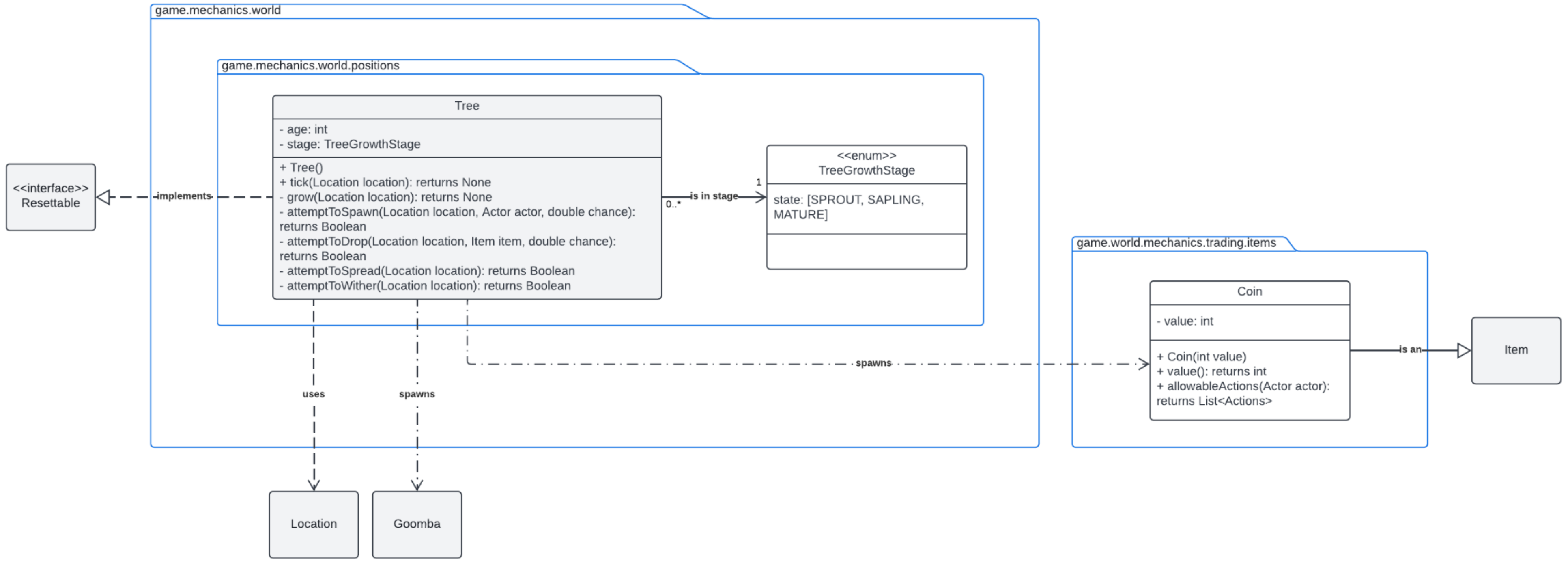
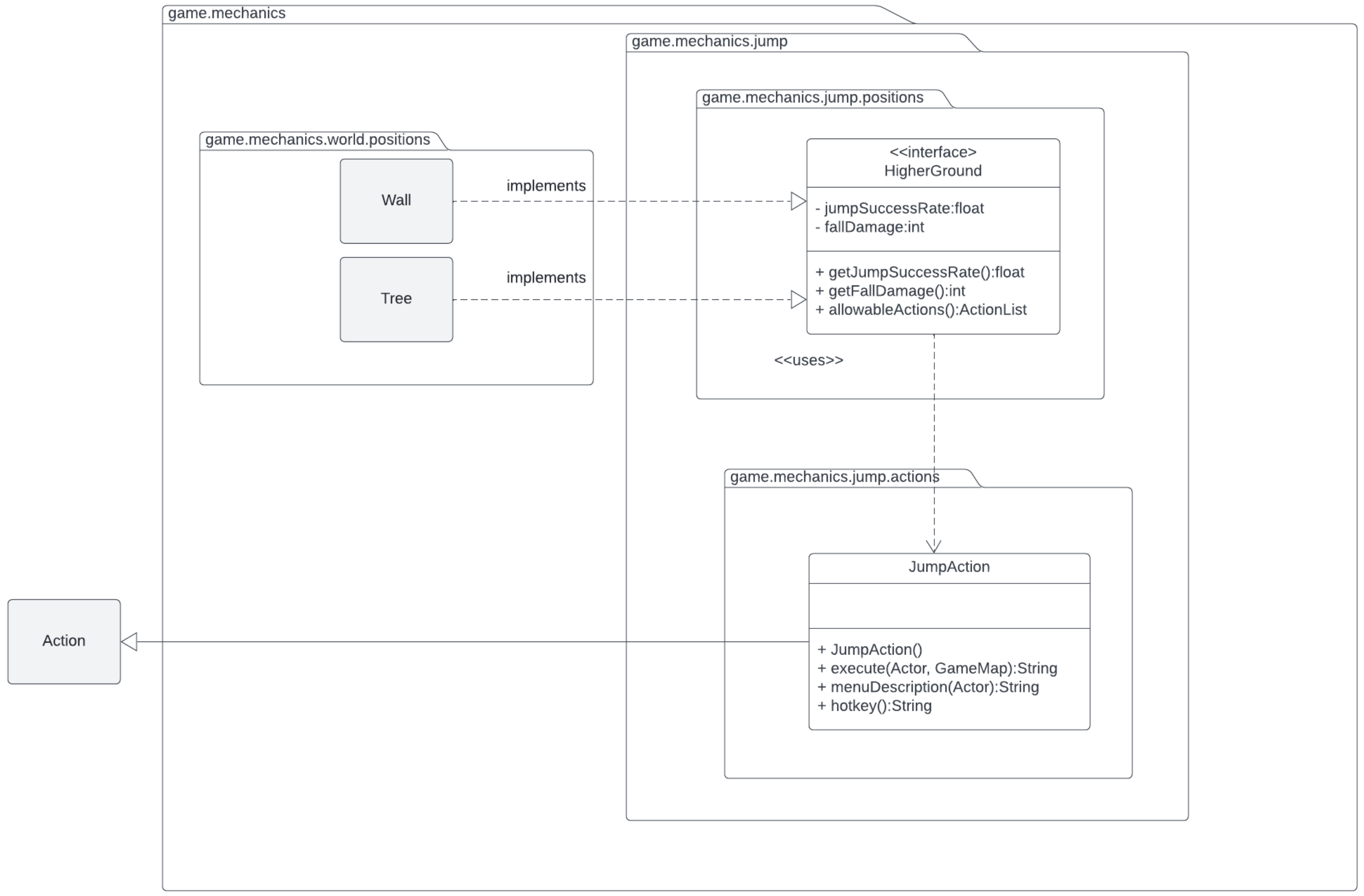
Assignment 1

