

FIT2099
Assignment 2
Due date: 02/05/2022 11:55 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 Teh Rong Wen (31882617)
- ☐ REQ2: by Lim Seng Onn (32200250)
- ☐ REQ3: by Ng Yen Kai (32217021)
- ☐ REQ4: by Teh Rong Wen (31882617)
- ☐ REQ5: by Lim Seng Onn (32200250)
- ☐ REQ6: by Ng Yen Kai (32217021)
- ☐ REQ7: by everyone (all 3 members in the team)

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, 8th April 2022
- ☐ Testing: Saturday, 9th April 2022
- ☐ Review: Saturday, 9th April 2022

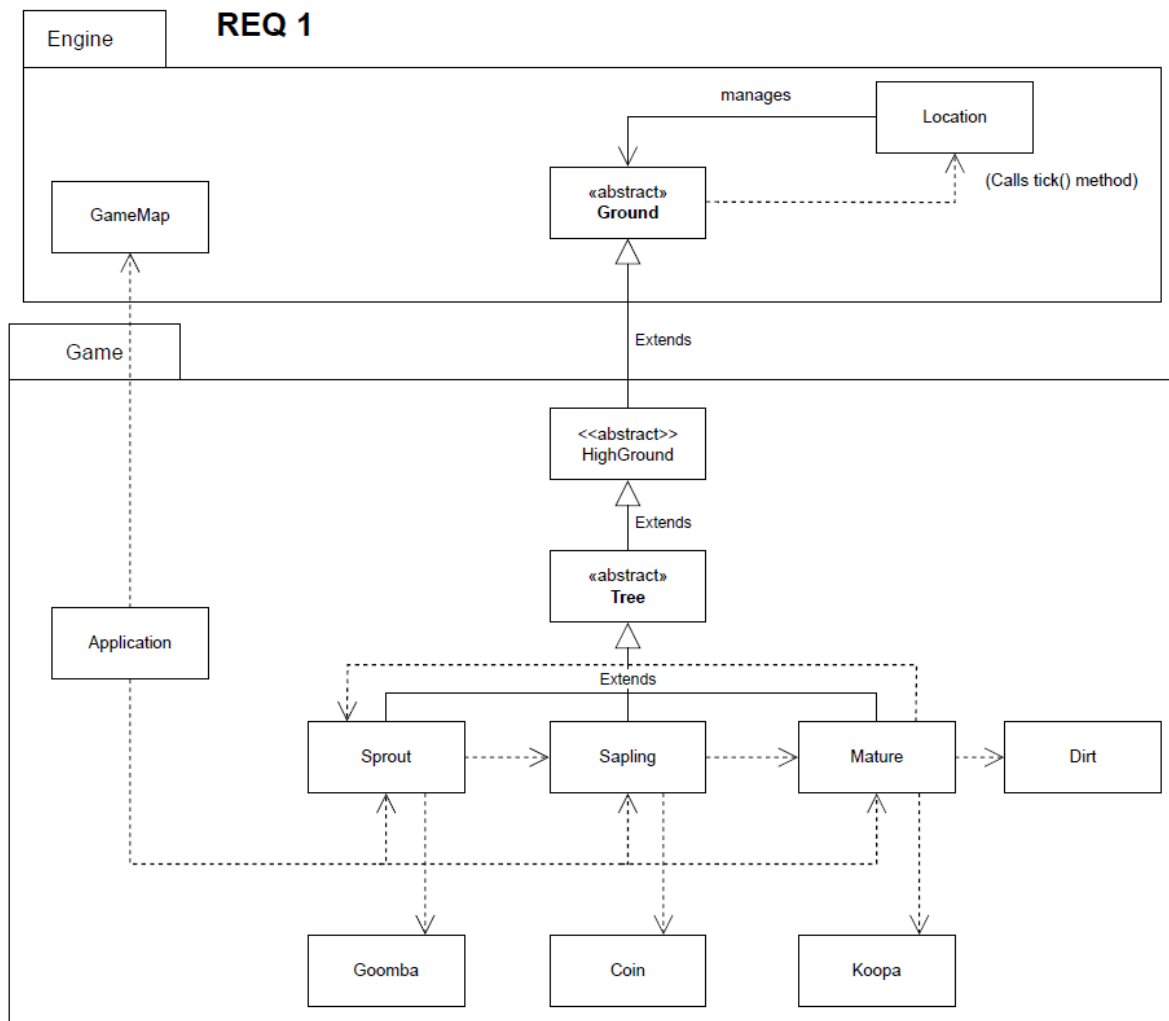
Signatures

Teh Rong Wen: I accept this WBA

Lim Seng Onn: I accept this WBA

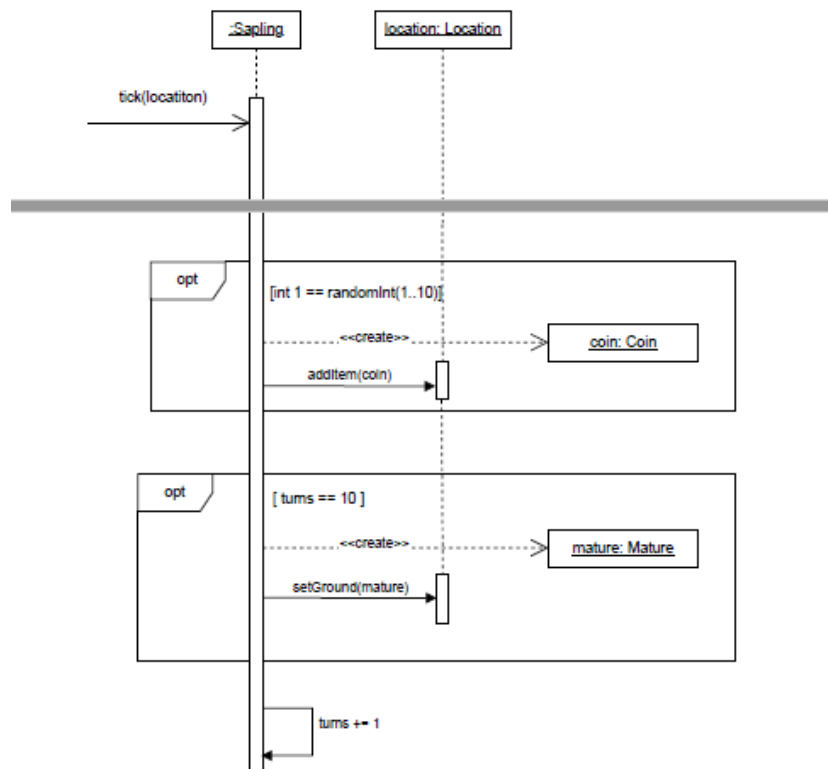
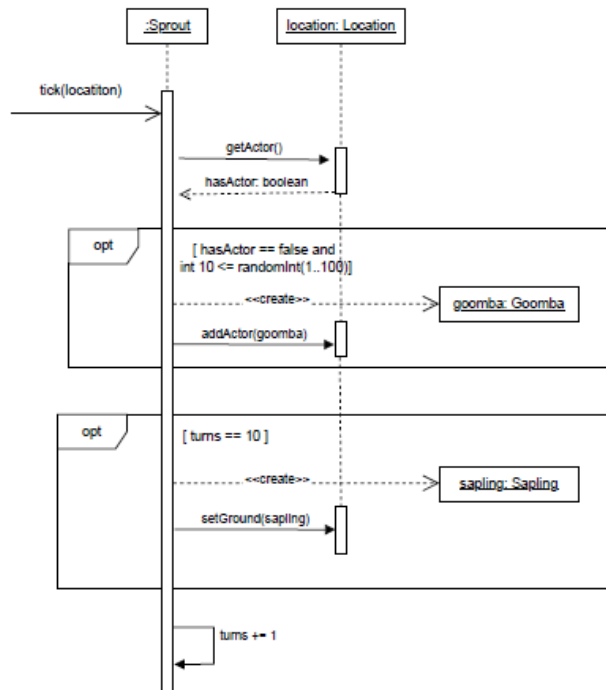
Ng Yen Kai: I accept this WBA

