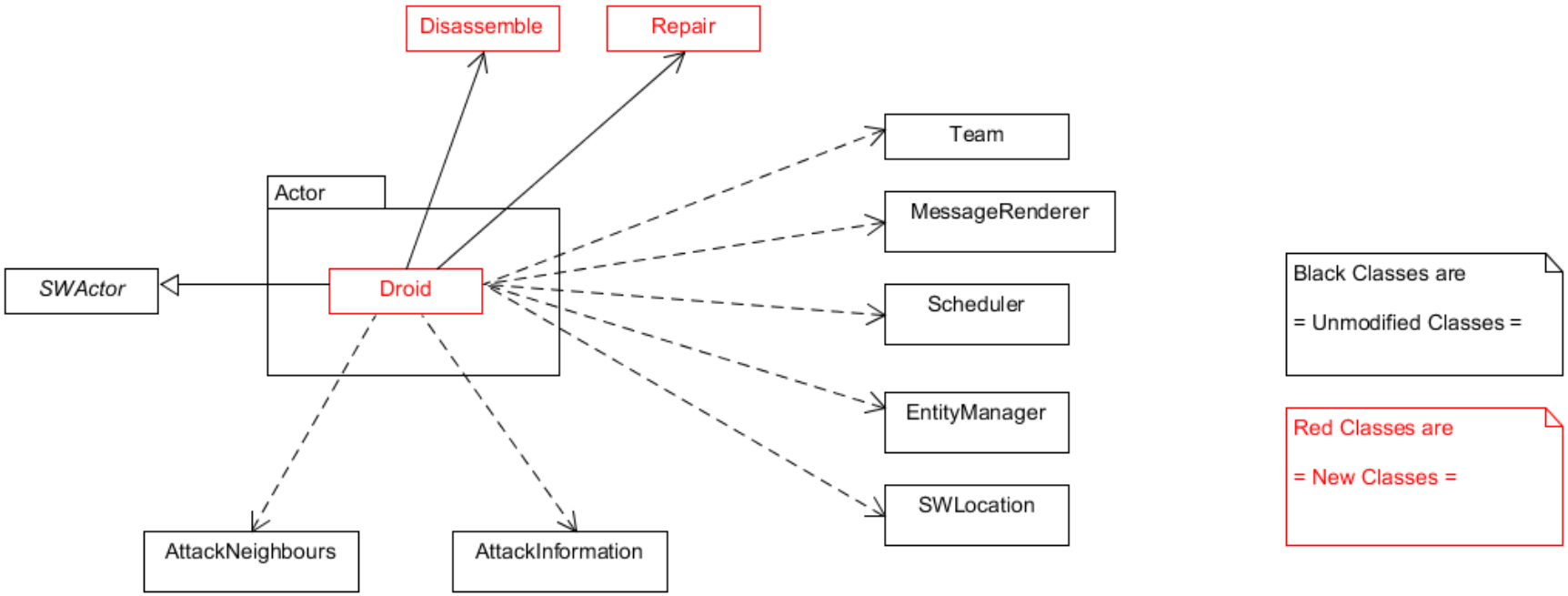
1) UML-Droid_Actor



UML Class Diagram For New Droid Class extending Actor

New Classes Roles & Responsibilities:

A) Droid

The Droid Class will inherit from SWActor and will be the class used to create generic Droids. It works very similarly to other classes that inherit from SWActor, like the TuskenRaider or BenKenobi . It does however, has its own unique methods/attributes that makes it a Droid class.

The key differences that separate the Droid class from other actors are:

1) Droid classes have the option of being created with an owner or without. An owner will be a SWActor object stored as an attribute in the Droid class. A method to change owners or remove owners will also be created in the Droid class.
2) Droid classes have a Follow behavior. Droids will follow their owners.
3) Droid classes have a TerrainDamage behavior. Droids will take damage when moving in Badlands.
4) Droid classes are created with the Oil affordance, which lets them be oiled.
5) Droid classes gain the Disassemble and Repair affordances when they are immobilized (health drops to zero).

B) Disassemble

An affordance that will be attached to droids when they are immobilized to allow them to be disassembled into DroidParts .

C) Repair

An affordance that will be attached to droids when they are immobilized to allow them to be repaired with DroidParts

Interactions to provide functionality:

In UML-Droid_Actor, we simply showed that Droid will function similar to existing SWActor subclasses (TuskenRaider and BenKenobi). This means that Droid will interact with existing classes shown in the UML in a similar manner.