1. Class Diagrams

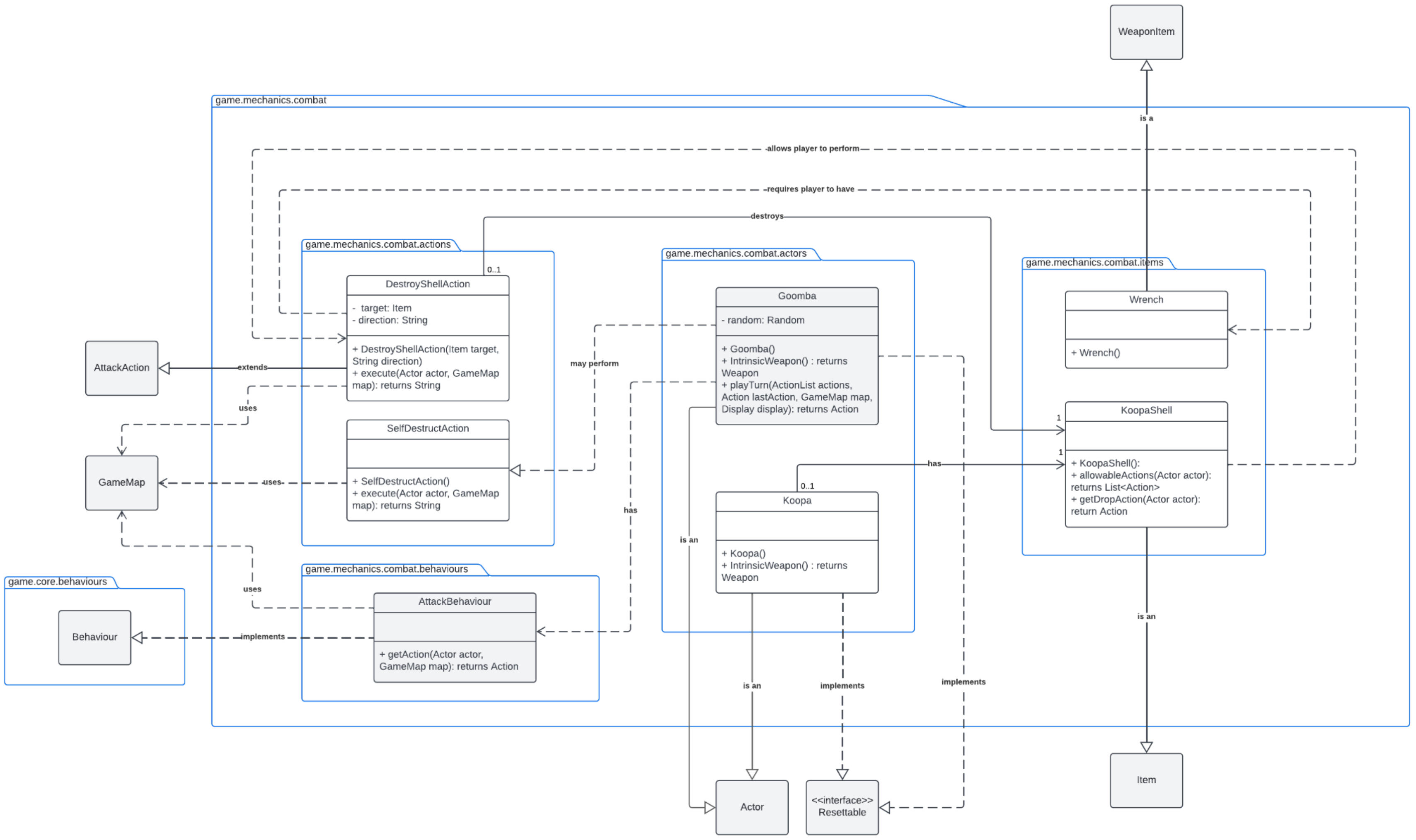
1.1 Requirement 1 - Trees


game.mechanics.world
game.mechanics.world.positions
Tree
- age: int
- stage: TreeGrowthStage
+ Tree()
+ tick(Location location): rerturns None
- grow(Location location): returns None
- attemptToSpawn(Location location, Actor actor, double chance): returns Boolean
- attemptToDrop(Location location, Item item, double chance): returns Boolean
- attemptToSpread(Location location): returns Boolean
- attemptToWither(Location location): returns Boolean
<<enum>> TreeGrowthStage
state: [SPROUT, SAPLING, MATURE]
is in stage
0..*
1
<<interface>> Resettable
implements
uses
spawns
spawns
Location
Goomba
game.world.mechanics.trading.items
Coin
- value: int
+ Coin(int value)
+ value(): returns int
+ allowableActions(Actor actor): returns List<Actions>
is an
Item