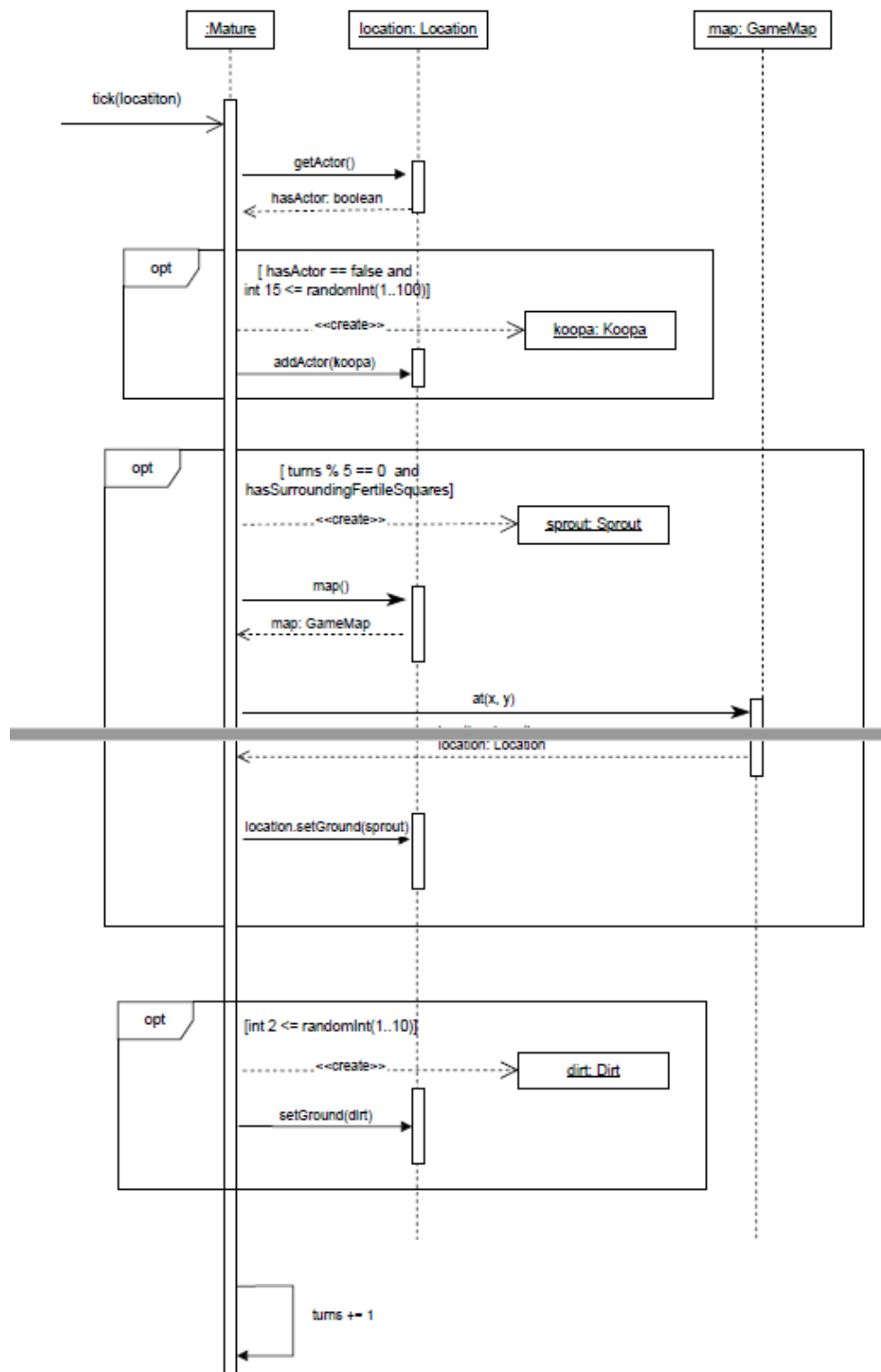
UML Class Diagram (REQ 1)



Sequence Diagram (REQ 1)

REQ1





Design Rationale (REQ 1)

Changes made: Changed symbol that extends from sprout, sapling and mature to Tree.

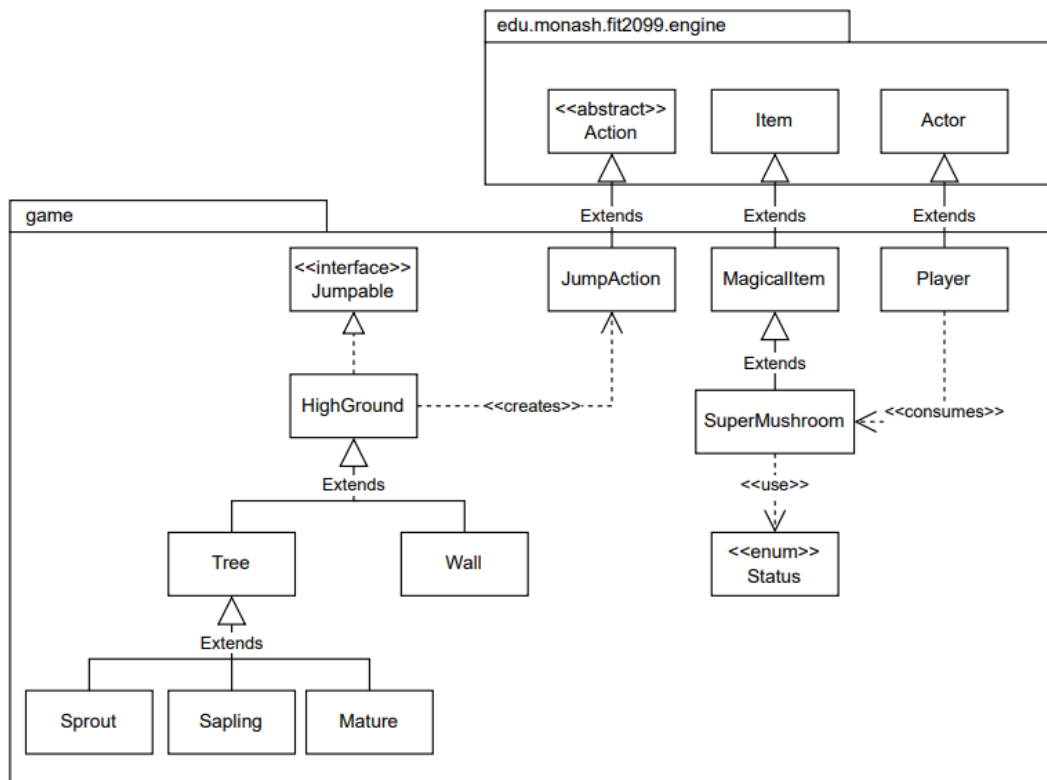
Added REQ7 reset game, Trees have 50% chance to be converted back to dirt.