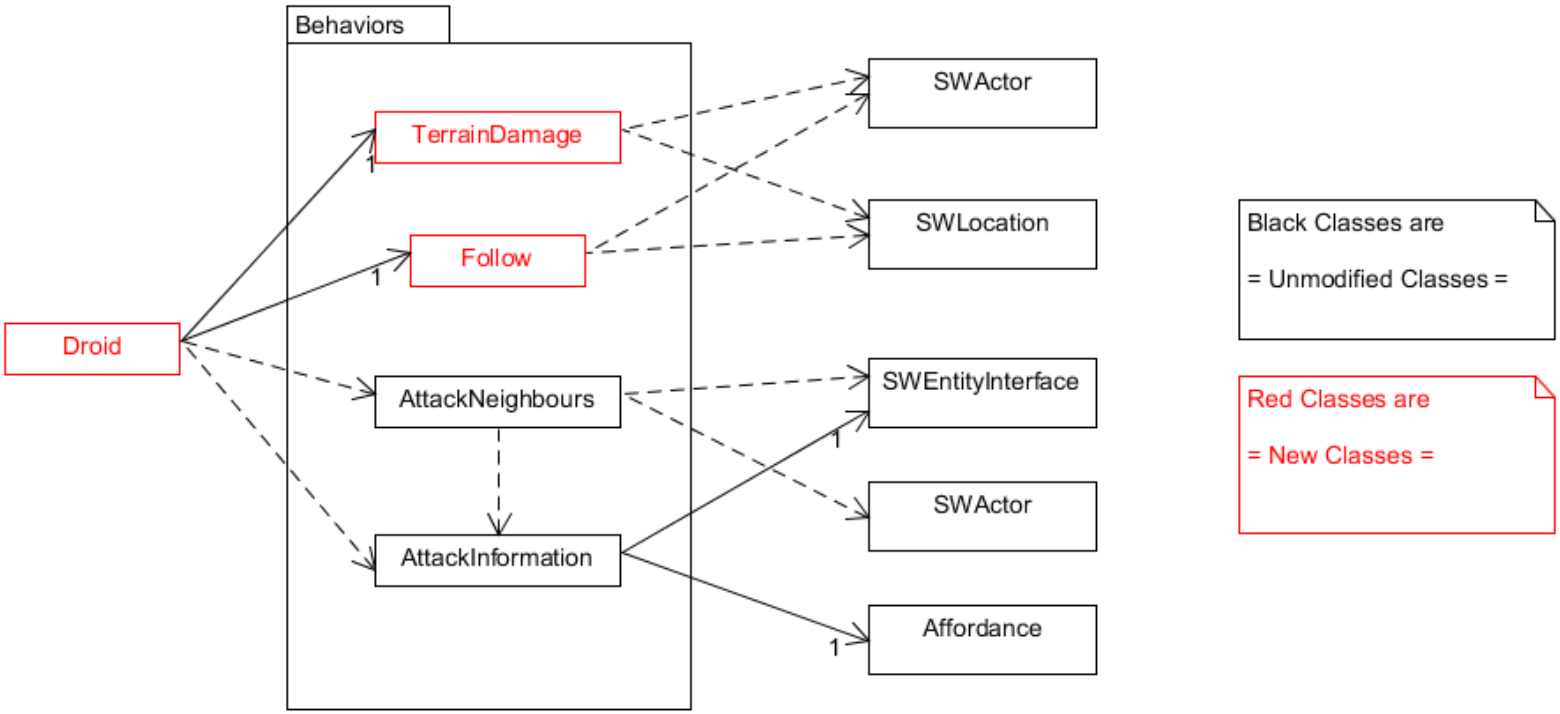
The extra functionalities shown here are that Droids are created with owners (Droids store SWActor classes as attributes), and droids gain Disassemble and Repair affordance when they are immobilized.

Design Rationale:

Droid extends from SWActor because it takes actions. Inheriting from SWActor allows Droids to make use of existing classes and interact with existing classes in a way that requires no modifications to them. This avoids extra work from unnecessary modification. This also allows us to make use of information hiding and encapsulation done by existing interactions.

Adding special attributes and methods in only the Droid class is also to supports unnecessary modification of existing classes to accommodate adding droids into the game.

2) UML-Droid_Behavior



UML Class Diagram For New Droid Class Behaviors

New Classes Roles & Responsibilities:

A) Droid

Explained in : 1) UML-Droid_Actor, behaviors shown here are how the Droid class will act.

