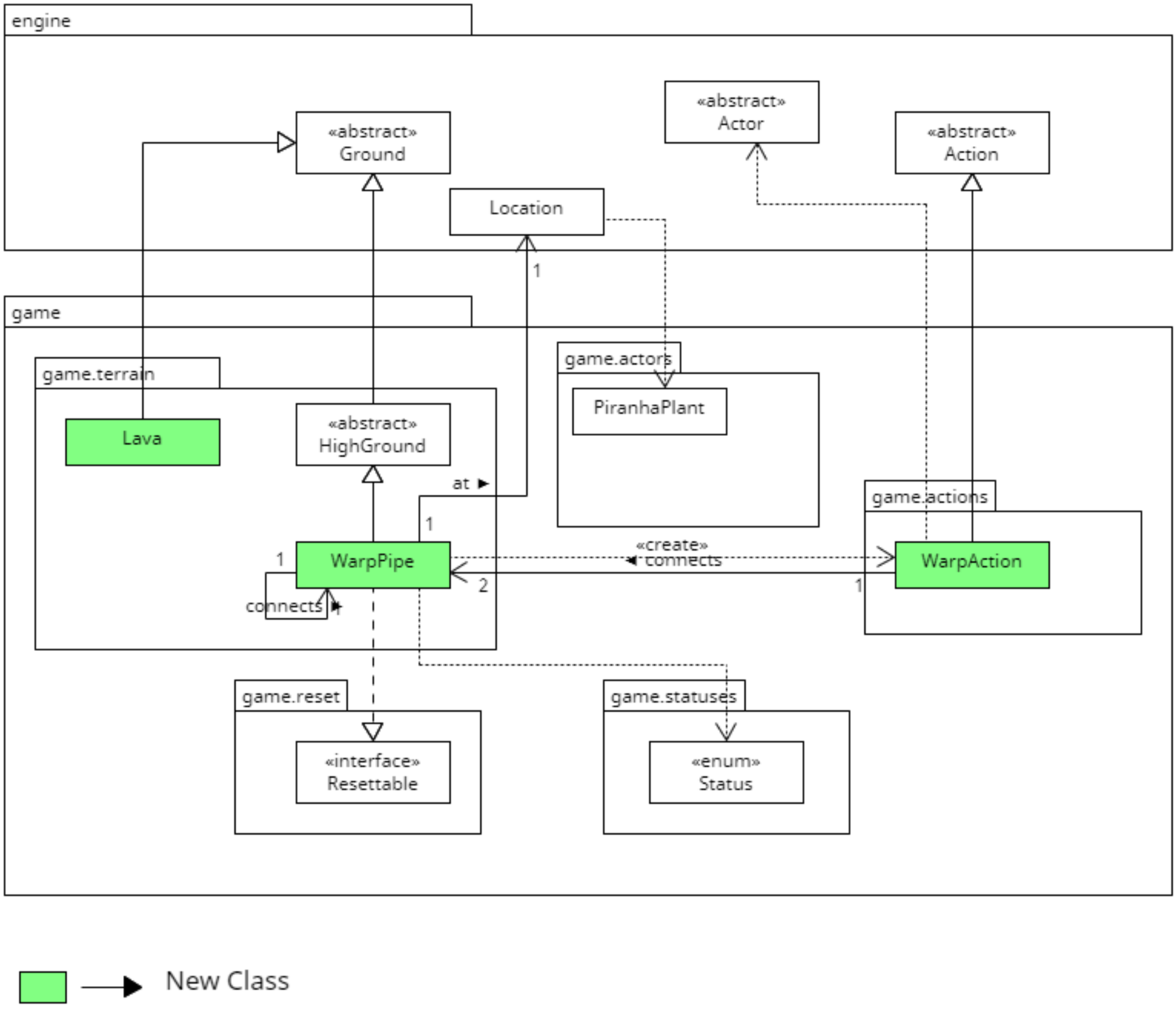

engine
«abstract»
Ground
«abstract»
Actor
«abstract»
Action
Location
1
game
game.terrain
Lava
«abstract»
HighGround
game.actors
PiranhaPlant
at
1
1
WarpPipe
connects
2
«create»
connects
1
game.actions
WarpAction
game.reset
«interface»
Resettable
game.statuses
«enum»
Status
New Class