REQ7 reset game, Trees have 50% chance to be converted back to dirt. By having Tree abstract class, sprout, sapling and mature no longer need to implement resettable interface. Only Tree class needs to implement the resettable interface.

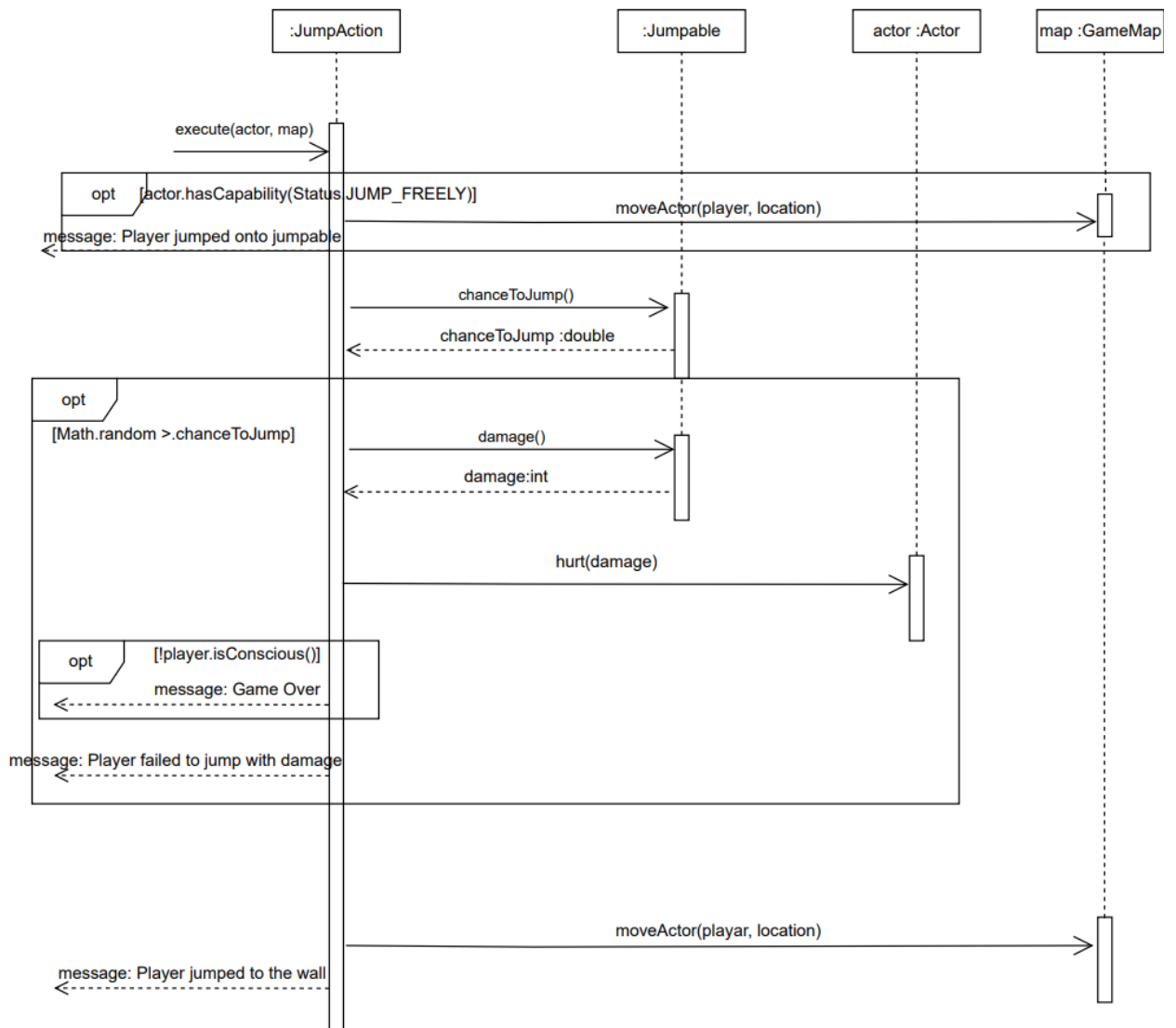
Sprout, sapling and mature extends tree. Since they are cycled and has common attribute/method such as turns(int, counter for number of turns in the map), they can be categorized as tree. If it does not extend tree abstract class, it could be hard to understand their relation (It has cycle and common method/attribute). In future implementation, it is easier to create method/attributes that they share.

Each tree subclasses requirements can be coded inside override tick(Location). It has Location as input param, so that can be used to add actor, add item, or change ground.

UML Class Diagram (REQ 2)



Sequence Diagram (REQ 2)



Design Rationale (REQ 2)

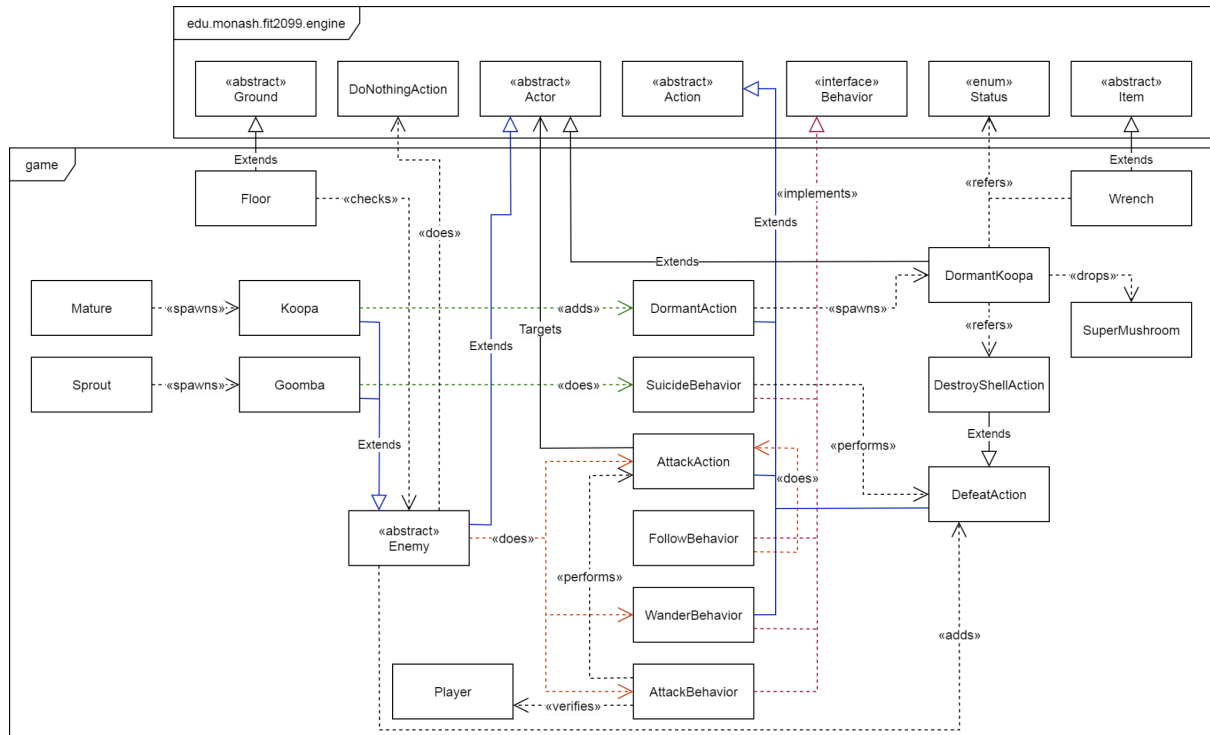
HighGround class created and extended by *Tree* class and *Wall* class. Since the different stages of the tree and wall allow the player to jump onto it. To follow the *Don't Repeat Yourself Principle*, all the implementation that relates to the jump action will be implemented inside the *HighGround* 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 such as trees and walls that allow the player to perform jump action on them. Based on object-oriented, the tree and wall are the objects. 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 *Tree* and *Wall*.