B) TerrainDamage

An affordance that will be attached to droids when they are immobilized to allow them to be disassembled into DroidParts .

C) Follow

An affordance that will be attached to droids when they are immobilized to allow them to be repaired with DroidParts

Interactions to provide functionality:

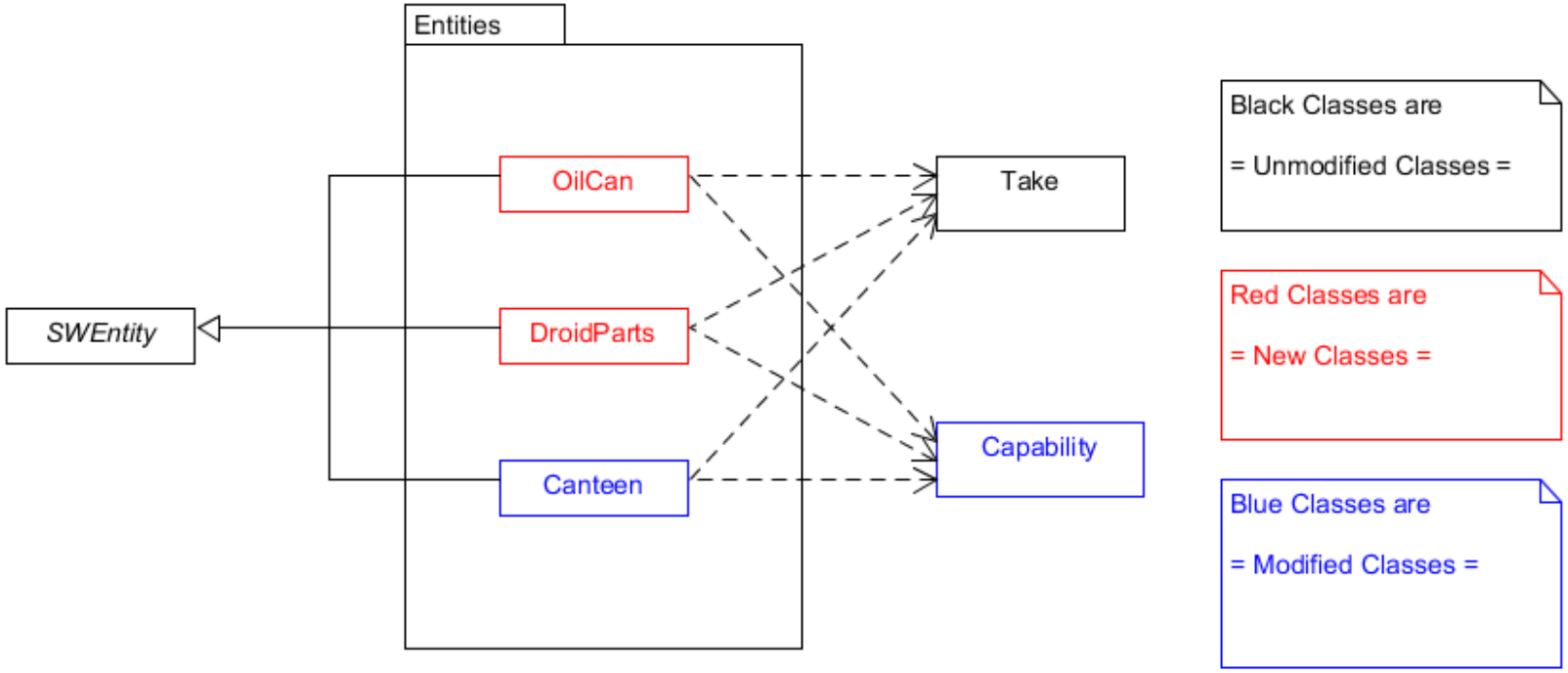
TerrainDamage is a new behavior for Droids to make them take damage when attempting to move in the Badlands. To determine the position of the Droid when moving and also the location of the Droid, TerrainDamage has to use methods in SWLocation. To actually damage the Droid (reduce its hitpoints), TerrainDamage has to use a method in SWActor.

Follow is another new behavior for Droids to make them 'follow' their owner. Droids 'follow' their owner by moving towards their owner until they are standing on the same spot on map as their owner. To do this, Follow in Droid has to make use of the SWActor class that they are storing and SWLocation to get the location of their owner on the map.

Design Rationale:

Adding new behaviors to the Droid class this way allows implementation of functionality without modification to existing classes. This helps us prevent unnecessary modification work and allows use existing information hiding and encapsulation methods as mentioned before in 1) UML –Droid_Actor. It also allows for easy addition of any new behavior if ever needed.