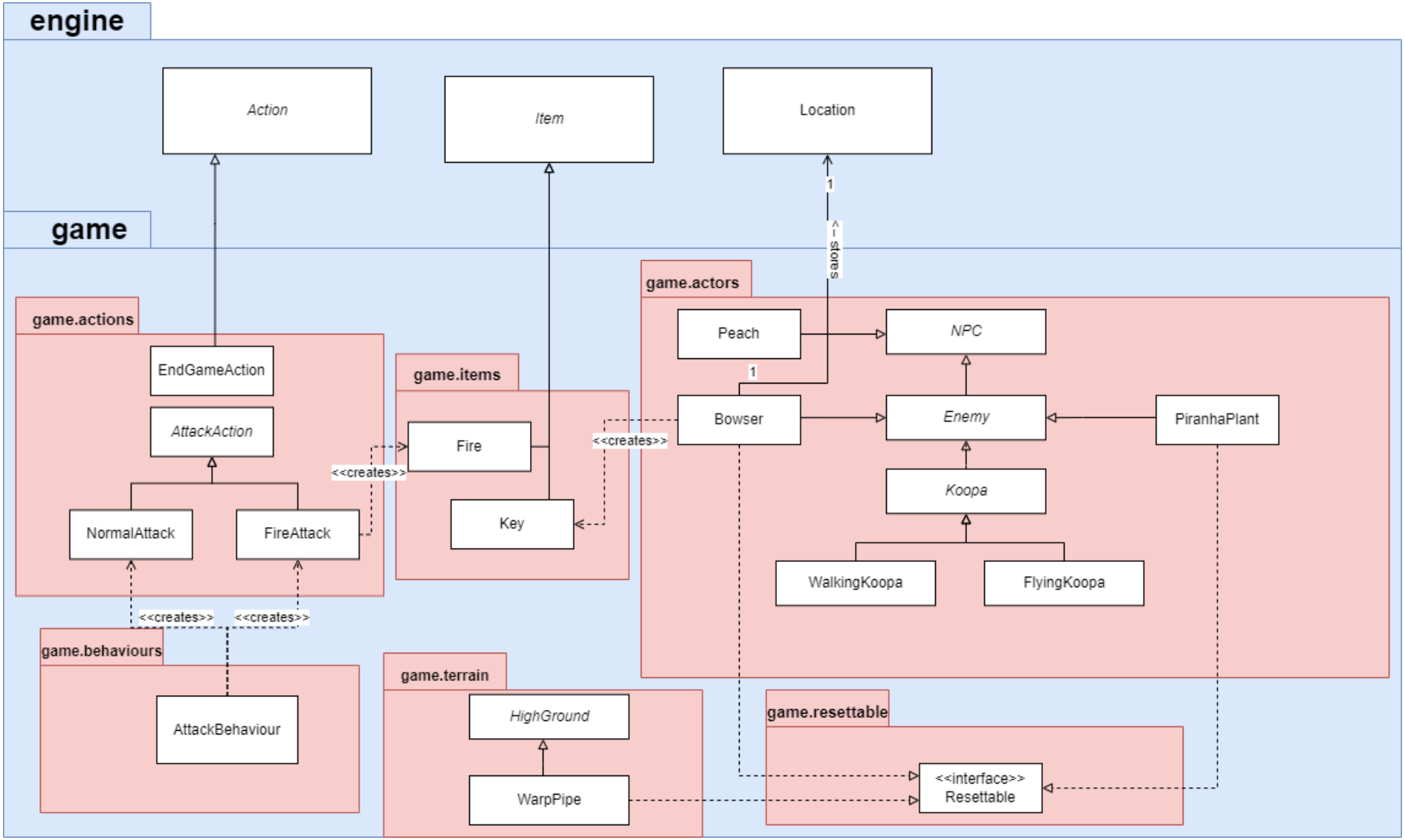
WarpPipe is a class that extends from HighGround. As HighGround already implements an Actor jumping onto it, we do not need to implement it in WarpPipe apart from setting the successRate and damage values following DRY. This also follows LSP.