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* and *HighGround* implements *Jumpable* can avoid checking the objects that the player wants to jump using the if-else statement.

Different stages of *Tree* and *Wall* have different success rates that *Player* can jump onto it and if the player fails to jump onto it, they will cause different amounts of damage on the player. Based on the open-closed principle, classes should be open for extension but closed for modification. This is the rationale behind the presence of the *Jumpable* interface. By doing this, the classes implementing it will have the same method but different implementation, even though there is another high ground that will be added into our game in future.

UML Class Diagram (REQ 3)



Design Rationale (REQ 3)

Per the requirements, enemies may not enter Floor as part of any of their actions. Thus, a new layer of abstraction `Enemy` was added, so Floor would not have to depend on each individual enemy in its checks for `canActorEnter`, creating high substitutability and lower coupling. The additional `Enemy` inheritance layer also helps for code reuse, as many of the enemy classes share the same features.

We also refactor the `target.isConscious()` conditional in `AttackAction` into its own class, `DefeatAction`. The first reason is that there is no reason for the attacker that called the `Action` to be the one to handle the logic for handling unconscious actors, breaking segregation. The second is that `AttackAction` should not be responsible for this to begin with, breaking single responsibility on top of its now misleading name. The third, used for the requirement, is that it allows for us to create a behavioral override to what happens to an actor when it is unconscious.

In its place, we might create a behavior which conditionally executes `DefeatAction`, a behavior which all relevant `Actor` subclasses should populate their behaviors with in their constructors. There is still a significant issue with this method - there is a time lag between an actor being killed and it being considered dead. This is due to a limitation with the engine - there isn't a good way to force a `playTurn` event for an arbitrary actor.

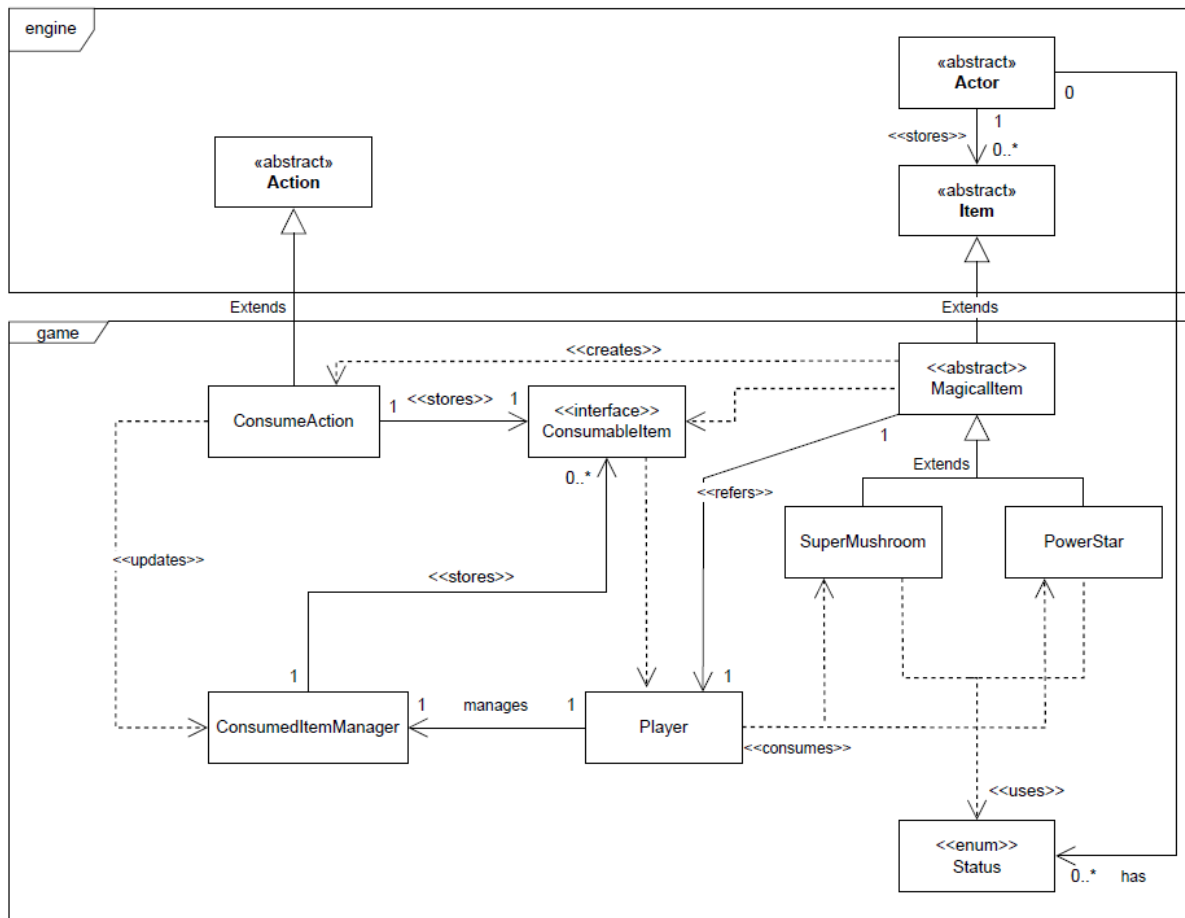
The solution ultimately used is to keep the resultant `Action` inside `AttackAction`'s attributes, such as `DefeatAction` or `DormantAction`, and have the correct action to perform on target's death be passed into `AttackAction` through its constructor. This would result in the need to pass in the action on defeat to all actions potentially defeating an actor, but allows only the `Enemy` subclasses overriding the on-defeat action to have the need for modification.

By having the `Enemy` control its own behavior on defeat, we would be able to override the default behavior, and replace it with ie. `DormantAction` for `Koopa`s with minimal coupling. The `DefeatAction` refactor also allows us to create `SuicideBehavior`, extending `DefeatAction`, which calls its super conditionally. This may be undesirable if the goal is to delete `Goomba` without having it drop its items, but it should be easy to override its `execute()` to eg. delete its inventory before calling `super()` on itself.