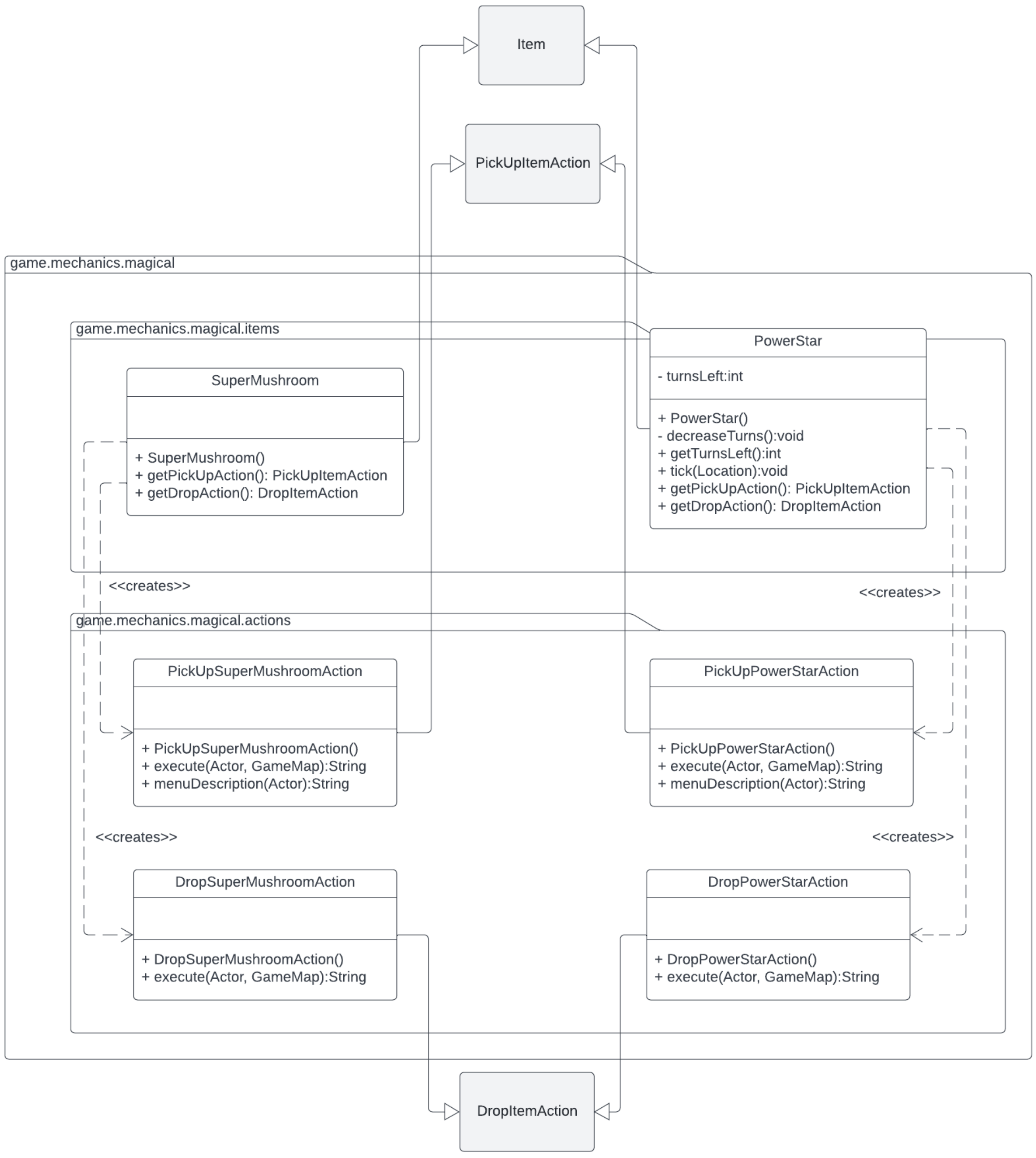
1.2 Requirement 2 - Jumping


game.mechanics
game.mechanics.jump
game.mechanics.jump.positions
game.mechanics.world.positions
Wall
Tree
implements
implements
<<interface>>
HigherGround
- jumpSuccessRate:float
- fallDamage:int
+ getJumpSuccessRate():float
+ getFallDamage():int
+ allowableActions():ActionList
<<uses>>
game.mechanics.jump.actions
JumpAction
+ JumpAction()
+ execute(Actor, GameMap):String
+ menuDescription(Actor):String
+ hotkey():String
Action

1.3 Requirement 3 - Combat

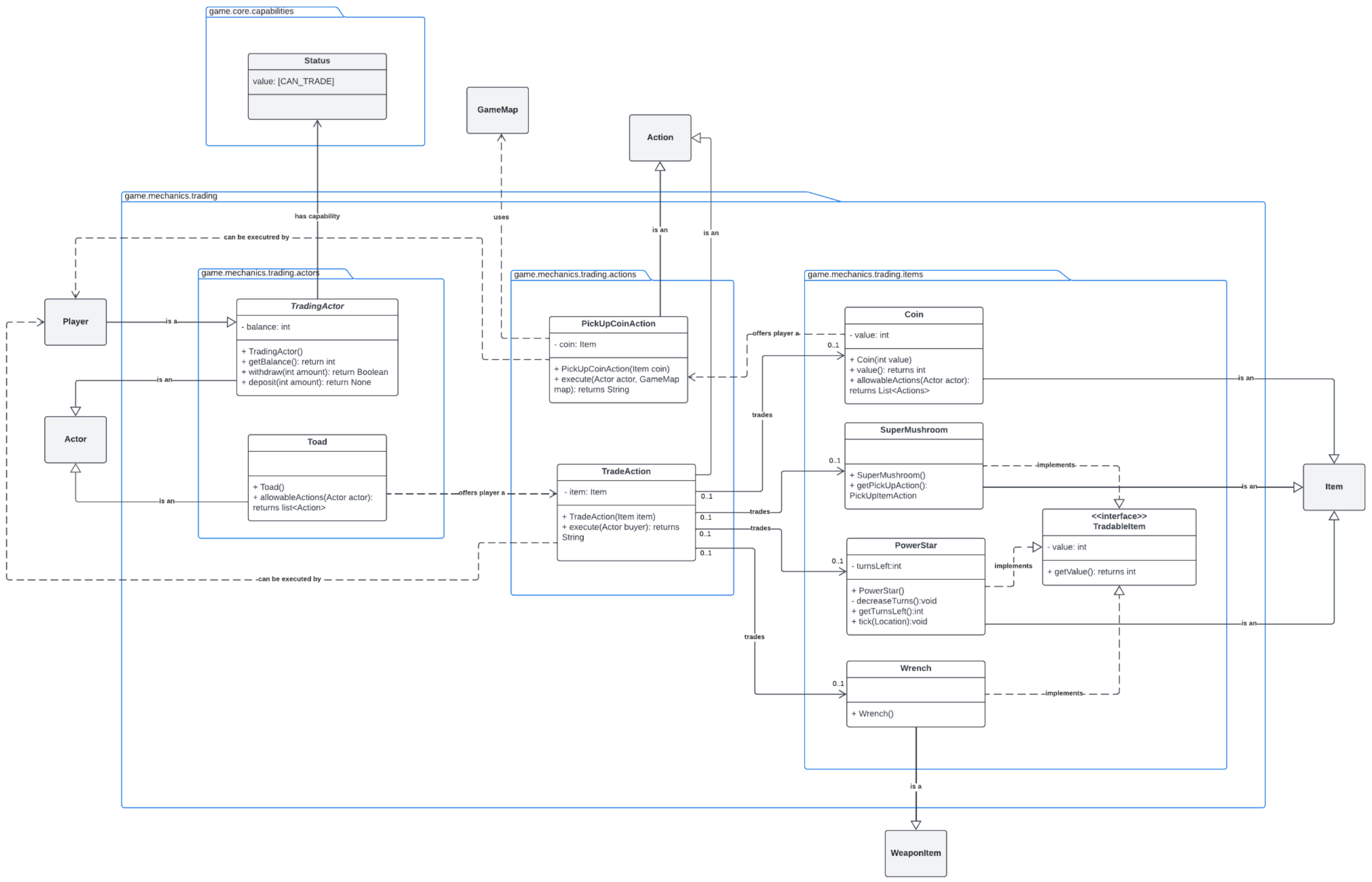

WeaponItem
is a
game.mechanics.combat
allows player to perform
requires player to have
destroys
game.mechanics.combat.actions
0..1
DestroyShellAction
- target: Item
- direction: String
+ DestroyShellAction(Item target, String direction)
+ execute(Actor actor, GameMap map): returns String
SelfDestructAction
+ SelfDestructAction()
+ execute(Actor actor, GameMap map): returns String
AttackAction
extends
uses
GameMap
uses
may perform
game.mechanics.combat.actors
Goomba
- random: Random
+ Goomba()
+ IntrinsicWeapon() : returns Weapon
+ playTurn(ActionList actions, Action lastAction, GameMap map, Display display): returns Action
has
0..1
Koopa
+ Koopa()
+ IntrinsicWeapon() : returns Weapon
has
is an
game.mechanics.combat.items
Wrench
+ Wrench()
1
KoopaShell
1
+ KoopaShell():
+ allowableActions(Actor actor): returns List<Action>
+ getDropAction(Actor actor): return Action
is an
game.core.behaviours
Behaviour
implements
uses
game.mechanics.combat.behaviours
AttackBehaviour
+ getAction(Actor actor, GameMap map): returns Action
is an
implements
implements
Actor
<<interface>> Resettable
Item

