FIT2099
Assignment 3
Due date: 23/05/2022 02:55 AEST

by

NG YEN KAI

TEH RONG WEN

LIM SENG ONN

WORK BREAKDOWN AGREEMENT

Deliverables:

Separate interaction(optional), class diagrams and design rationales for each requirement.

- ☐ REQ1: by Lim Seng Onn (32200250)
- ☐ REQ2: by Ng Yen Kai (32217021)
- ☐ REQ3: by Teh Rong Wen (31882617)
- ☐ REQ4: by Teh Rong Wen (31882617)
- ☐ REQ5: by Ng Yen Kai (32217021)

Testing and Reviewing:

All members will be responsible for testing the requirements they have been assigned to.

- ☐ Lim Seng Onn will be reviewing Teh Rong Wen's tasks/requirements.
- ☐ Teh Rong Wen will be reviewing Ng Yen Kai's tasks/requirements.
- ☐ Ng Yen Kai will be reviewing Lim Seng Onn's tasks/requirements.

Deadlines

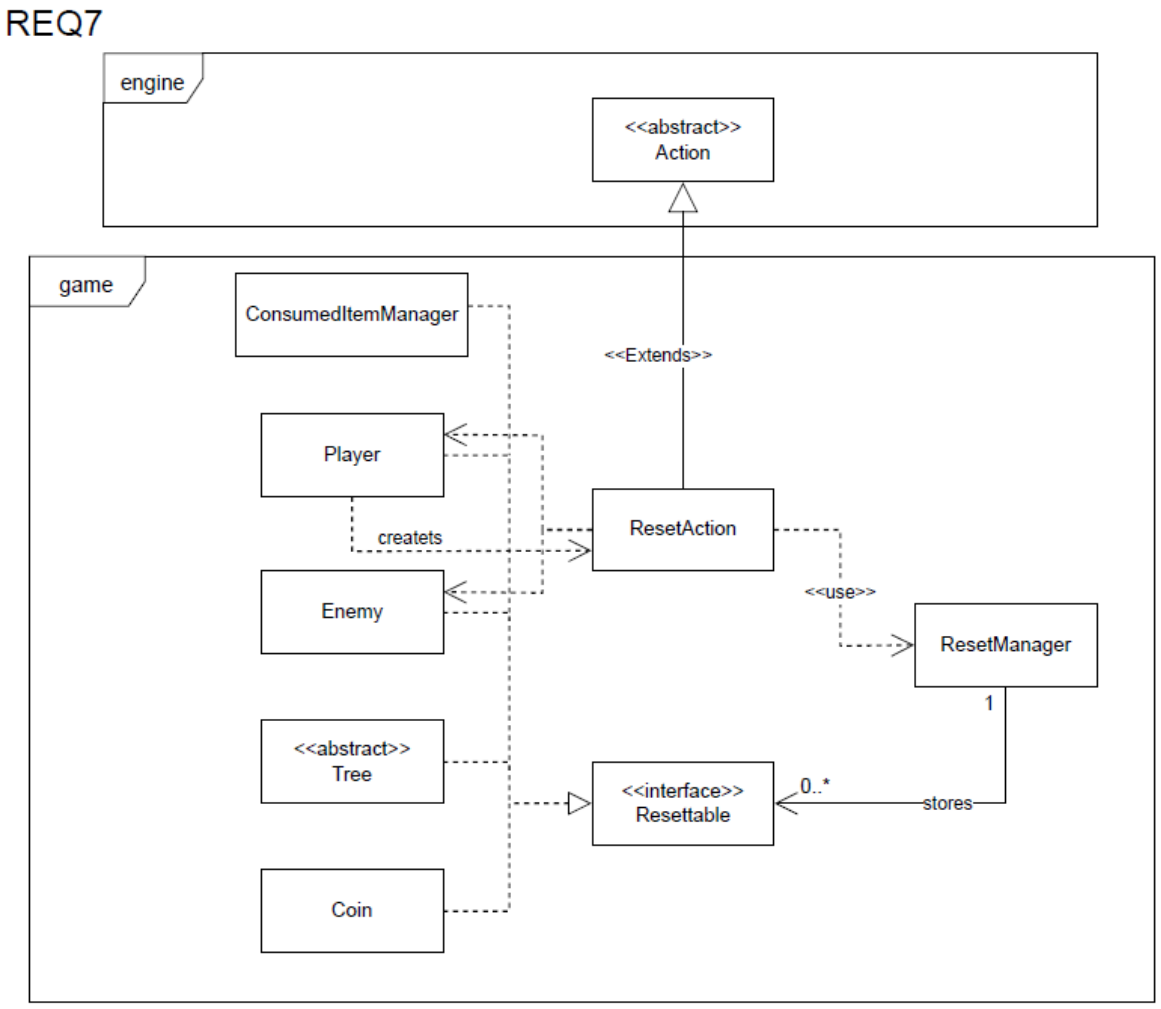
The dates by which the deliverable, test, and review need to be completed.