We create a new `DormantKoopa` class in addition to `Koopa`, because both actors have vastly different behaviors, most of which are inherited from `Enemy` in `Koopa`, so it would be easier to think of them as separate entities. The `SuperMushroom` drop is also kept inside the dormant `koopa`, dropped upon defeat by `DestroyShellAction`, which refers to `DefeatAction` with a different menu description.

UML Class Diagram (REQ 4)

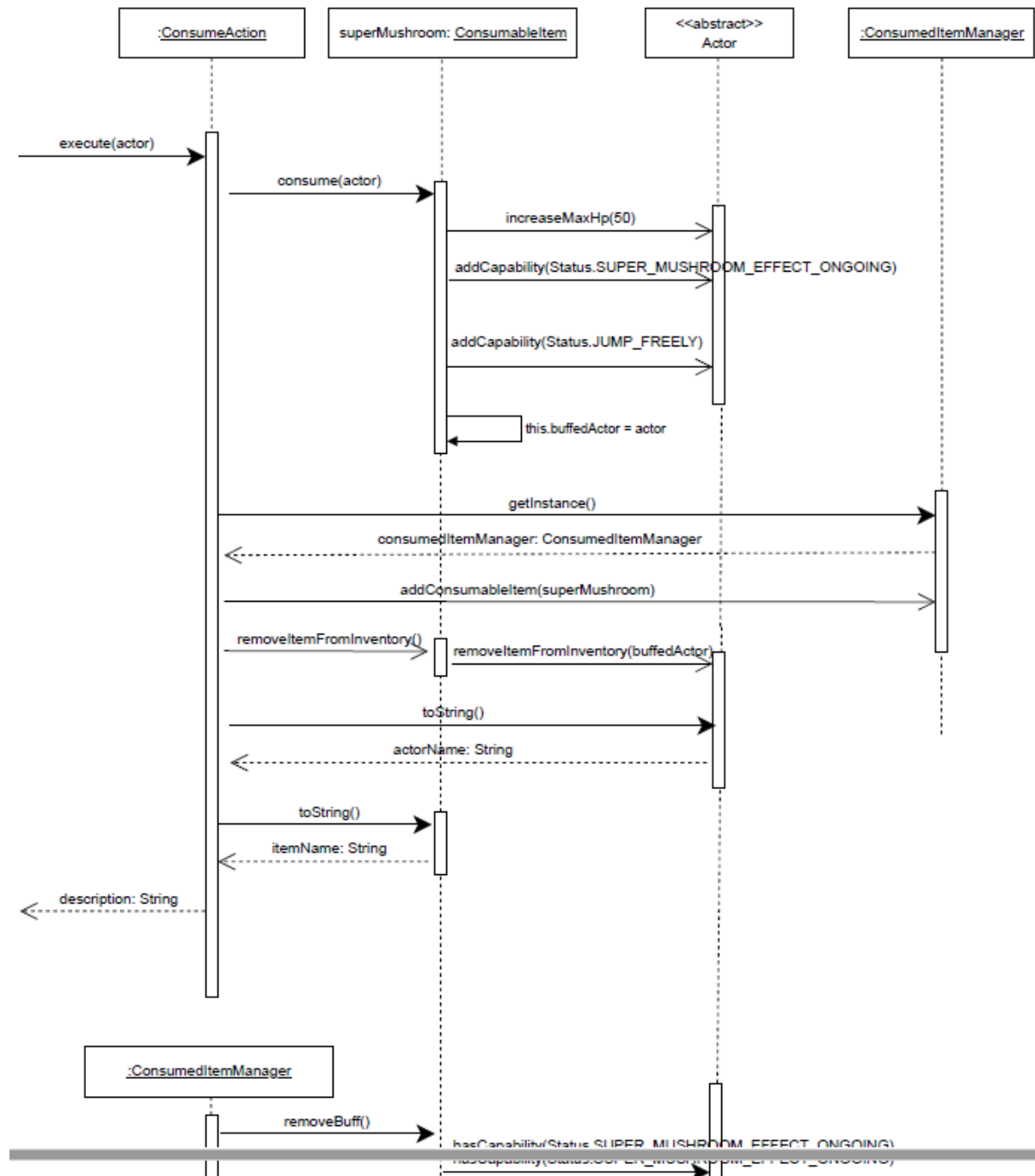
REQ 4

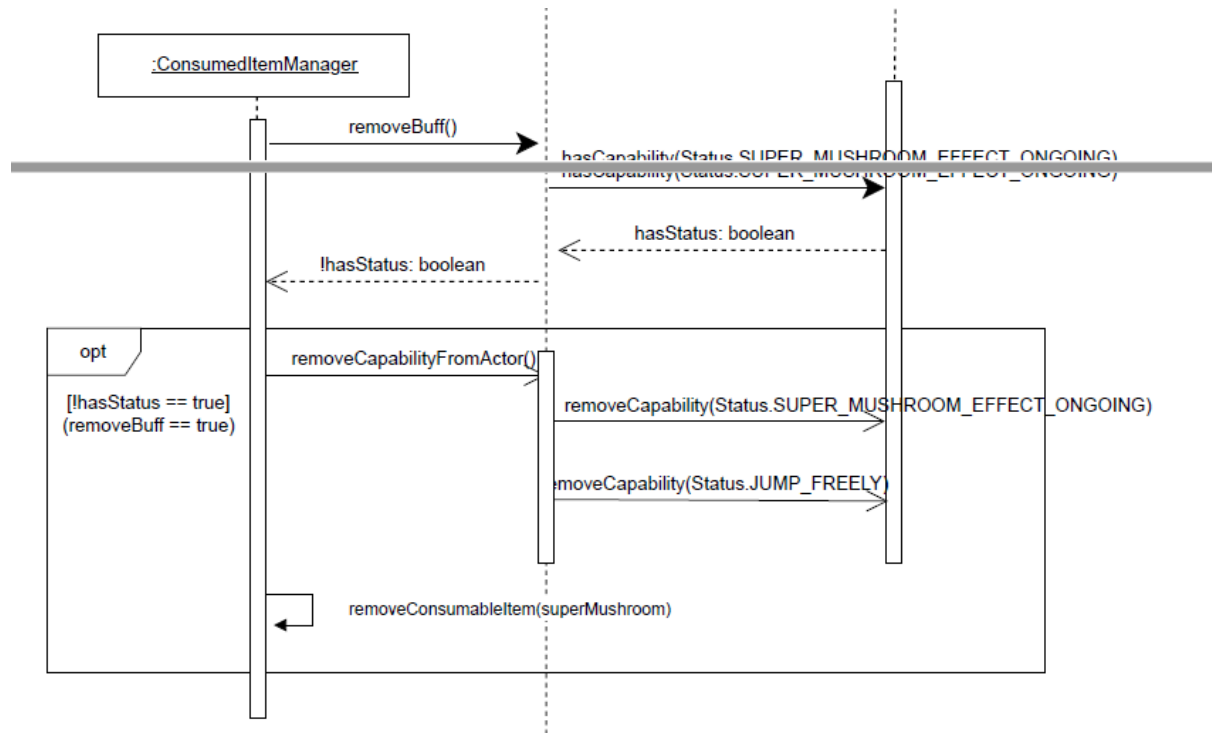


Sequence Diagram (REQ 4)

