Design Rationale

Assignment 1

- Leave Affordance

  Affordances are also defined as Actions. Entities have a Take affordance, which allows them to pick an item up. This affordance is realized by *SWActionInterface*. Star Wars Affordances also extends *SWAffordance*, and *SWAffordance* is realized by *SWActionInterface* as well. Thus, it makes sense for Leave to extend *SWAffordance* (and by definition, is realized by *SWActionInterface)*. Leave should have similar methods to Take, with the condition for **canDo()** requiring that the actor have an item in their hand. The **act()** method should remove the item from the actor's hand, and place the item in the Star Wars Entity Manager.

- Force ability

  Force ability should be implemented as an integer value that is initialized in all starwar actors. Several actors who are known to be able to use the *Force* should extend a *Jedi* class. The checkForce function will be called along with gerAction(this function return a list of action that can be performed by the particular actor). The function will return a boolean value determine if a character can wield lightsabre

- Lightsabres

  Lightsabres are an item that has already been defined in the code base. To satisfy the condition that "only people who are attuned to the *Force* can wield one as a weapon", all that is required is to modify the constructor of *Lightsabre*. There will be a simple Boolean check to determine if the *Actor* that is currently in possession of the *Lightsabre* has sufficient *Force* ability to use it as a weapon. The cut-off point for one to wield a *Lightsabre* as a weapon could be, on a scale of 1-100, 75 for example.

- Ben Kenobi

In order to implement the training functionality whereby *Ben Kenobi* can train Luke (the player) if they are in the same location, the *Player* actor should be a subclass of *Jedi*, first and foremost. *Player* will have a *Force* value, and subsequently should also have a method to check the maximum *Force* value to possess (so as to limit the effectiveness of training). *Ben Kenobi* will then have a method **trainLuke()**, which is called whenever the player is at the same location on the grid. The method can be set to increase *Force* ability by a set amount, or a variable amount (depending on several external factors).

- Droids

Droids should resemble a non-playable *Actor*. *Droid* could extend *SWEntity*, which allows it to reuse the methods already defined in it. An *Actor* can have multiple *Droids*, but each *Droid* should only have one owner. Thus, each *Droid* could possess an owner variable, which indicates that the *Droid* in question is owned by an *Actor*. The *Droid* would be placed in the same team as its owner. A *Droid* cannot die, so when its health points run out, (if it were to be attacked, or is moving in the Badlands) it loses its capability to move. It is also possible that a *Droid* is autonomous; in which the owner variable will be set to null. Some methods that a *Droid* may have include:

1. **ifOwned()**, which check if the owner attribute of the droid has been set to anyone
2. **ifOwnerInNeighbour** (), it will call the getNeighbour() in Location to determine the neighbour of the droid and call whereIs() in EntityManger to get the location of its owner and compare them in order to return a Boolean value represent if owner in the neighbour of the droid
3. **ifDirectionAvailable(compassBearing c)**, it will call the function hasExit() in Location to determine if the direction in valid
4. **takeDamageInBadlands()**, which decreases the *Droid's* health points when walking through the Badlands
5. **checkIfMobile(),** it will check if the hitpoint of the droid is positive to determine if the Move() function is called in the following code

We also create a new action called "Own" extend affordance in starwars.actions package to help us setup the relationship between owner and droid