- ☐ Deliverable: Friday, 20th May 2022
- ☐ Testing: Saturday, 21th May 2022
- ☐ Review: Saturday, 21th May 2022

Signatures

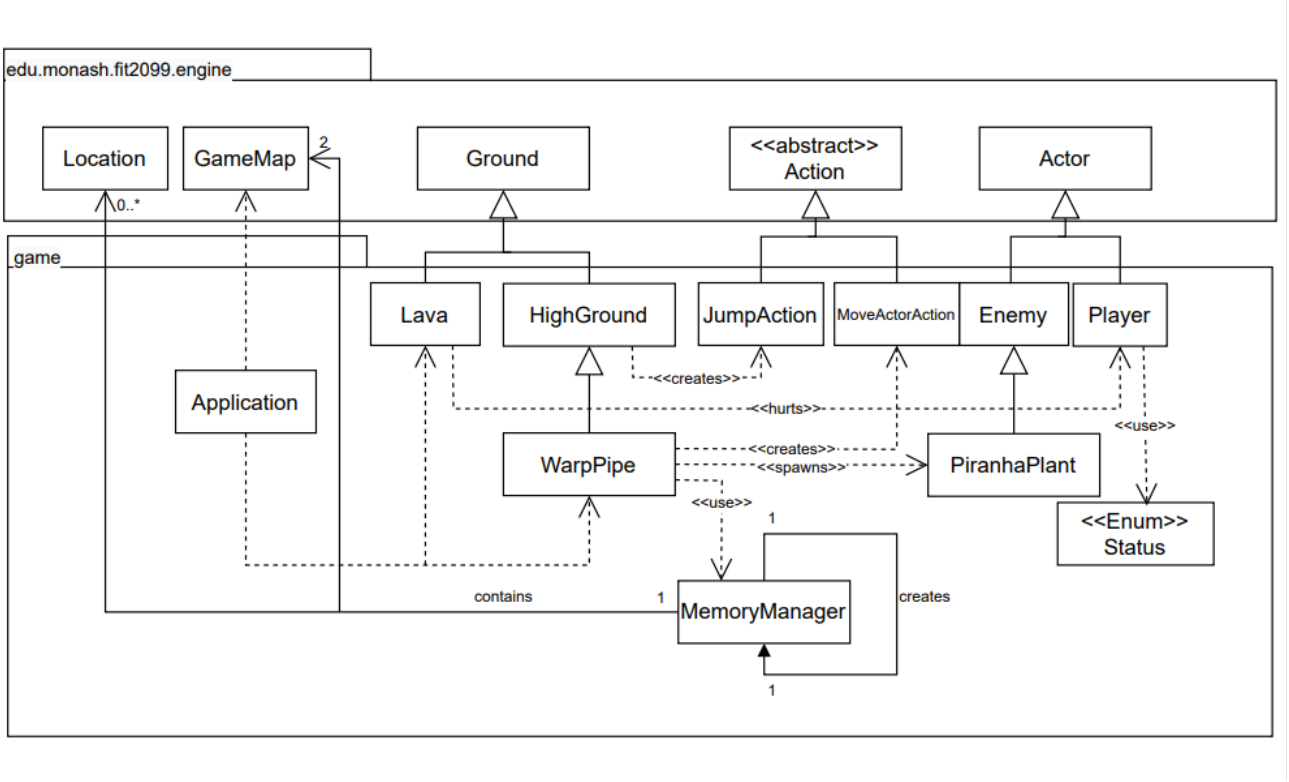
Teh Rong Wen: I accept this WBA
Lim Seng Onn: I accept this WBA
Ng Yen Kai: I accept this WBA

Assignment 2, UML Class Diagram (REQ 7)