1.4 Requirement 4 - Special Items

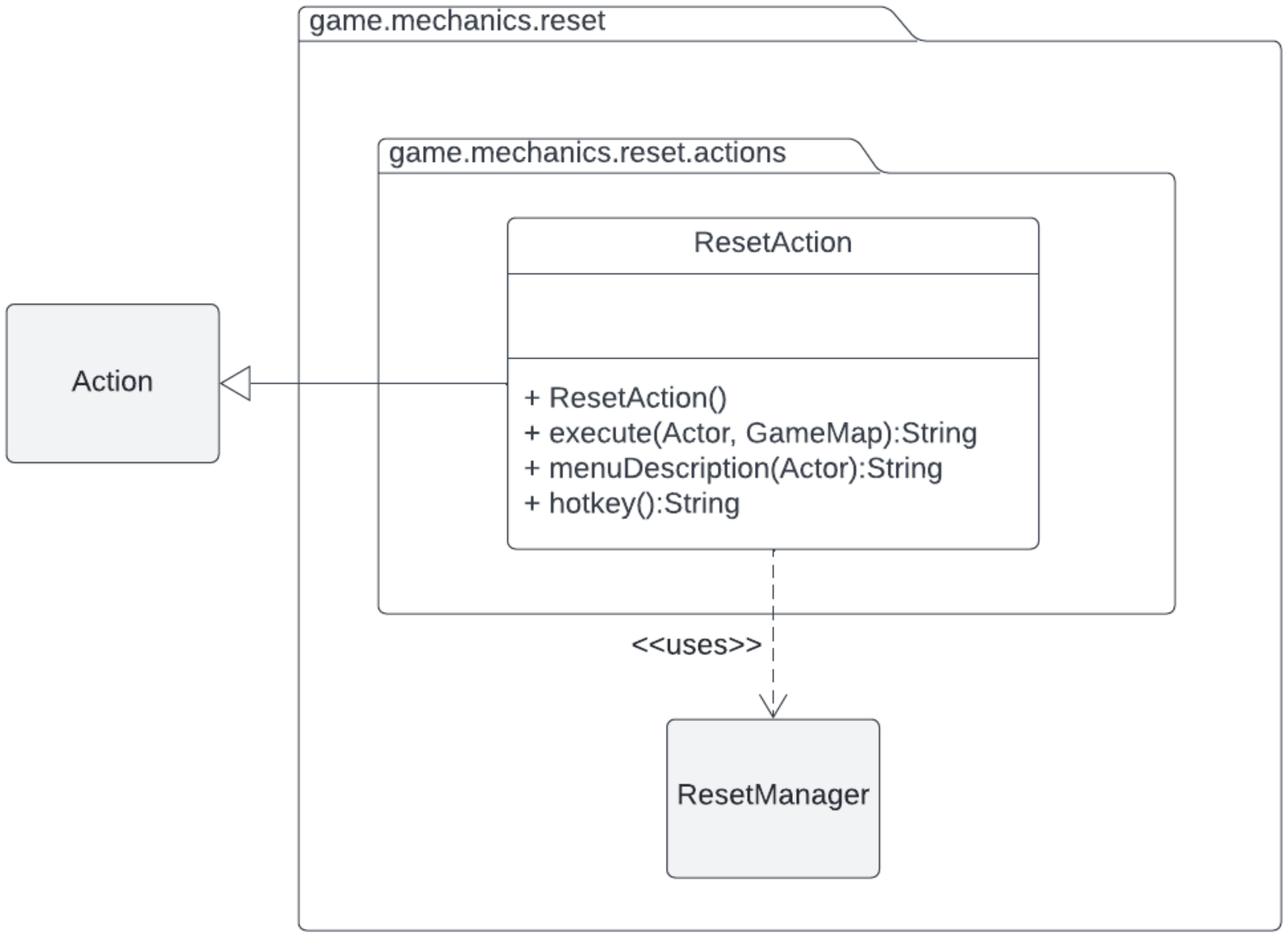


Item
PickUpItemAction
game.mechanics.magical
game.mechanics.magical.items
SuperMushroom
+ SuperMushroom()
+ getPickUpAction(): PickUpItemAction
+ getDropAction(): DropItemAction
PowerStar
- turnsLeft:int
+ PowerStar()
- decreaseTurns():void
+ getTurnsLeft():int
+ tick(Location):void
+ getPickUpAction(): PickUpItemAction
+ getDropAction(): DropItemAction
<<creates>>
<<creates>>
game.mechanics.magical.actions
PickUpSuperMushroomAction
+ PickUpSuperMushroomAction()
+ execute(Actor, GameMap):String
+ menuDescription(Actor):String
PickUpPowerStarAction
+ PickUpPowerStarAction()
+ execute(Actor, GameMap):String
+ menuDescription(Actor):String
<<creates>>
<<creates>>
DropSuperMushroomAction
+ DropSuperMushroomAction()
+ execute(Actor, GameMap):String
DropPowerStarAction
+ DropPowerStarAction()
+ execute(Actor, GameMap):String
DropItemAction