WarpPipe has an association to itself. This is as a WarpPipe needs to know which other WarpPipe to warp to making it a 1 to 1 association.
WarpPipe also has a 1 to 1 association to Location. A Ground class does not have an association to Location but this is needed in WarpPipe in order to know which Location the WarpPipe is at. This is important as the Player warps to a Location, not a Ground.

When an actor is standing on the WarpPipe a WarpAction is created.
WarpAction extends from Action and handles warping an actor from a WarpPipe to another.
This means WarpAction needs an association with WarpPipe, storing the source pipe and the connecting pipe (1 to 2)
The actual warping can be done with the moveActor method in GameMap, effectively using the engine code.
The Location to warp to is an attribute in the connecting pipe.

The Lava Map is created and added to World in Application.
If more maps were added it might be useful to implement the maps elsewhere rather than Application to adhere to SRP but as this is the only map added it seemed unnecessary.
The Lava Map contains a new ground type Lava which extends from Ground.
Ground already has a dependency on Location in the Tick method so no new dependencies are needed.
Every tick, the Lava will check Location if there is an Actor and damage them.

Requirement 2 UML Diagram + Documentation by Andy



Added a Peach class that extends NPC because future behaviours could be implemented for her.

Added EndGameAction such that if the player has a key and interacts with Peach, a victory message will appear, and the player will be removed from the map (ending the game)

Added Bowser class that extends Enemy and implements Resettable. Intrinsic damage is overridden to allow for the correct attack damage

Bowser stores the location that he spawns in and will be moved back to this location when reset is called, resetting health, and removing the FollowBehaviour from the behaviours hash map if the key (follow behaviour priority) exists.

Made AttackAction abstract so that different types of effects can be implemented/applied to attacks such as generating fire, allowing for flexibility, extensibility and following DRY. Extracted a block of code in AttackAction that calculates damage into its own method (getDamage). Addresses the Long Method code smell

Added FireAttack and NormalAttack.

Bowser will have a capability FIERY such that in AttackBehaviour a FireAttack will be returned.