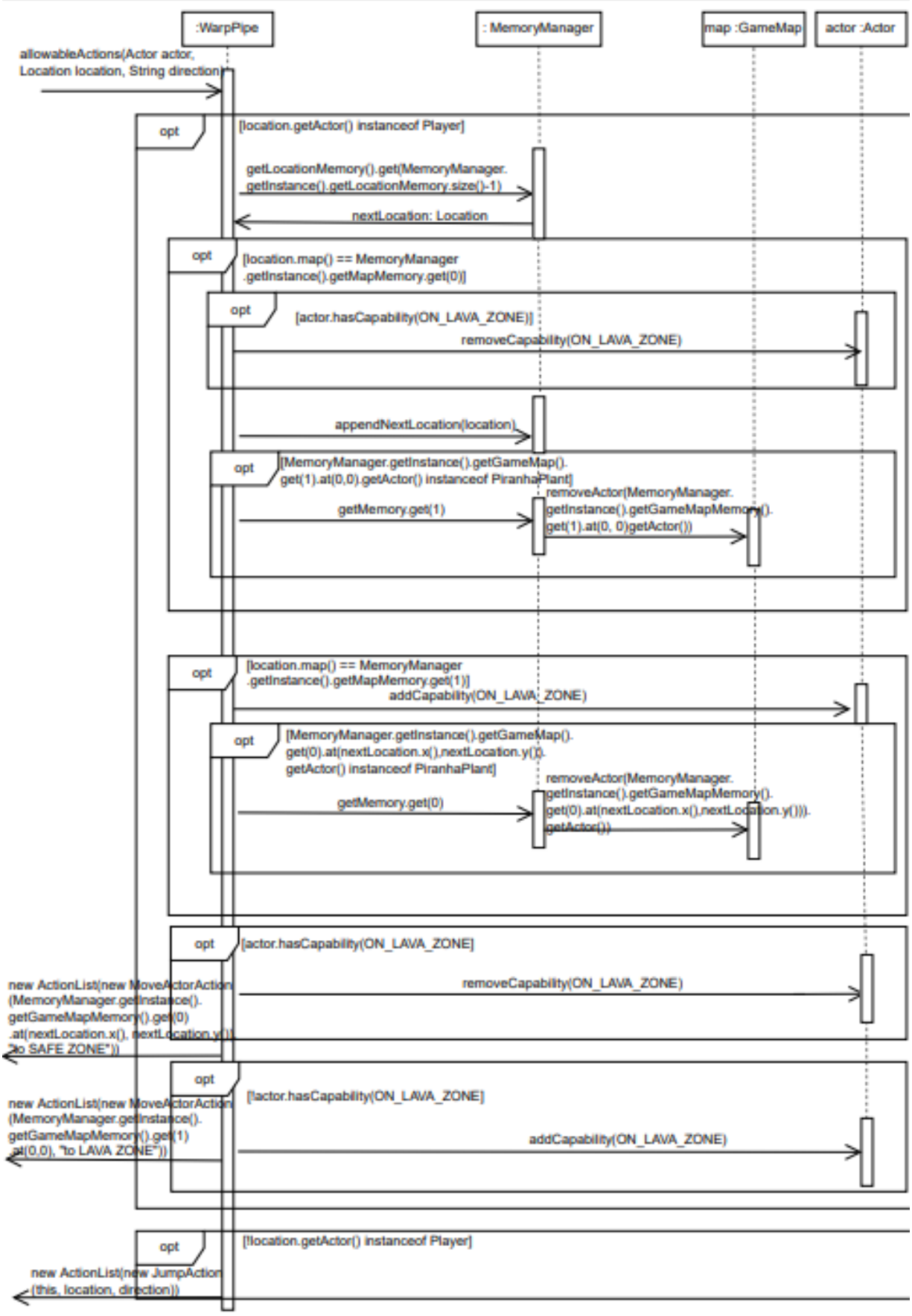


- Removed ResetTheGame item.
- Reset action will be created and added into allowable actions inside the player playturn method.

UML Class Diagram (REQ 1)



Sequence Diagram (REQ 1)



Design Rationale (REQ 1)

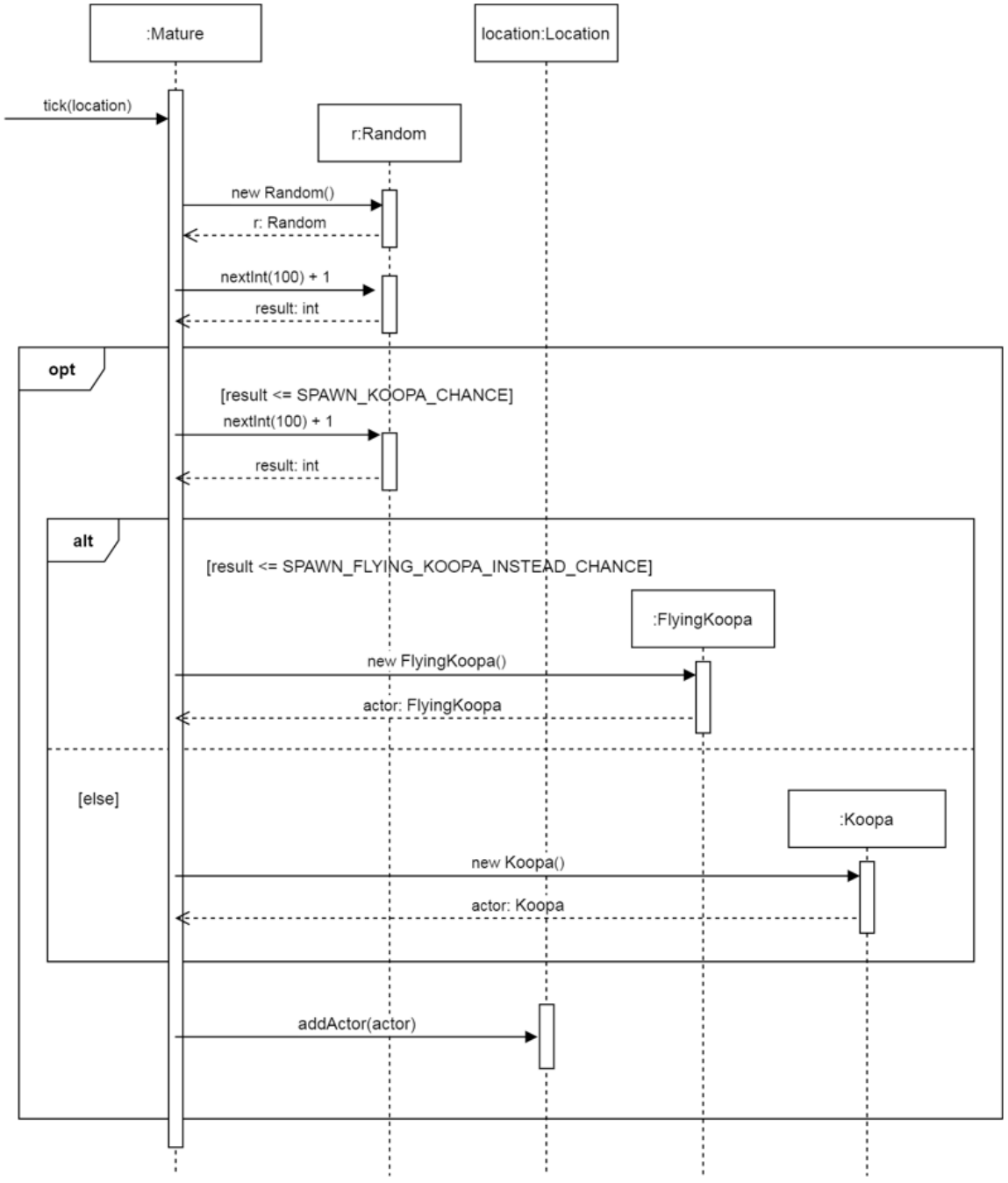
HighGround class created and extended by *WarpPipe* class, and it allows the player to jump onto it. To follow the Don't Repeat Yourself Principle, all the implementation that relates to the jump action will be implemented inside the *HighGround* class. By doing this, it will also be easy for maintenance or extension if there is another high ground object added into this game.