1.5 Requirement 5 - Trading

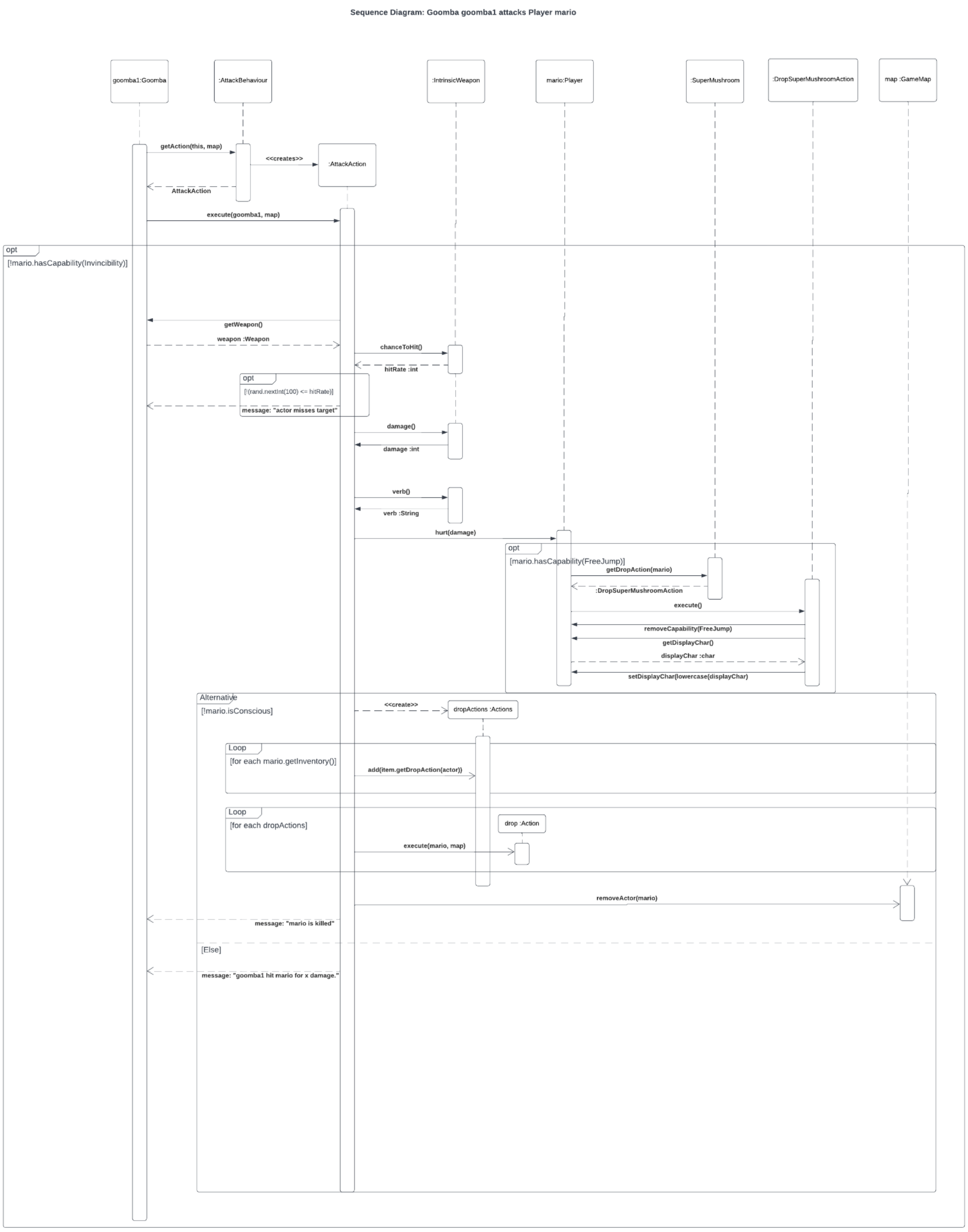


1.6 Requirement 7 - Resetting the Game

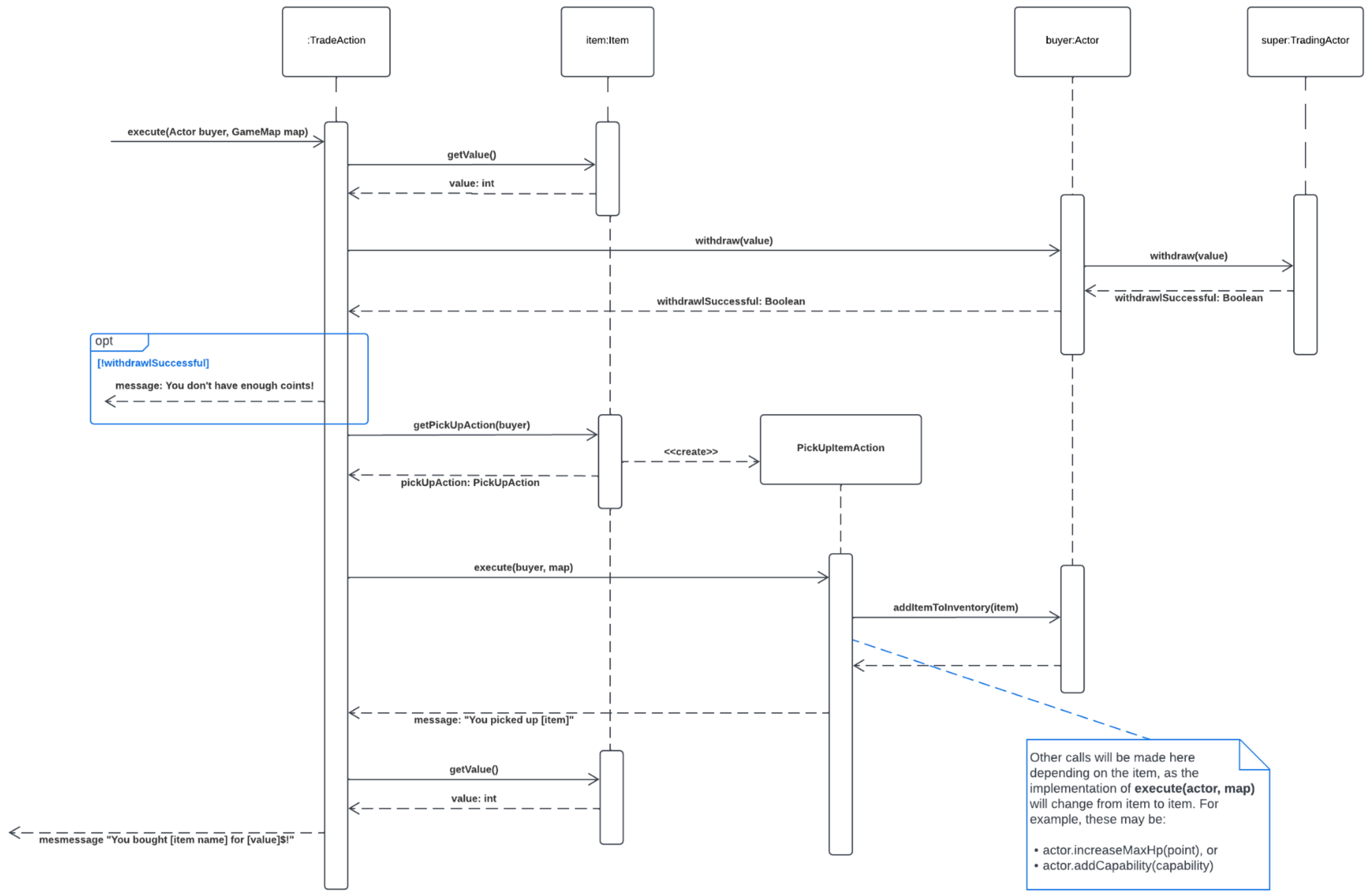

game.mechanics.reset
game.mechanics.reset.actions
ResetAction
+ ResetAction()
+ execute(Actor, GameMap):String
+ menuDescription(Actor):String
+ hotkey():String
Action
<<uses>>
ResetManager