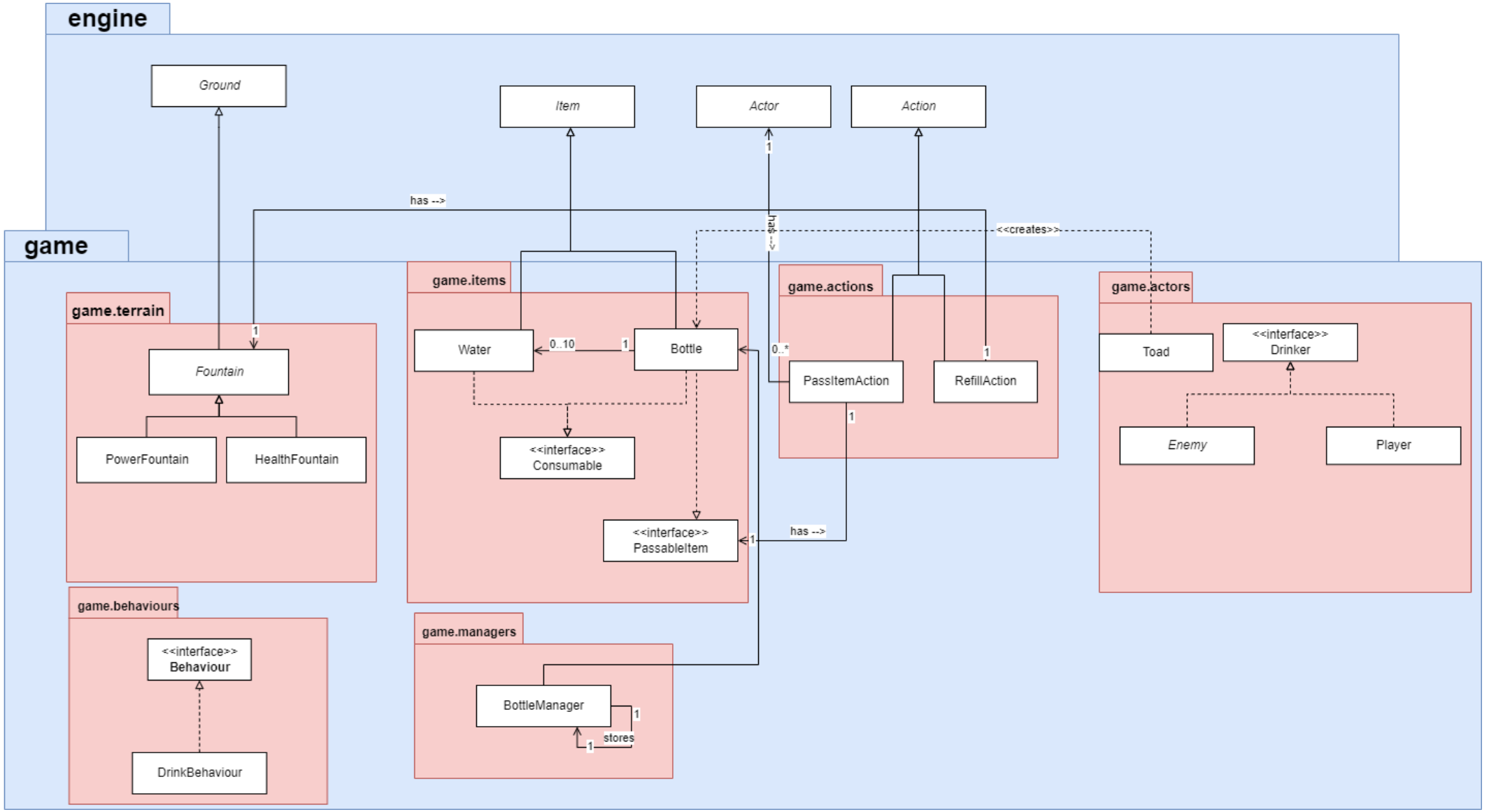
Added Fire item class that accepts a 'turns' parameter indicating how many turns left until the fire disappears from the ground. Tick method will check if there is an actor standing on the fire and will apply damage if it is the case.

Added DeathMechanic class, extracted and moved the death logic of AttackAction (dropping items, removing actor) into a static method named 'death' in the DeathMechanic class. In the PlayTurn method of player and NPC, if the actor is not conscious, DeathMechanic.death(actor) is called. This allows the mechanics of death to stay fully functional even if an actor dies from sources other than an attack. This also addresses the Long Method code smell.

Added PiranhaPlant class that extends Enemy and implements Resettable, intrinsic damage is overridden to allow for the correct attack damage. The amount of health increased by reset is kept as a constant inside a Utils class full of constants and will be referred to when resetting

Made Koopa class abstract, added new WalkingKoopa and FlyingKoopa class that extends Koopa. This implementation follows DRY and allows for extensibility. The FlyingKoopa will have a capability CAN_FLY such that the canActorEnter method for high ground returns true if an actor has this capability. LSP is not violated here because both forms of Koopa have all the functionalities of their parent if not more, so the two can be substituted in anywhere the code expects their parent class

Requirement 3 (including optional challenge) UML Diagram + Documentation by Andy



Created PassableItem interface containing passItem(receiving actor, passing actor) method. This allows for extensibility as any item that needs to be passed in the future can directly implement this interface.