HighGround---<<*creates*>>---> *JumpAction*. In this game, there are different high ground objects that allow the player to perform jump action on them. Based on object-oriented, the high ground is the object. We can simply create JumpAction inside the Player class, however by doing this, we need to know which object the player wants to jump, it will require additional dependency between Player and different high ground. Besides, we also need to check whether the ground allows the player to jump onto it or not, checking the object classes using the if-else statement will also increase dependency. To align our design with the Reduce Dependency Principle, we discard this alternative. *HighGround*---<*creates*>---> *JumpAction* can avoid checking the objects that the player wants to jump using the if-else statement.

MemoryManager has two roles. One is to remember the last location that player on the current map before moving to the lava zone. So that when a player wants to move back to the safe zone, it can use the memory to bring him back to the last location of that player on the safe map before moving to the lava zone. Another role is to remember the game map that has been added into the system.

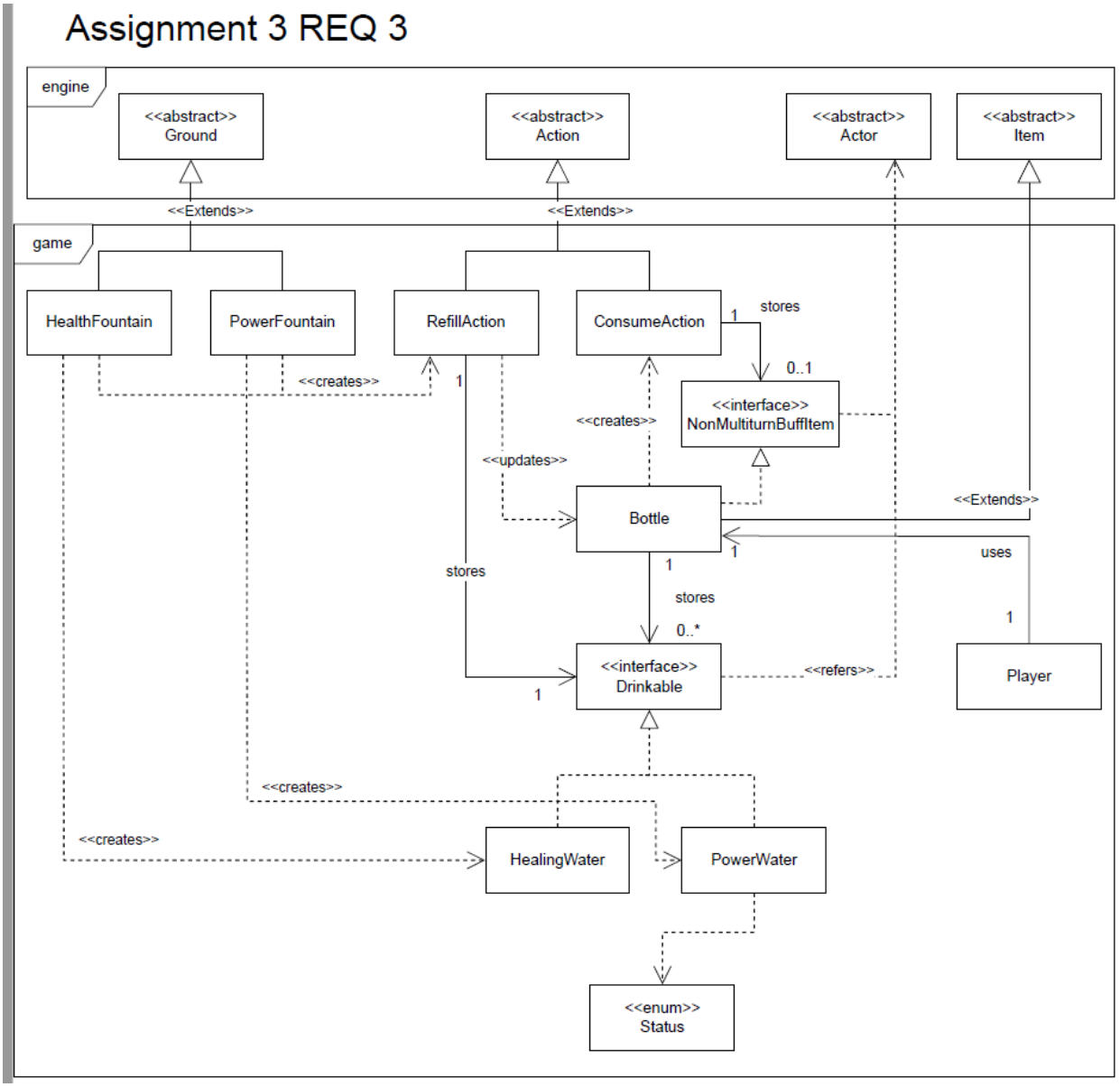
UML Class Diagram (REQ 2)