3) UML-DroidHealing_Entity



UML Class Diagram For New OilCan/DroidParts Entities & Canteen Modification

New Classes Roles & Responsibilities:

A) OilCan

A new SWEntity subclass that represents an oil can. It can be picked up (taken) and gives the owner the capability to oil a droid.

B) DroidParts

A new SWEntity subclass that represents droid parts. It can be picked up (taken) and gives the owner the capability to repair a droid.

Modified Classes:

A) Canteen

To accommodate drinking from the canteen for healing, canteen will be modified to give the owner the capability to drink from the canteen when it is filled. This capability will be removed once it is drank from.

B) Capability

Capability has to be modified to accommodate all the new capabilities. This includes “oil”, “repair” and “drink”.

Interactions to provide functionality:

OilCan and DroidParts work similarly to other SWEntities (objects that can be picked up and “used”). When picked up, a SWActor gains the capability to oil droids and repair droids. They add the functionality of being able to oil and repair Droids.

Note that only certain SWActors can repair droids. Thus, DroidParts alone does NOT give an SWActor the ability to repair droids. Certain SWActors will be created with an added capability “REPAIRER”. This capability combined with the capability given by droidparts will allow the SWActor to repair droids.

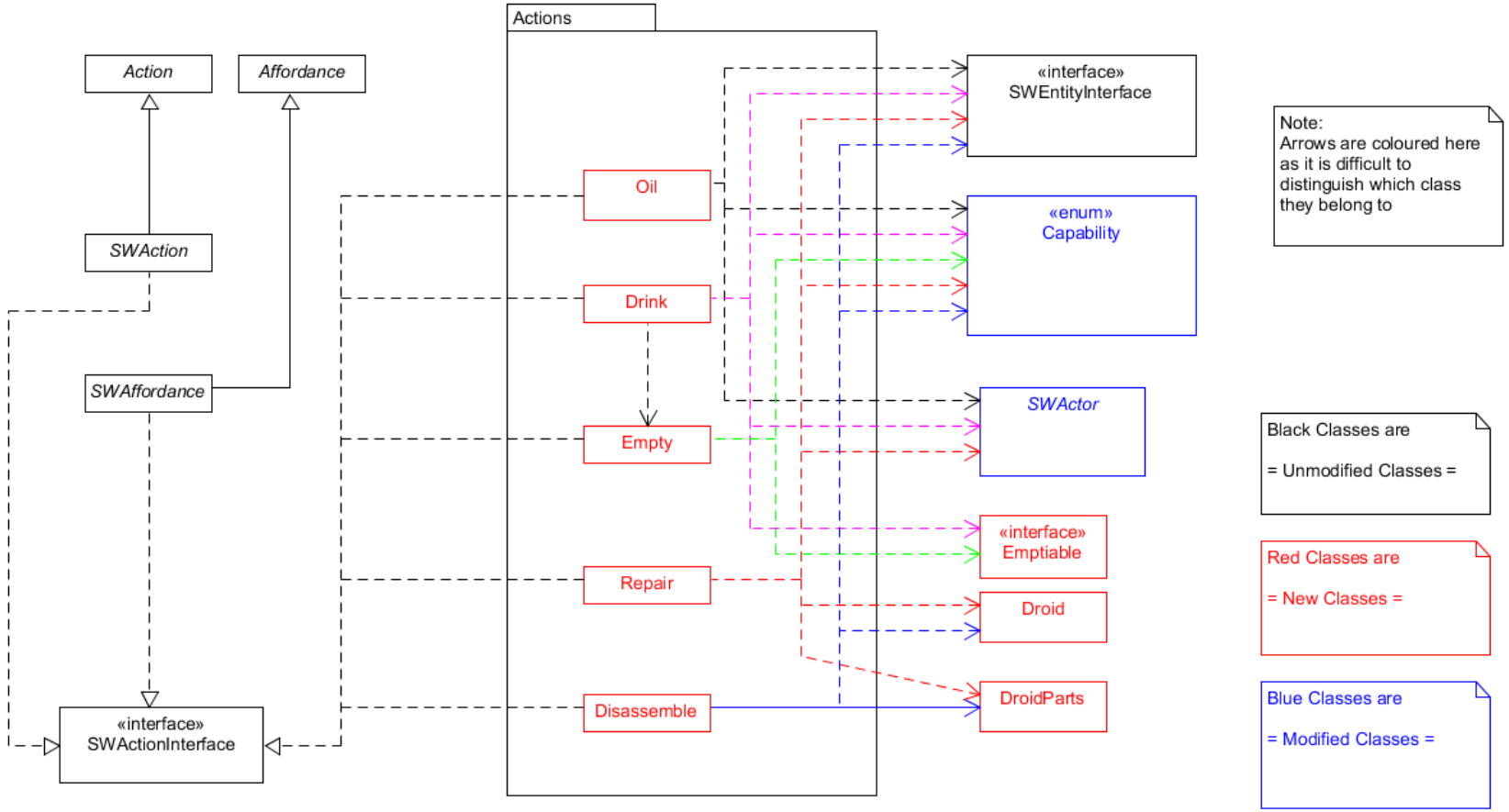
Canteen has to be modified to accommodate the action of drinking from it. A filled canteen gives the “DRINK” capability to the canteen holder. After drinking, the canteen will also be emptied. Further details on this is explained in 4) UML DroidHealing_Actions.