Added PassItemAction extending Action that stores the passingActor, and calls passItem with this passingActor in the execute block.

Added Bottle class that extends Item and implements Consumable and PassableItem.

Added BottleManager singleton class. This class keeps a hash map storing the actor string as a key, and a Bottle as value. This allows for extensibility as any future actor can have a bottle, rather than just one instance of bottle existing. Bottle's unique passItem method body will register the Bottle into the hash map with the receiving actor's string as the key and add the same bottle instance into the actor's inventory.

BottleManager also allows for simple refill logic since we don't need to iterate through the player's inventory nor typecast to refill the bottle. It is also easier to check if an actor has a bottle or if an actor's bottle is at max capacity.

Added abstract class Fountain that extends Ground.

Created Water class that implements Consumable and stores the Fountain that created it, so that the effects can be accessed.

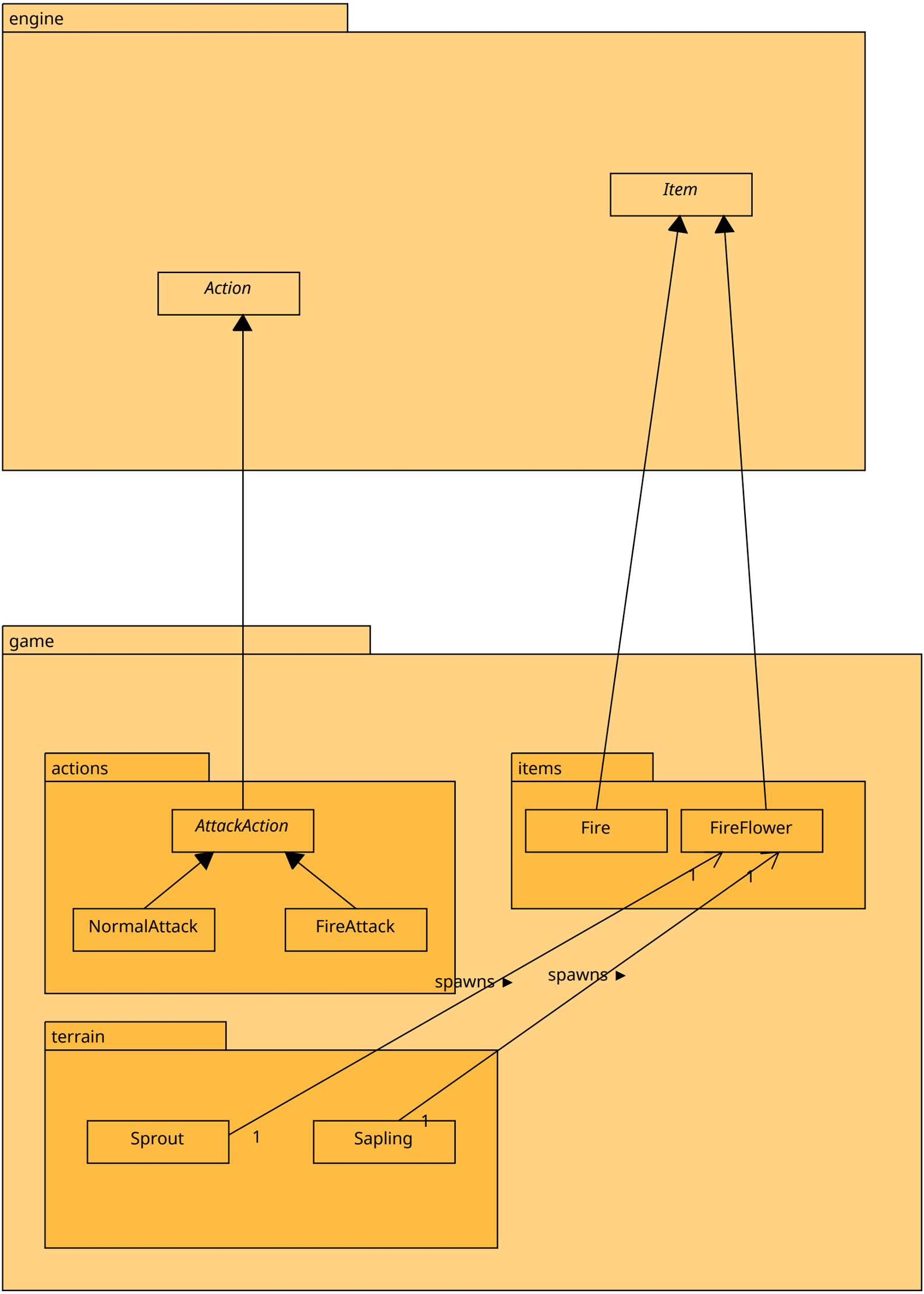
Added RefillAction extending Action that stores the fountain the player will refill from. Added refill method to BottleManager that accepts an actor string and an instance of fountain as parameters so that it can create and add a new instance of Water to the player's bottle in the hash map.