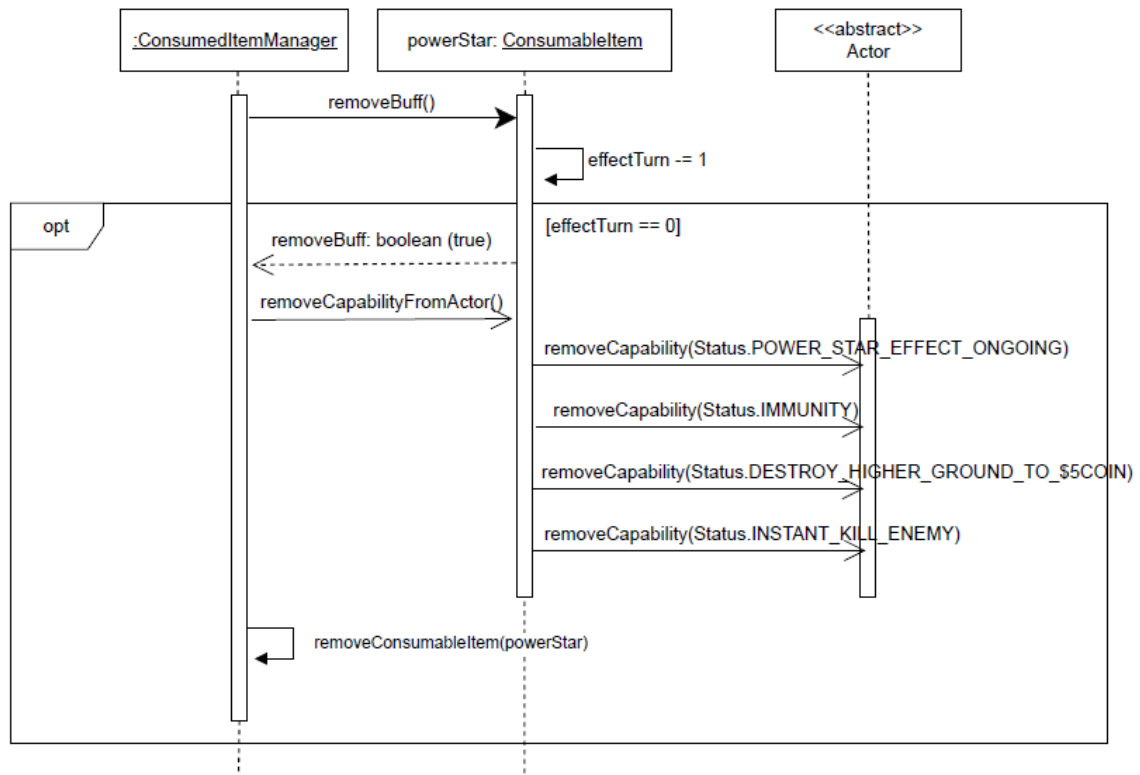
REQ4

Super Mushroom





Power Star



Note:

execute(actor) for Power Star will be similar with execute(actor) for Super Mushroom shown above.
Did not include it because it is repetitive/similar.

Design Rationale (REQ 4)

Changes made inside uml:

- **Added magical item abstract class, consumable item interface, consume action, consumed item manager.**

Changes made inside design rationale:

- **Removed 1 action for 1 consumable item and added 1 action class needed for multiple consumable items.**
- **Items that were consumed no longer stays in actor inventory. Consumed items will be stored inside consumed item manager.**
- **Player will store the instance of Consumed item manager.**

1 action class needed for multiple consumable items. When the player has consumable item in the inventory, the player has option to consume action. For the new design, when item returns new consume action to the player allowable actions, it passes itself (consumable item) into consume action. Since consume action stores instance of consumable item, it can execute buffs of the item to actor by calling consumable item interface method. The buffs are then added to the actor inside the item class (such as super mushroom, power star). By doing this, it no longer needs to create 1 action for 1 consumable item.

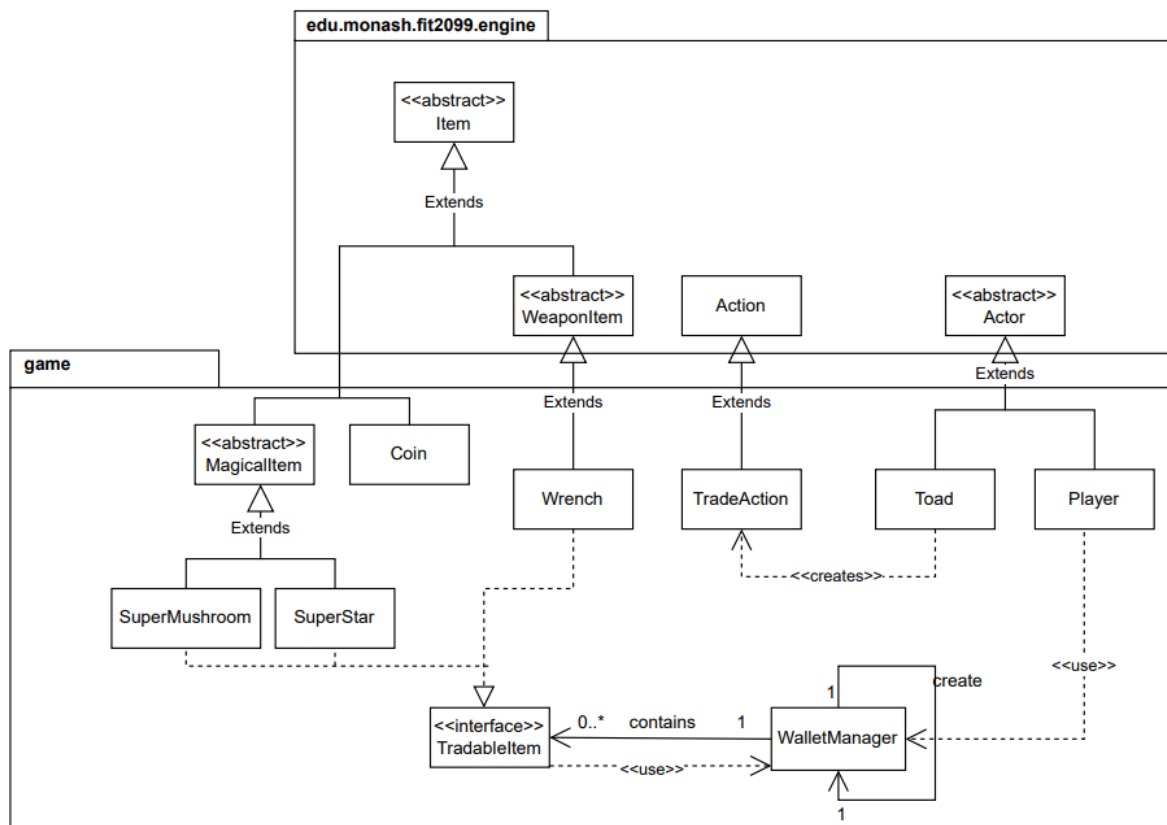
Consumed item manager. When the actor consumes the consumable item, consume action will add consumable item into consume item manager and remove item from actor's inventory. By doing this, inventory will no longer contain item that is consumed which makes the design cleaner and easier to manage.