Sequence Diagram (REQ 2)

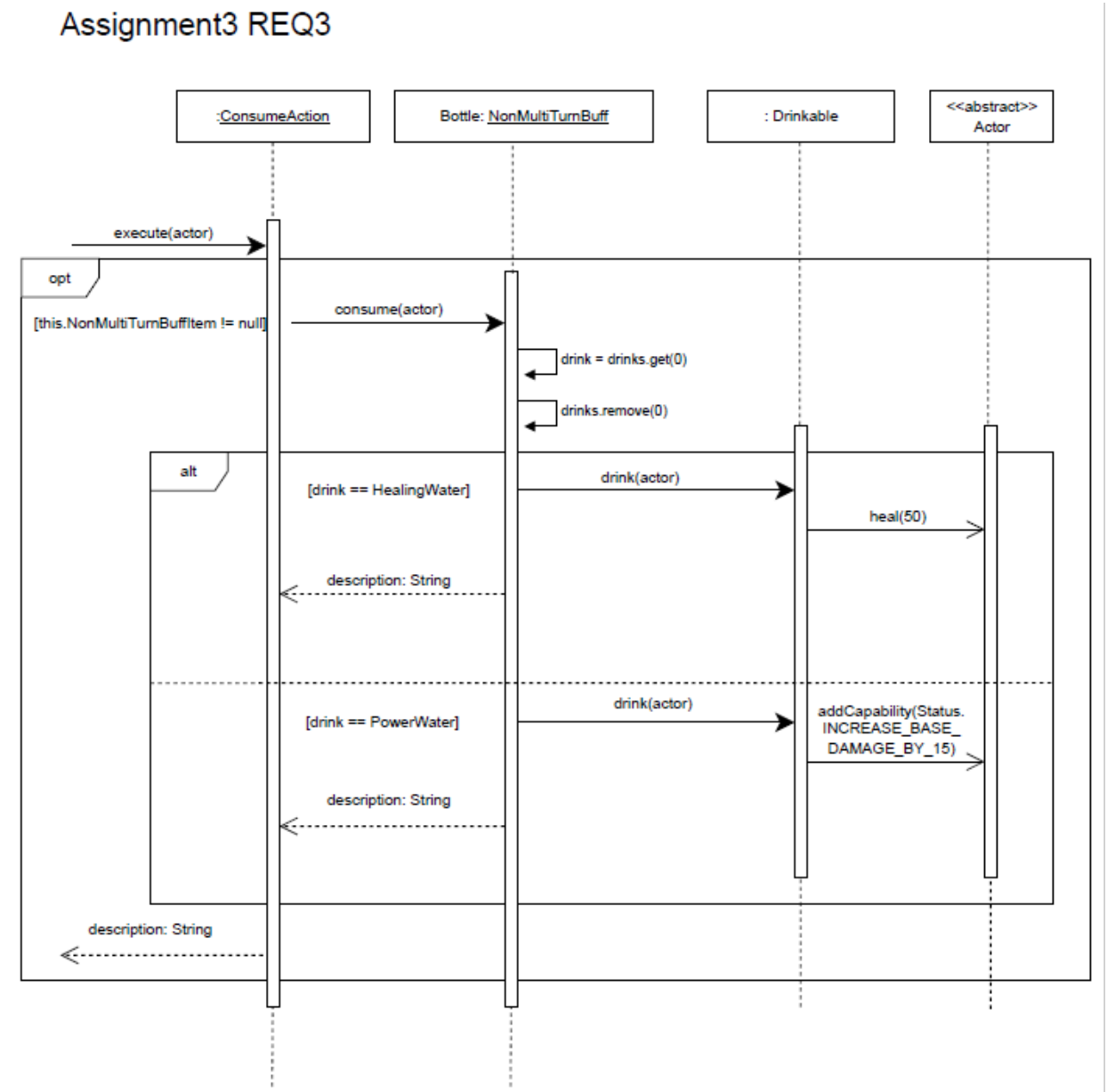


Design Rationale (REQ 2)