2. Sequence Diagrams

2.1 Goomba attacking a player



2.2 Player buying an item from Toad


:TradeAction
item:Item
buyer:Actor
super:TradingActor
execute(Actor buyer, GameMap map)
getValue()
value: int
withdraw(value)
withdraw(value)
withdrawlSuccessful: Boolean
withdrawlSuccessful: Boolean
opt
[!withdrawlSuccessful]
message: You don't have enough coints!
getPickUpAction(buyer)
<<create>>
PickUpItemAction
pickUpAction: PickUpAction
execute(buyer, map)
addItemToInventory(item)
message: "You picked up [item]"
getValue()
value: int
mesmessage "You bought [item name] for [value]$!"
Other calls will be made here depending on the item, as the implementation of **execute(actor, map)** will change from item to item. For example, these may be:
• actor.increaseMaxHp(point), or
• actor.addCapability(capability)

3. Design Rationale

3.1 Overview

This section outlines how we designed each requirement and why we designed each requirement in the way we did.

As we decided on a design, we made sure to refer to object oriented programming practices and particularly the SOLID principles:

- Each class has one responsibility and one responsibility only.
- Requirements were met by adding new classes rather than modifying existing engine classes.
- Subclasses always function correctly when called as their parent class (polymorphism).
- There is a different class/interface for every use case so that each use case only has the functionality it needs
- High-level and low-level classes depend on abstractions.

3.2 Other Notes

Class diagram legend

- Existing classes are represented in grey
- Completely unmodified classes are represented by a rectangle containing only the name of the class while modified/new classes are represented with standard UML class notation
- If a class is modified, only the changes are shown in the class diagram
- The newly added features are in their packages. Classes outside the packages are part of the existing system

Package structure

When structuring the packages, we essentially had two choices - to package by layer, or to package by feature (or game mechanic). Package by layer means the top level packages would be “layers” (e.g. actors, items, actions, etc.), with each layer being subdivided into mechanics (e.g. actors.combat, actors.trading, etc.). Packaging by feature would be the opposite of that. One is not inherently better than the other, and has trade offs when applied to our specific case which we needed to consider.

When packaging by layer, it is immediately clear which package a given class belongs to, as typically all classes in the game will extend some functionality of the engine. This would also result in the program mimicking the structure of the engine, so if you understand how the engine is structured, you understand both.

In doing so, however, it may become harder to reason about the code due to the resulting low cohesion and high coupling. It will be difficult to see what the program does just by looking at its folder structure (every mechanic will be spread out across all layers), and individual layer packages will be highly coupled to one another.
The other option is to package by “feature” (or in our case, mechanic). Doing so will result in low coupling between packages (for example, the trading package will be largely self-contained and not depend on any other package). It will also result in higher cohesion as it will be easier to see what the program does just by looking at its folder structure.

Going with this approach, however, makes the design more difficult. Which mechanic does the status enum that enables capabilities fit in? It’s not immediately clear.

We decided to go with the second approach (package by feature) as this will hopefully result in code that is easier to implement, extend and maintain. To mitigate the issue of setting boundaries between packages and deciding where feature-ambiguous classes should go, we created a “core” package which houses functionality used by many mechanics. The resulting package structure looks something like this

- game
    - mechanics
        - world
            - actors
            - actions
            - items
              positions
            - …
        - jump
            - actors
            - actions
            - items
            - positions
            - …
        - combat
            - actors
            - actions
            - items
            - positions
            - …
        - magical
            - actors
            - actions
            - items
            - positions
            - …
        - trading

        - actors
        - actions
        - items
        - positions
        - …
    - reset
        - actors
        - actions
        - items
        - positions
        - …
- core
    - actors
    - actions
    - Items
    - positions
    - …

3.3 Class descriptions and rationale

3.3.1 Requirement 1 - Trees

Class Tree *extends Ground implements HigherGround*
This is an existing class in the game. Most of the functionality will be in the tick method, which will call corresponding helper method(s) depending on the growth stage of the tree. At the end of every tick, the grow() metod will be called which increments the age and sets the appropriate growth stage for the tree if required. The growth stage will be stored in an enum.

Attributes:

- age: int *(the age of the tree in turns)*
- stage: TreeGrowthStage *(an enum describing the growth stage of the tree)*

Methods/Constructors:

+ tick(Location location): None *(executes relevant tree behaviour based on growth stage and executes growth method)*
- grow(Location location): None *(increments age, and changes the growth stage of the tree if it passes an age threshold)*
- attemptToSpawn(Location location, Actor actor, double chance): returns Boolean *(spawn the provided actor at the trees location with the provided chance)*
- attemptToDrop(Location location, Item item, double chance): returns Boolean *(spawn the provided item at the trees location with the provided chance)*
- attemptToSpread(Location location): returns Boolean *(spawn another sprout at one of the surrounding exits with the provided chance)*
- attemptToWither(Location location): returns Boolean *(self destructs with the provided chance)*

As the trees behaviour happens every turn, it should be placed in its tick method. The behaviour depends on the trees growth stage, and a simple switch-case statement using an enum is sufficient to implement this. The actual behaviour is extracted into private helper methods, to allow for reuse if other actors or items are required to be spawned in the future.

Enum TreeGrowthStage
A simple enum representing a trees growth stage, with values of SPROUT, SAPLING and MATURE

3.3.2 Requirement 2 - Jumping

Interface HigherGround
This interface contains functionality for jumping to a wall or tree. Both Wall and Tree implement this interface. An interface was chosen to implement this feature because only some of the ground types should implement these features. Using an abstract class to represent high ground type grounds would mean that there are many levels of inheritance, however just using an interface simplifies the design.

Attributes:

- jumpSuccessRate: float *(the success rate of the jump)*
- fallDamage: int *(amount of hit points lost after a failed jump)*

Methods:

+ getJumpSuccessRate(): float *(gets the jump success rate)*
+ getFallDamage(): int *(get fall damage amount)*
+ default allowableActions(): ActionList *(gets allowable actions)*

The method allowableActions is a default method that returns an ActionList with a JumpAction inside. Anything implementing HigherGround (e.g. Wall or Tree) will use this method within their own allowableActions method. We are using a default method so that any class that implements HigherGround would have easy access to any actions associated with being on a higher ground (i.e. JumpAction).

