# Updated Design Rationale

## Reservoir

* The *Reservoir* must implement a method to check conditions (its current health points).
* There is already an inbuilt function **takeDamage()** inherited from superclass. We simply need to make another method which, after taking damage, checks the current health points of the *Reservoir* entity, and updates its descriptions and symbol, if needed.
* We call that method **updateStatus()**, and call it after the *Reservoir*’s health points have been deducted.

## Grenade

* The *Grenade* has the ability to be detonated. Therefore, all instances of *Grenade* are initialized with a *Detonate Affordance*.
* To achieve the required functionality of detonation, the following two methods are added to *SWLocation* and *SWWorld*:
  + In *SWLocation*, the method **getNeighboursLoc(int steps)** will get all the *SWLocation*s that can be reached within the specified steps, and separate them into an array list, based on the steps taken from the current *SWLocation* instance to reach a target *SWLocation*.
  + In *SWWorld*, the method **getNeighboursContents()** will return all the *SWEntities* that are within a specified amount of steps from a *SWLocation*.
* After being detonated, the only reference to the grenade instance is the itemcarried in SWActor, thus we set it to null to clear it

## Sandcrawler

* The *Sandcrawler* moves in the same way that *BenKenobi* does. Therefore, *Sandcrawler* must have a *Patrol* variable, and the constructor for *Sandcrawler* must assign that variable to the same *Patrol* path that *BenKenobi* has. But it will only move in second turn, which is controlled by moved attribute. The *Sandcrawler* must also have an inner world, which is a class of its own. We have decided to name this inner world as *SWMobileWorld*.
* The *Sandcrawler* moves with a turn delay. One way to achieve this is by setting a flag, perhaps called *moved*, which is set to false. In the **act()** method, it checks that flag. If it is false, set it to true and simply exit the method. If it is true, perform several actions, and set that flag to false again. One downside of this method is that it is not a very elegant or efficient method. However, it is sufficient to produce the correct logic behind the turn delay, and is rather short, ensuring that it can be refactored in the future easily.
* *Droids (Actually anything with collectable capability)* are collected by the *Sandcrawler*. Whenever the *Sandcrawler* happens upon collectable entity*,* it will pick it up and throw it into its inner world. This functionality can be implemented in the *Sandcrawler*’s **act() method.** A droid is initialized in the internal world of the sandcrawler.
* A *SWActor* may also enter a *Sandcrawler*, if it is in the same *Location*. This is dependent on the *SWActor*’s force value. We have created *Enter Affordances and Exit Action*, which are assigned depending on several conditions. The reason of using *SWAction* as *Exit* is because the actor inside *sandcrawler* should be able to exit at any time. For *Enter*, the *SWActor* must have non-zero force value (which means they can use the force). The *SWActor* must be placed in the same current *Location* of the *Sandcrawler* when exiting, (0, 0) in our case. Thus, we have implemented several methods which allows us to move not just *SWActor*s in and out of the inner and outer worlds. These methods are called **enterInnerWorld(SWActor a)**and**exitInnerWorld(SWActor a)**. When actor exit the internal world of sandcrawler, it may not appear in the same location with sandcrawler since the event of move may be executed after the exit of actor, but the actor indeed exited to the correct location at the point the command execution.