UML Class Diagram (REQ 3)



Sequence Diagram (REQ 3)



Design Rationale (REQ 3)

NonMultiTurnBuffitem. NonMultiTurnBuffItem interface added and consume action stores this interface. NonMultiTurnBuffItem has method to consume the item. Consume action now manages 2 items, MultiTurnBuffItem and NonMultiTurnBuffItem. The difference between both interface is that NonMultiTurnBuffItem will not be added into Consumed item manager after consumed, instead, the item will be removed after used. Since there will only be 2 types of consume action, multi turn and non multi turn buff, consume action manages both interfaces and it is not split. By doing this, only 1 class are needed which makes class design simpler.

Bottle implements NonMultiTurnBuffitem. Bottle implements NonMultiTurnBuffItem interface because all drinkable water does not have multi turn effect (healing water and power water).

Bottle stores Drinkable interface. Bottle have array list and it stores drinkable items. Drinkable has method drink and parameter actor to add the buff to the given actor.

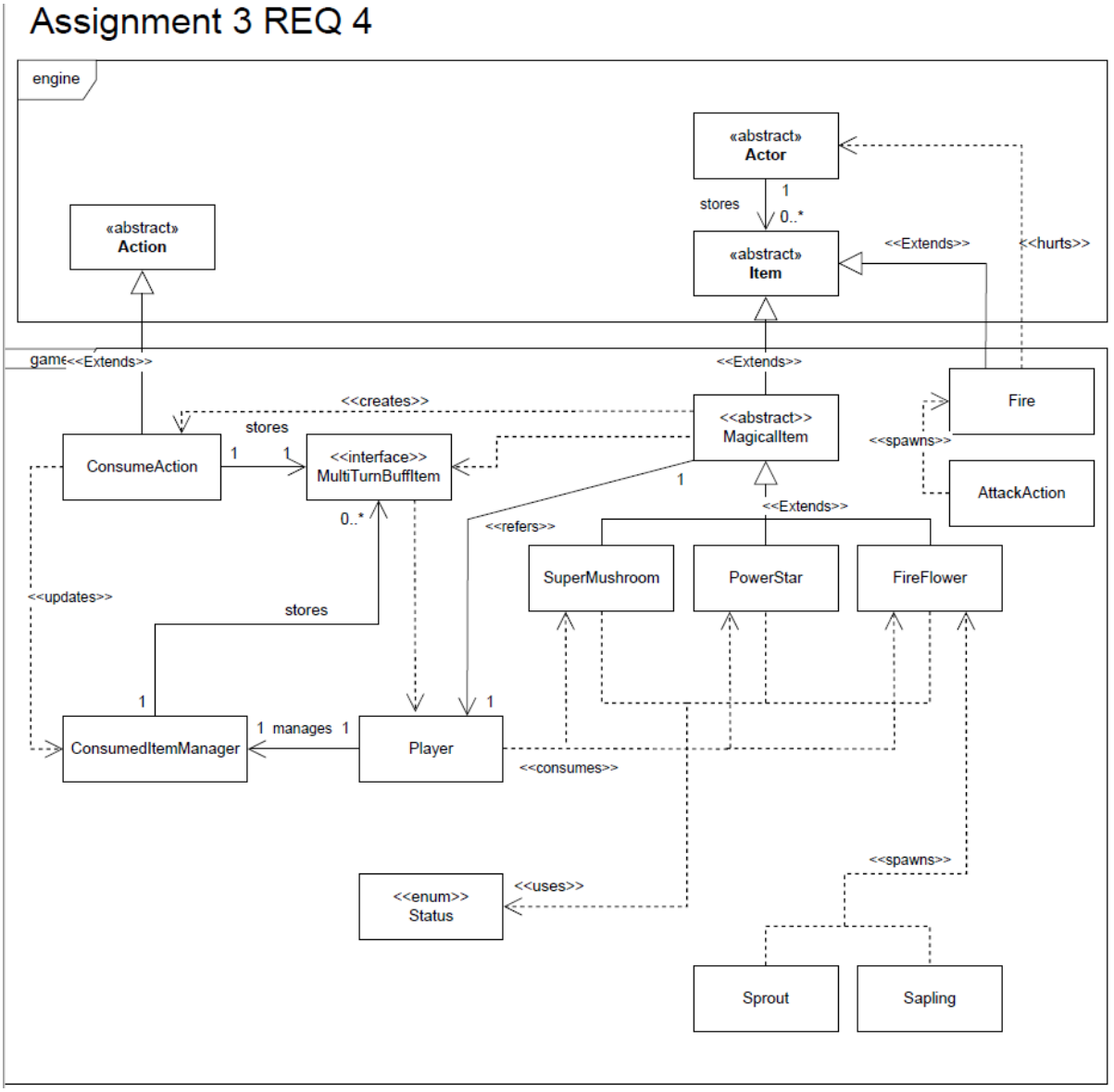
Only one bottle can exist at a time. Bottle has private constructor and getInstance method to ensure only one bottle exists. When bottle has not been created, getInstance creates new bottle and returns itself (that bottle). When bottle already exists, it returns the existing bottle which is stored as attribute.