Class JumpAction *extends Action*
This class represents the action of jumping to a HigherGround. This class extends the Action class.
The player is given the option of jumping to a HigherGround when this action is in the action list (and when the player does not have a PowerStar).

Methods:

+ execute(Actor, GameMap): String *(to implement the actor jumping)*
+ menuDescription(Actor): String *(description for menu)*
+ hotkey(): String *(button to press for jumping)*

3.3.3 Requirement 3 - Combat

Class Goomba *extends Actor*
The basis of the Goomba class is already in the game and additional functionality will be added to it. The first thing to be added is an override of the IntrinsicWeapon() with Goomba stats. Secondly, in it's constructor, two behaviours will be added to the behaviours priority hash map - a follow, and an attack behaviour. The self destruction will be implemented in the playTurn method using the random number generator, before the Goomba acts on any of its behaviours.

Attributes:
- random: Random *(a random number generator to implement self destruction with a 10% chance)*

Methods/Constructors:
+ Goomba() *(the follow and attack behaviours will be added inside the constructor. The priority order should be 1. Attack, 2. Follow, 3. Wander)*
+ IntrinsicWeapon(): *(this parent method should be overridden with Goombas stats)*
+ playTurn(ActionList actions, Action lastAction, GameMap map, Display display): Action *(the self destruction will be implemented here, before the behaviours are cycled through. There will be a 10% chance to return the SelfDestructAction)*

Deciding whether to self destruct was placed in the playTurn method rather than the self destruct action as it's the playTurn's responsibility to figure out what action to take, including cycling through behaviours and simulating randomness.

Class Koopa *extends Actor*
The Koopa class will be similar to the Goomba class. It will also override the IntrinsicWeapon method to implement its attack stats. Similarly, it will also implement follow and attack behaviours in an identical way. It will have a KoopaShell item, which will be dropped when its defeated.
Attributes:
- behaviours: Map<Integer, Behaviour> *(this will store all Koopa behaviours in priority order)*

Methods/Constructors:
+ Koopa() *(the follow and attack behaviours will be added inside the constructor. The priority order should be 1. Attack, 2. Follow, 3. Wander. This is also where the KoopaShell item will be added to Koopas inventory)*
+ IntrinsicWeapon(): *(this parent method should be overridden with Koopa stats)*

In essence, once the Koopa is defeated it will be removed, and it will drop its Koopa shell. So it will not exactly be "disabled", but for the player it will appear this way as the shell will take its place. The dropping of the mushroom will be implemented in the shell class.

Class KoopaShell *extends Item*

This will be a non-portable item, added to a Koopa when it's spawned and dropped by the Koopa when it dies. This KoopaShell will allow the player to destroy it (with a DestroyShellAction) if the player has a wrench, which will make the shell drop a super mushroom when it's destroyed.
Attributes:

- None

Methods/Constructors:

+ KoopaShell() *(the Koopa shell will be instantiated as unportable)*
+ allowableActions(Actor actor): *(this method will return a list containing DestroyShellAction if the actor has a wrench)*
+ getDropAction(Actor actor): return Action *(this item class method will be overwritten in the KoopaShell to allow the Koopa to drop it when it dies. This will ensure actors can drop it but can't pick it up)*

Class Wrench *extends WeaponItem*
This is a simple weapon, whose damage and hit chance will be specified in the constructor.
Attributes:

- None

Methods/Constructors:

+ Wrench() *(the Wrench stats will be passed to the super constructor inside this constructor)*

DestroyShellAction *extends AttackAction*
This will be one of the available actions of the KoopaShell class. As the mechanics will be similar to an attack, this class will extend AttackAction and override the target to be an Item instead of an Actor, and override the execute method to remove the KoopaShell and drop a Super Mushroom in its place. The KoopaShell will make this action available only if the player has a wrench in their inventory.
Attributes:

- target: Item

Methods:

+ DestroyShellAction(Item target, String direction) *(instantiates the target and direction, similar to an attack)*
+ execute(Actor actor, GameMap map): return String *(this will destroy the KoopaShell and spawn a super mushroom in its place)*

SelfDestructAction *extends Action*
This is an action that will be returned with a 10% chance from the playTurn method of Goomba as its chosen action. It will essentially simply remove the Goomba from the map
Attributes:

- None

Methods/Constructors:

+ SelfDestructAction()
+ execute(Actor actor, GameMap map): String *(this will simply remove the actor from the game map)*

AttackBehaviour *extends Action*
This is a behaviour used by hostile enemies that will enable them to automatically attack the player
Attributes:
- None

Methods/Constructors:
+ getAction(Actor actor, GameMap map): returns Action *(this will configure and return and attack action on a player, if one is possible. Will return null otherwise)*

3.3.4 Requirement 4 - Magical Items

Class SuperMushroom *extends Item implements TradableItem*
This class represents the Super Mushroom magical item. It extends the Item class.
SuperMushroom will have an allowable action of consuming it.
SuperMushroom will have the capability to jump with 100% success rate.
In the player class, the hurt method will be overridden to check whether the player has a SuperMushroom. If so, the getDropAction will be called to remove the SuperMushroom and its effects from the Player.
The getPickUpAction method will be modified to create and return a PickUpSuperMushroomAction object as a PickUpItemAction type with polymorphism (because PickUpSuperMushroomAction is a subclass of PickUpItemAction).
The getDropAction method will be modified to create and return a DropSuperMushroomAction object as a DropItemAction type with polymorphism (because DropSuperMushroomAction is a subclass of DropItemAction).
These two uses of polymorphism follow the Liskov Substitution Principle.

Methods:
+ getPickUpAction(Actor, actor): PickUpItemAction *(returns an action for picking up the Super Mushroom)*
+ getDropAction(Actor, actor): DropItemAction *(returns an action for dropping the Super Mushroom)*

Class PickUpSuperMushroomAction *extends PickUpItemAction*
This class represents the action for picking up a SuperMushroom instance. It extends the PickUpItemAction class.
The execute method will contain the functionality for increasing the player's max HP, changing the display character to uppercase and gives the capability of jumping with 100% success rate.

Methods:
+ PickUpSuperMushroomAction() *(constructor)*
+ execute(Actor actor, GameMap map): String *(is called when the actor picks up the SuperMushroom)*
+ menuDescription(Actor actor): String *(gives a description of the action for the menu)*

Class DropSuperMushroomAction *extends DropItemAction*
This class represents the action for dropping a SuperMushroom instance (which occurs when a player with a SuperMushroom is damaged). It extends the DropItemAction class.
The execute method will contain the functionality for changing the display character back to lowercase and removing the capability of jumping with 100% success rate.