Added Drinker interface with the methods to increase base attack and hit points. Both the player and the enemies will implement this interface.

Added abstract method giveEffects(Drinker) to Fountain so that any type of fountains with different effects of consumption can easily be integrated into the system. Added PowerFountain and HealthFountain. Fountain will generate RefillAction if the player is standing on it.

Water's consumedBy(Actor) method will call fountain.giveEffects(Drinker) which requires type-casting Actor to Drinker because there is no other way to pass an actor into a method that requires a Drinker as a parameter under the constraint that ALL enemies must be able to drink from the **fountain** (accessing Drinker methods). If it was ONLY the player that was allowed to drink (from a single existing instance of Bottle), then we can simply allow the bottle to accept an instance of Drinker in its constructor and pass the Player into it during instantiation. An abstract class implementation of Drinker would still require downcasting from Actor to Drinker.

Added DrinkBehaviour that scans the current ground the Enemy stands on and checks if it is a Fountain using capabilities, if so, the fountain will generate a ConsumeAction for water using allowableActions.


engine
Item
Action
game
actions
items
AttackAction
Fire
FireFlower
NormalAttack
FireAttack
1
1
spawns ▶
spawns ▶
terrain
Sprout
1
Sapling
1

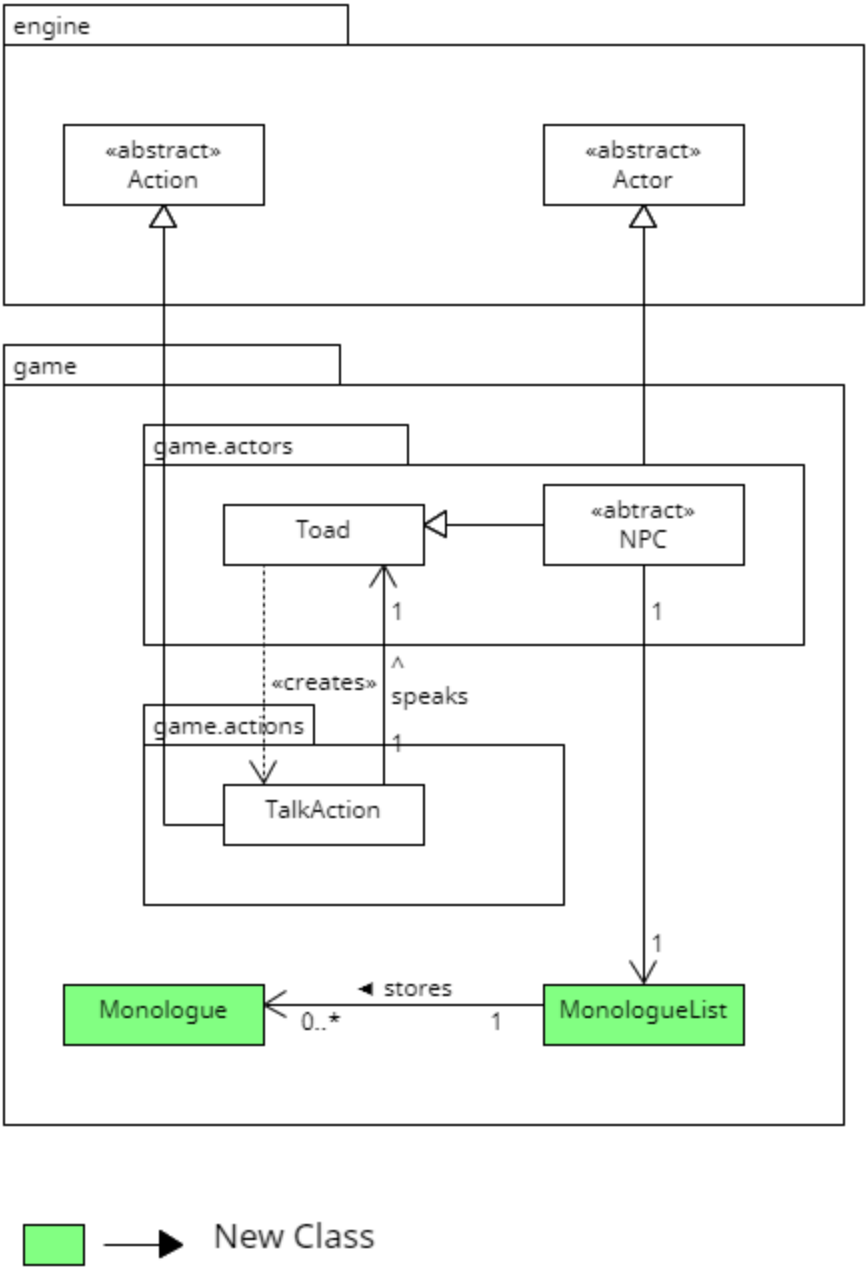
Design rationale:

Fire flower is implemented as item and not grounds as it is spawned and not grown. This allows to implement Consumable on this fire flower. Further, allowing us to follow the open close principle.