How refill action is created and what does it store. Health fountain creates healing water and creates refill action and passes healing water as the parameter when actor is on top of the fountain. Refill action will store drinkable item. By doing so, when player performs refill action, it can directly append drinkable into bottle. This makes refill action simpler and does not need to create any new classes.

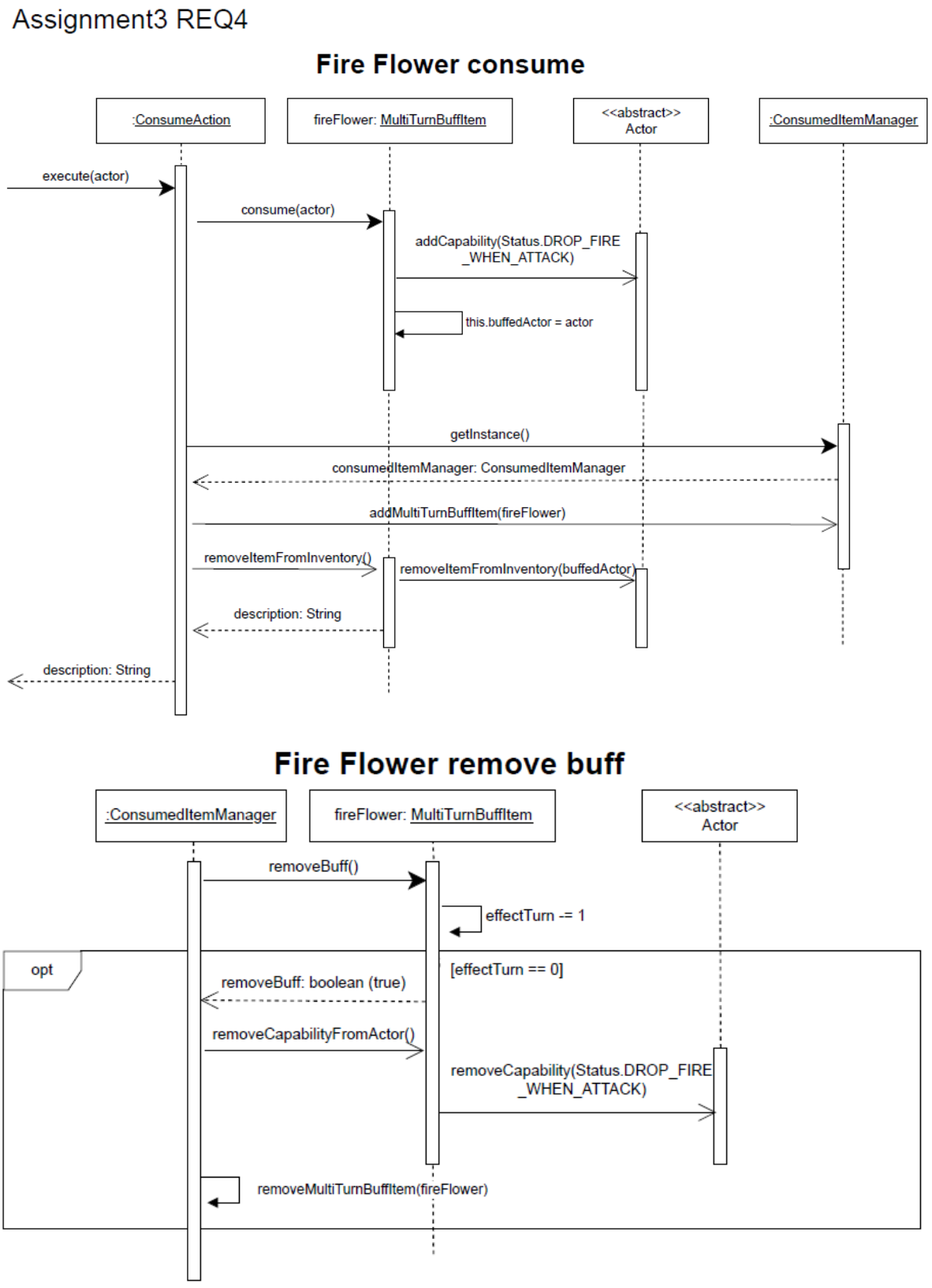
Bottle acting like a stack. Bottle stores an array as attribute which stores drinkable item. When the bottle is consumed, it takes the first item in the array and removes that item from the array. Then it calls drink method in drinkable to perform buff. To add drinkable item into bottle, add method from the array can be used to add drinkable item to the back of the array. This makes bottle act like a stack.

Increase base damage of the player. When player drinks power water, power water method adds new capability status.INCREASE_BASE_DAMAGE_BY_15 to that player. New attribute called base Damage are created in the player. When player has capability INCREASE_BASE_DAMAGE_BY_15, it will remove that capability from the player and increase base damage by 15 and it is computed inside player playTurn method every turn.

