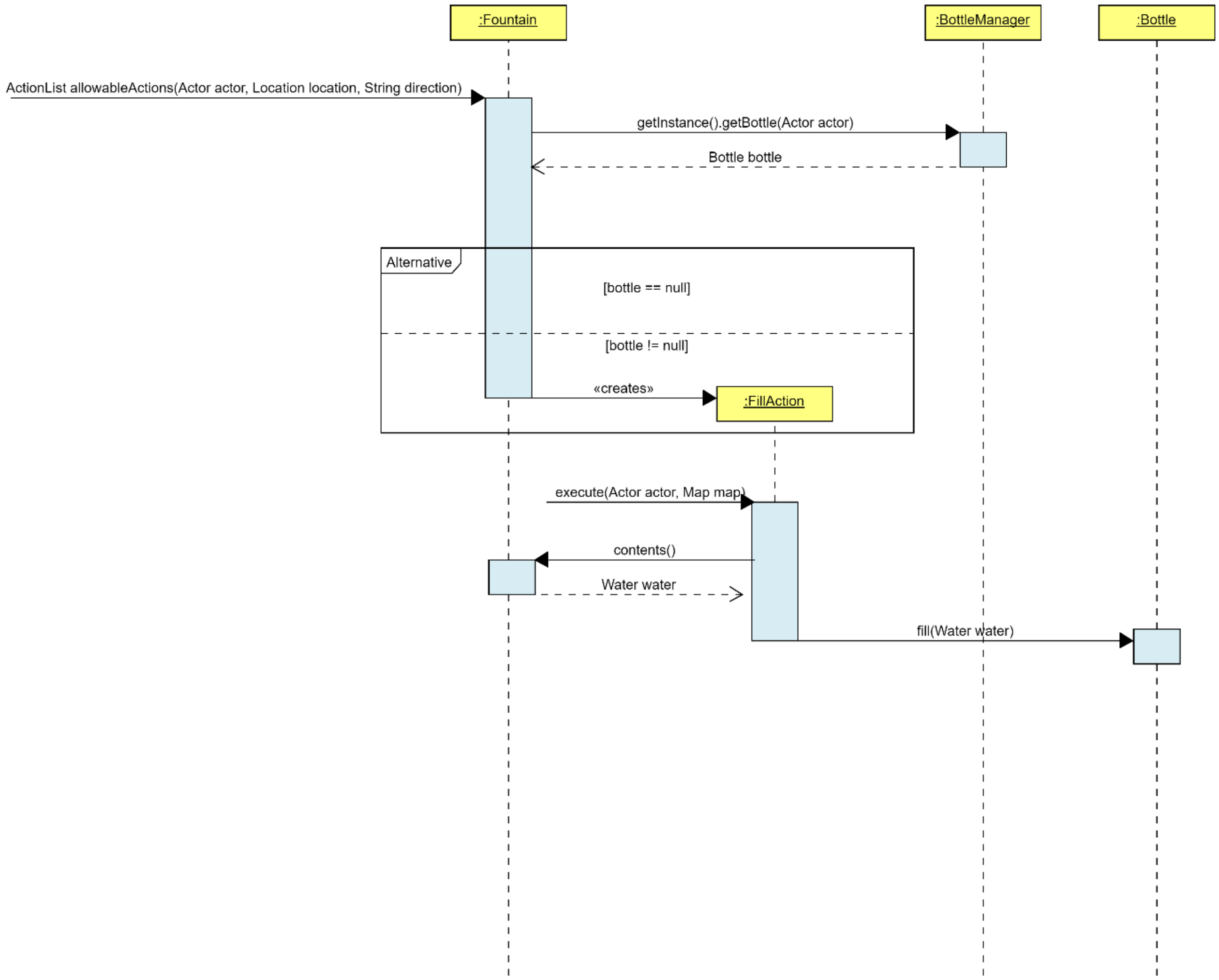
Dependency Inversion Principle

- Each new item/weapon is independent from each other, changes to one does not affect the other. This is because all these weapons/items only depend on the abstract weapon/item class instead of each other.

Interface Segregation Principle

- Interface for Consumable, Purchasable and Upgradable denoting 3 different possible items. An item can implement any number of these interfaces.
- Interfaces allow for flexibility in making an item available to specific actions.

Sequence Diagram for Fountain & Fill Action


:Fountain
:BottleManager
:Bottle
ActionList allowableActions(Actor actor, Location location, String direction)
getInstance().getBottle(Actor actor)
Bottle bottle
Alternative
[bottle == null]
[bottle != null]
«creates»
:FillAction
execute(Actor actor, Map map)
contents()
Water water
fill(Water water)