Fire is also implemented as an Item so that it can be removed from the map after a set amount of turns and can be dropped on the map as “v” when using fire attack. Both implementations the dependency inversion principle as they depend on abstractions.

AttackAction has been made abstract to allow Liskov Substituion principle. This abstraction allows us to create Normal Attack and Fire Attack that extend Attack Actions, also it means its open for extension in future too, if we do need to add some other sort of attacks.

Also when fire flower is consumed, as it is not specified in specifications. The 20 turns counter just prints the next 20 turns counter if 2 fire flowers are consumed and so on.


engine
«abstract»
Action
«abstract»
Actor
game
game.actors
Toad
«abtract»
NPC
1
1
«creates»
^
speaks
game.actions
1
TalkAction
1
Monologue
stores
0..*
1
MonologueList
New Class

Monologue is a new class to represent a sentence an Actor can say. This allows us to easily extend the functionality being
much better than simply using a String. Monologue stores a String representing the sentence to be said and a conditi on
in the case that it is talking to another Actor. This condition is a Capability the Actor must have in order to speak that Monologue.

MonologueList is a class to store Monologues. This is useful as it allows us to add functionality such as randomly selecting a Monologue, adhering to OCP and SRP

As all NPCs can talk, a 1 to 1 association with MonologueList is added to store all the Monologues the NPC can spe ak.
The playTurn method in NPC checks if there is the TALK status, picking a random Monologue to speak, if not then it will
add the TALK status. This implements the talking on alternating turns for every NPC.

With the new Monologue and MonologueList the TalkAction was modified to use these classes.