UML Class Diagram (REQ 4)



Sequence Diagram (REQ 4)



Design Rationale (REQ 4)

Changes made from previous uml:

- Added fire flower extending magical item.
- Added sprout and sapling which creates fire flower.
- Added attack action and fire item. Attack action creates fire.
- Renamed ConsumableItem into MultiTurnBuffItem.

Other changes made:

- MultiTurnBuffItem consume method now returns string. It used to return nothing.

Fire flower is considered as magical item. Since fire flower is an item and it has multi turn buff, it extends magical item and inherits item and MultiTurnBuffItem. When player consumes fire flower, it will be added into consumed item manager to manage buff effect.

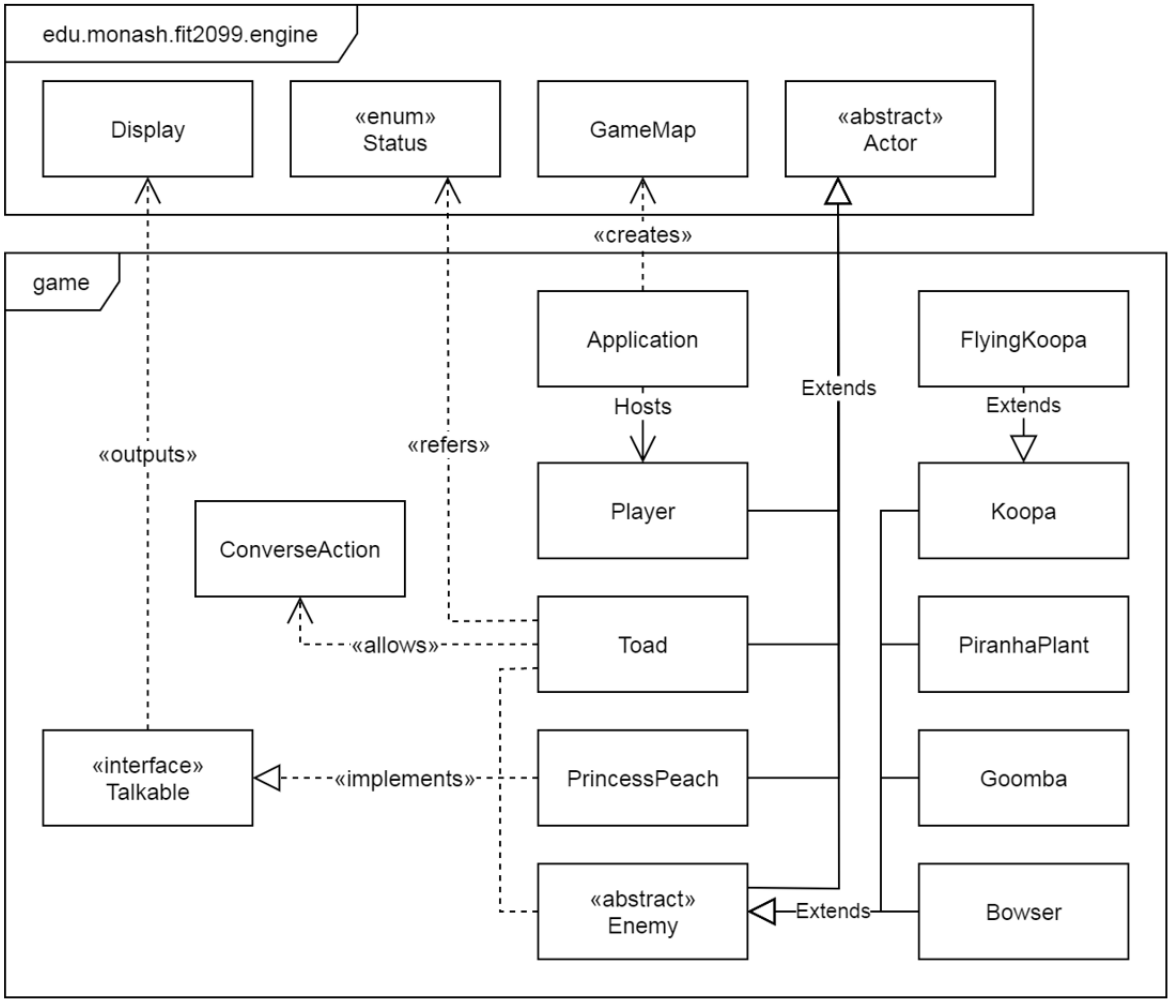
Drop fire at target location. When actor has capability drop_fire_when_attack, it creates and drops fire at target location after a successful attack. This will be done inside attack action execute method.

Spawning fire flower from sprout and sapling will be done inside tick method. If 10 turns pass, it drops fire flower at 50% chance.

When actor stands on top of fire, it hurts the actor by 20 damage. This is done inside the fire's tick(location) method. It first checks whether there is actor standing on top of the fire, if yes, it gets that actor and uses hurt method to damage the actor.

Fire will be removed from the ground after 3 turns. This is done inside tick(location) method. Integer attribute called turns is stored and it keeps track of turn. Turns will be incremented inside tick method and if turns is 3, it removes it self from the ground.

UML Class Diagram (REQ 5)



Design Rationale (REQ 5)