Player will store the instance of Consumed item manager. Our assumption is that only the player can consume items. So, player stores consumed item manager and manages item buffs by calling a ticker for consumed item manager every round.

Magical item implements consumable item interface. Magical item abstract is created and implements consumable item interface. If we want to create new item that implements consumable item, that new item can just inherit magical item abstract class. When we want to add 10 more consumable item, each of them does not need to implement consumable item interface (reduces work).

Consume action and consumable item interface. Any item that implements consumable item interface should have buff effect which runs for several turns until condition has met. Consume action will store the consumable item into the consumed item manager because consumable item will always have buff effect and consume action manages only the item that has buff effect. (Which should be stored inside the consumed item manager to check if buffed effects should be removed every turn)

UML Class Diagram (REQ 5)



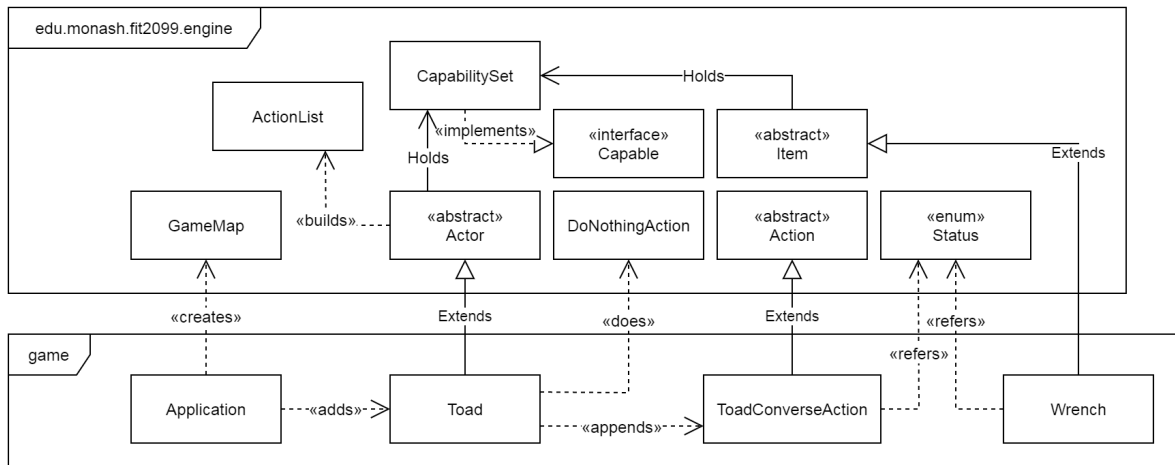
Design Rationale (REQ 5)

`<<interface>>` *TradableItem* implemented by SuperMushroom, PowerStar and Wrench. To follow *open-closed principle*, these three items share the same method but different implementations. By using an interface as an abstraction, it can open for extension of the same method but different implementation.

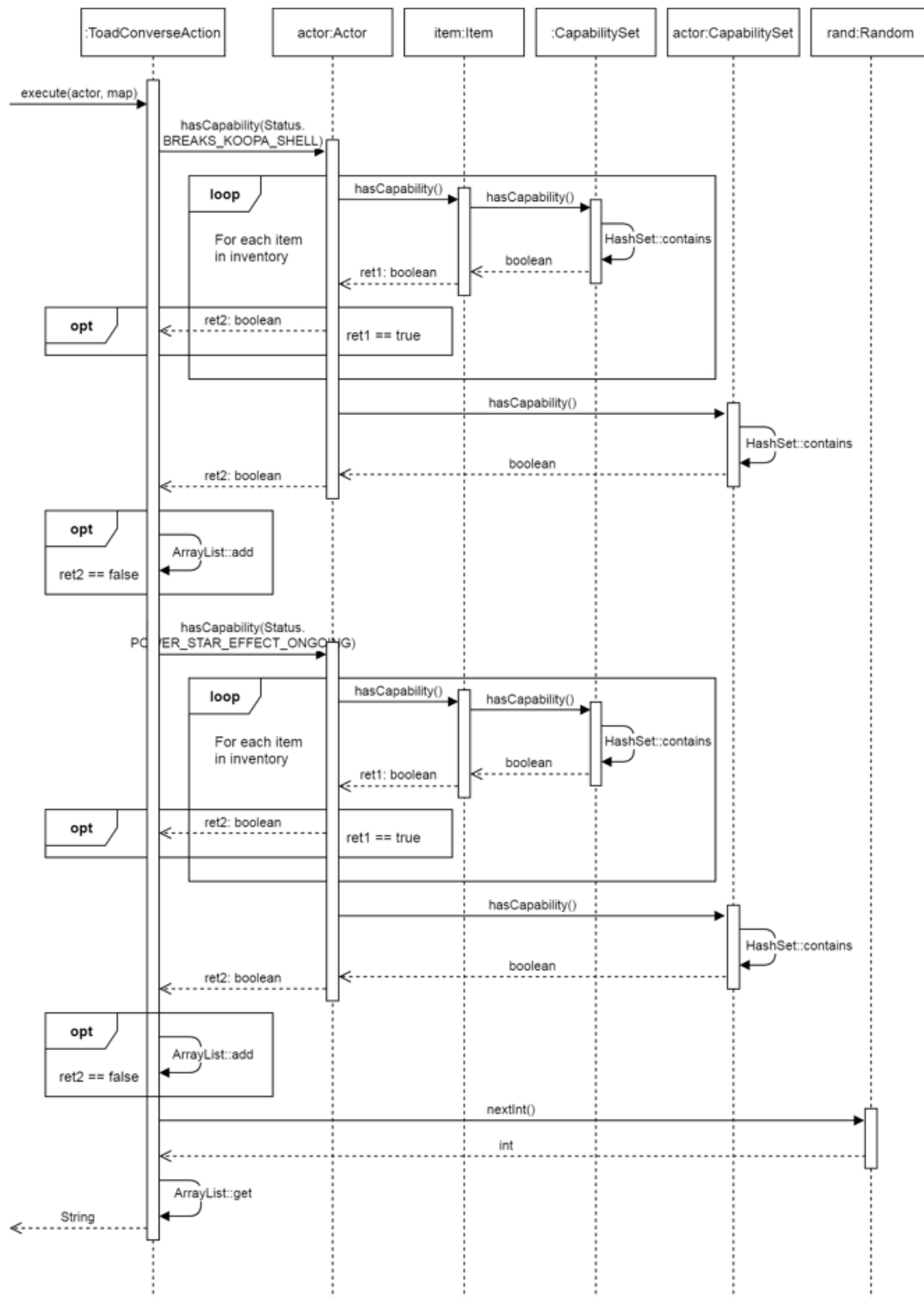
TradeAction class is created and *Toad*---`<<creates>>`---*TradeAction*. In this action, it simply checks the player's wallet balance whether sufficient to buy the corresponding item. There are only 3 items that can be sold to the player which are wrench, powerstar and supermushroom. Once the player buys a powerstar or supermushroom from Toad, these items cannot be dropped by the player. To achieve this requirement, whenever the player buys one of these two items from Toad, a new instance on these items' class which has the attribute `portable=false` will be created and added to the player's inventory. The reason for doing this is because any item that is not portable will not allow pick up and drop action.