Design Rationale:

Adding OilCan and DroidParts as SWEntities and having them interact with Take and Capabilities in the same manner as other SWEntities once again allows us to avoid unnecessary modifications to existing classes, and maintain information hiding and encapsulation methods.

Slight modifications have to be made to the Capability and Canteen classes to accommodate the new functionality, but this is minor, requiring little work and no modifications to other classes.

4) UML-DroidHealing_Actions



UML Class Diagram For New Actions Involving Droids & Healing

New Classes Roles & Responsibilities:

A) Droid

Explained in 1) UML-Droid_Actor. Here, Droid adds/removes the Repair affordance once damaged/repaired. Disassembling will cause the Droid class to delete itself.

B) DroidParts

Explained in 3) UML-DroidHealing_Entity. Here, the repair action uses up (deletes) DroidParts and the Disassemble action creates a DroidParts object.

C) Oil

An affordance added to droids that allow them to be oiled

D) Drink

An affordance added to the canteen when it is filled that allows it to be drank from.

E) Empty

Resets capacities of SWEntities with capacities (only canteen as of now).

F) Emptiable

Interface for SWEntities that can be emptied.

G) Repair

An affordance that will be attached to droids when they are immobilized to allow them to be repaired with DroidParts

H) Disassemble

An affordance that will be attached to droids when they are immobilized to allow them to be disassembled into DroidParts .

Modified Classes:

A) Capability

Capability has to be modified to accommodate all the new actions possible from the new affordances (Oil, Drink, Empty, Repair, Disassemble).

B) SWActor

SWActor has to be modified to enable healing of the SWActor object. It already has a takeDamage() method, we add a healDamage() method that does the opposite for healing.

Interactions to provide functionality:

Oil interacts with SWEntity to obtain "oilable" targets on the map, Capabilities to determine if a SWActor can oil a target and with SWActor to heal the SWActor that is oiled.

Drink interacts with SWEntity to obtain the "drinker", Capabilities to determine if a SWActor can drink and with SWActor to heal the SWActor that is drinking. It interacts with the Emptiable interface to empty the Canteen after drinking.

Empty empties the canteen that is drank from after Drink interacts with the Emptiable interface.

Repair interacts with SWEntity to obtain repair targets. Capabilities to determine if a SWActor can repair and with SWActor to heal the SWActor(Droid) that is repaired. It uses methods in Droid to remove the repair affordance on Droids after repair and DroidParts to remove the DroidParts SWEntity after it is used for repair.

Disassemble interacts with SWEntity to obtain disassemble-able targets and Capabilities to determine if a SWActor can disassemble a droid. It interacts with the Emptiable interface to empty the Canteen after drinking. It uses methods in Droid to remove the Droid after disassembling it (deleting the Droid object) and DroidParts to add a DroidParts SWEntity after a droid is disassembled.

Design Rationale

Oil, Drink, Repair, and Disassemble are subclasses of Affordance similar to other actions to allow for use of existing interactions with existing classes. As mentioned In the other 3 UMLs, this avoids unnecessary modification work, allows use of existing encapsulation methods, and is a very maintainable way of adding new actions.

Modifying SWActor to add a healDamage() method helps reduce repeating work when healing an SWActor is required. It is also easy to implement and understand since it is similar to the existing takeDamage() method.

Repair and Disassemble are special compared to other actions. Repair consumes a DroidParts object, while Disassemble creates a DroidParts object. Thus, they have to be designed in such a way that they interact with the DroidParts object.