Methods:
+ DropSuperMushroomAction() *(constructor)*
+ execute(Actor actor, GameMap map): String *(is called when the actor drops the SuperMushroom)*

Class PowerStar *extends Item implements TradableItem*
This class represents the Power Star magical item. It extends the Item class.
PowerStar will have an allowable action of consuming it. Once consumed, it is added to the actor's inventory and removed once the number of turns left is 0. The number of turns is adjusted at every tick.
PowerStar will give the actor the capability to simply walk to higher grounds and destroy higher grounds, dropping a coin in the process. It will give the capability to not take any damage. It will also give the capability to instantly kill enemies with a successful attack.

The getPickUpAction method will be modified to create and return a PickUpPowerStarAction object as a PickUpItemAction type with polymorphism (because PickUpPowerStarAction is a subclass of PickUpItemAction).
The getDropAction method will be modified to create and return a DropPowerStarAction object as a DropItemAction type with polymorphism (because DropPowerStarAction is a subclass of DropItemAction).
These two uses of polymorphism follow the Liskov Substitution Principle.

Attributes:

- - turnsLeft: int (the number of turns until it disappears)

Methods:

- - decreaseTurns(): void *(reduces number of turns left by 1)*
- + getTurnsLeft(): int *(get the number of turns until it disappears)*
- + tick(Location): void *(called every turn - override this method to call decreaseTurns())*
- + getPickUpAction(Actor, actor): PickUpItemAction *(returns an action for picking up the Power Star)*
- + getDropAction(Actor, actor): DropItemAction *(returns an action for dropping the Power Star)*

Class PickUpPowerStarAction *extends PickUpItemAction*
This class represents the action for picking up a PowerStar instance. It extends the PickUpItemAction class.
The execute method will contain the functionality for giving the player the capabilities to walk to HigherGround grounds and destroy them, be immune to enemy attacks, and instantly kill enemies with a successful attack.

Methods:

- + PickUpPowerStarAction() *(constructor)*
- + execute(Actor actor, GameMap map): String *(is called when the actor picks up the PowerStar)*
- + menuDescription(Actor actor): String *(gives a description of the action for the menu)*

Class DropPowerStarAction *extends DropItemAction*

This class represents the action for dropping a PowerStar instance (which occurs when the PowerStar’s number of turns left is zero). It extends the DropItemAction class.
The execute method will contain the functionality for removing the capabilities the PowerStar gives.

Methods:

+ DropSuperMushroomAction() *(constructor)*
+ execute(Actor actor, GameMap map): String *(is called when the actor drops the PowerStar)*

3.3.5 Requirement 5 - Trading

Abstract Class TradingActor *extends Actor*
This is an extension of the actor class which enables trading capabilities. Any actor who is required to be able to trade (i.e. Player), should extend this class instead of actor. It has a private integer balance, which stores the total amount of money an actor has. Money can be deposited with the deposit(int amount) method (for example, in the pick up coin action), and withdrawn with the withdraw(int amount) method (for example, in the trade action).

Essentially, coins aren't stored in a player's inventory - instead their value gets added to an actor's balance and then they are destroyed.

Attributes:

- balance: int *(stores the total amount of money an actor has)*

Methods/Constructors:

+ TradingActor() *(the TradingActor constructor adds a CAN_TRADE capability to the actor, allowing other actors (e.g. Toad) to check if they should offer an actor a trade action)*
+ getBalance(): return int *(returns the actors total balance)*
+ withdraw(int amount): return Boolean *(attempts to withdraw the specified amount. Returns true if successful, or false if insufficient funds)*
+ depost(int amount): returns none *(adds the specified amount to the actors balance)*

Class Toad *extends Actor*
This class represents the games vendor, Toad. In the allowable actions, it checks if the provided actor has a CAN_TRADE capability, in which case it offers a list of TradeActions, one for each item.

Attributes:

- none

Methods/Constructors:

+ Toad() *(this constructor simply sets Toads display character 'o')*
+ allowableActions(Actor actor): returns list<Action> *(if actor has CAN_TRADE capability (i.e. is a TradingActor), returns a list of TradeActions, one for each item for sale)*

Class Coin *extends Item*
This class represents a coin. It allows actors with CAN_TRADE capabilities a pick up coin action. The coins value is set at instantiation

Attributes:

- value: int *(the value of the coin)*

Methods/Constructors:

+ Coin(int value) *(instantiates the coin with provided value)*
+ value(): returns int *(a simple getter for the coins value)*
+ allowableActions(Actor actor): returns List<Action> *(if actor has CAN_TRADE capability, a PickUpCoin action is offered)*

Class PickupCoinAction *extends Action (NOTE: change to extends PickUpAction)*
This action is offered by a coin if the provided actor has a CAN_TRADE capability. In the execution of the action, the coin's value is added to the actor's balance and the coin is then destroyed.

Attributes:
- coin: Item *(the coin which offers the action)*

Methods/Constructors:
+ PickUpCoinAction(item coin) *(instantiates the action)*
+ execute(Actor actor, GameMap map): returns String *(deposits coin into the actors wallet and destroyed the coin)*

Class TradeAction *extends Action*
This action is offered by Toad if the provided actor has a CAN_TRADE capability. It stores the item offered for trade. In the execution of the action, the value of the item is attempted to be withdrawn from the buyer's balance. If this is successful, the item is added to the actor's inventory. If the actor has insufficient funds, the item is not traded and a relevant message is printed.

Attributes:
- item: Item *(the item being traded)*

Methods/Constructors:
+ TradeAction(Item item) *(instantiates the action with the item for sale)*
+ execute(Actor buyer, GameMap map): returns String *(attempts to withdraw items value from buyers balance. If successful, the item is traded. If insufficient funds, the item is not traded. A relevant message is returned at the end)*

Class Wrench *extends WeaponItem implements TradableItem*
This class represents a wrench. It implements the TradableItem interface, which requires it to set its value i.e. price.

Attributes:
- none

Methods/Constructors:
+ Wrench() *(the constructor in which the wrench attack stats are set)*

Interface TradableItem
A simple interface which requires the items which implement it to have an attribute of value, representing its price in dollars to enable trading.

Attributes:

- value: int

Methods/Constructors:

- none

3.3.6 Requirement 7 - Resetting the game

ResetAction
This class represents the action of resetting the game. It extends the Action class, because resetting the game is an action (and it needs to be listed in the menu, be executed via a hotkey, etc.)
Once the execute method has been called, the action is then removed from the ActionList. In the execute method, the ResetManager is fetched, and everything in the list of resettables will be reset.

Methods:

+ execute(Actor, GameMap): String (to implement the resetting of the game)
+ menuDescription(Actor): String (description for menu)
+ hotkey(): String (button to press for resetting the game)