WalletManager has a list of tradable items that were bought by the player. It is a static class with a private constructor. The reason for doing this is because we only want an instance of WalletManager and the only way to get the instance is using the `getInstance()` method. WalletManager allows us to keep track with item bought by player and deduct the price of item from its wallet balance

UML Class Diagram (REQ 6)



Sequence Diagram (REQ 6)



Design Rationale (REQ 6)

For this requirement, we follow the conventions already laid out in the other game classes, and we conform to the design decisions made in the other requirements.

We create a new Toad class, extending Actor, as had been done in requirement 5, as the toad acts very much like an actor with no behavior (self-initiated actions), which we could just leave as a no-op with the DoNothingAction class, and otherwise functions very much like any other NPC. Thus, it is best implemented as a subclass of Actor for reusability and substitutability.

We add a new (subclass of) Action to the list of allowableActions in Toad, named ToadConverseAction. This action adds the "talk" feature to toad, and its addition into the list adds another option with which to interact with the actor.

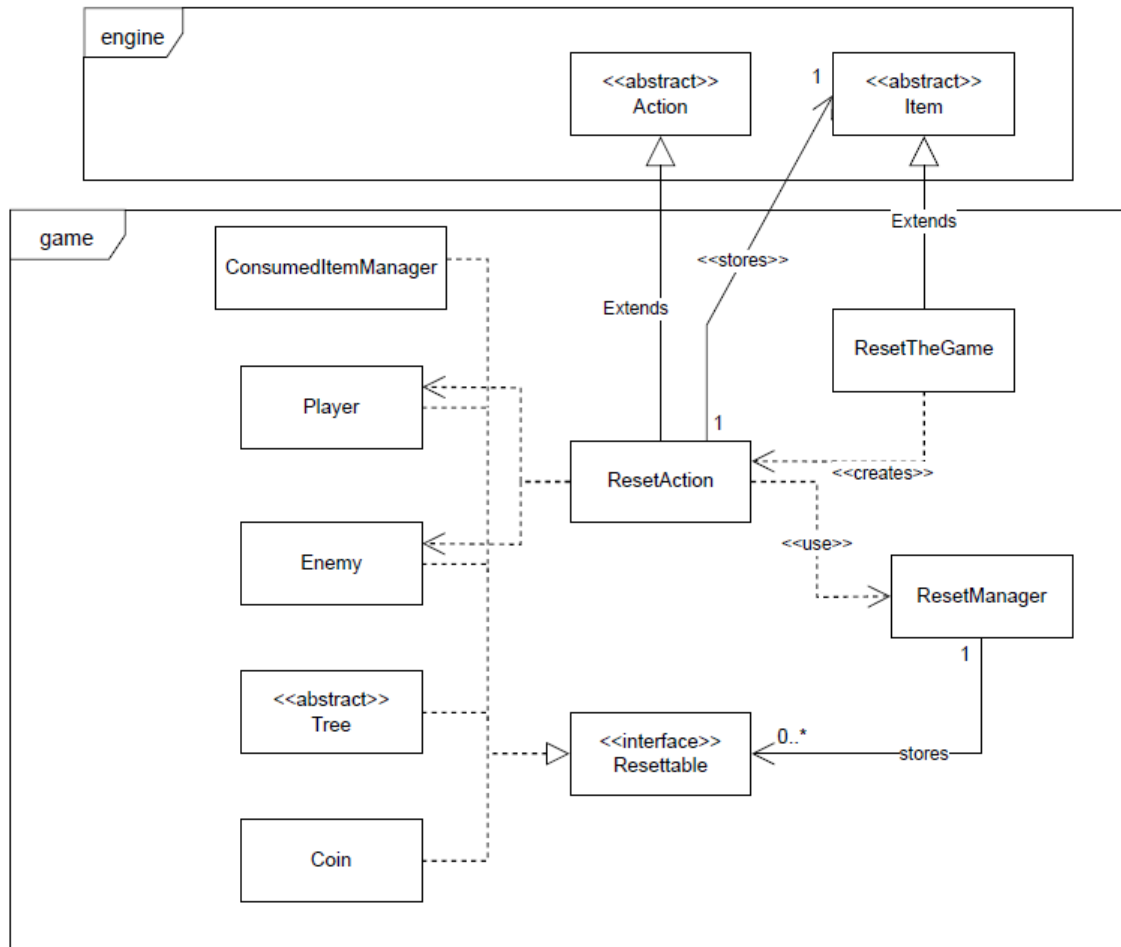
The action is not made more general, ie. as ConverseAction, because many of the conditional logic and dialog strings are not reusable. We also do not create another inheritance layer or interface as an abstraction for dialog-only actions, because it would not allow for much additional code reuse in terms of raw logic, and would not decrease coupling any further, as toad would still need a direct reference to its own ToadConverseAction, and there is no justifiable need to iterate between dialog-related actions in any class for the foreseeable future.

The toad character is then added to the game through an addActor call to GameMap inside Application. The call is done in the aforementioned class, because Toad is a unique NPC with a lifetime that lasts for the entire length of the game. Thus, there is no need for any more specific class to manage the one Toad instance created at the start of the game. There may be need in the future to refactor the initial population of actors into its own manager class, for further logic, or multiple toad merchants in different areas or maps for in-game accessibility, but it should not be difficult as only the change of a single, atomic line in Application is necessary.

We also create a new Status value for the special effect of Wrench to destroy Koopa shells, used not only for the DestroyShellAction from requirement 3, but also for one of the checks inside ToadConverseAction. An alternative for both cases is to iterate through the Actor's inventory and compare each Item to an instance of Wrench. This requires a lot more logic and code to perform, does not reuse the hasCapability of Actor.

UML Class Diagram (REQ 7)

REQ7



Design Rationale (REQ 7)

We reuse the provided Resettable interface and ResetManager singleton class to implement the partial game reset feature. All classes relevant to the reset implement Resettable. This includes Player itself, the one who initiated the action, who would reset his own effects and restore to full health.

Since we have already made the classes Coin and Enemy from other requirements, we can just have them implement Resettable instead, reducing connascence with other enemy subclasses, or the parent Item class, especially since other items in the world are not affected.

We then add a ResetAction extending Action to implement the hotkey override feature and present a menu prompt. It then executes run() in ResetManager to loop through all the stored Resettables to execute their relevant methods.

Inside Player's ResetInstance, we also set a flag in player's attributes to prevent the ResetAction from showing up in its allowableActions list again, through a conditional check for this flag in Player's allowableActions override.

Inside Tree, we store its Location as an attribute on a call to tick(), instead of setting a flag to reset the Ground it is standing on to Dirt inside tick(). This way, we get to reference Location and do the logic inside resetInstance instead.

This prevents a race condition where the tree may stick around for an extra turn if tick() somehow ran before resetInstance gets to set the relevant flag, if the logic surrounding the calls to both magic were to be changed in the future. This creates a smaller connascence of timing by omitting the need to keep any of the two methods in mind when modifying the calling logic to the other.

