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 *SWActors*. This implementation allows for easy arithmetic operations to increase the force ability of an actor, especially when a method to train someone in the *Force is needed.* An actor who is unable to use the *Force* will have a force value of 0. As shown in the class diagram, the **checkForce (int Force)** function can be called along with **getActions()** (this function returns a list of action that can be performed by a particular *Actor* based on their *Force* potency). The function will return a Boolean value to determine if a character can wield a *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, *Ben Kenobi* will need to have a new method implemented. *Player* will have a *Force* value, and subsequently should also have a method to check the maximum *Force* value to possess (to ensure there is a set ceiling for 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**(), which will call the **getNeighbour()** in Location to check the neighbouring locations of the droid. It will also call **whereIs()** in EntityManager to get the location of its owner, and compare them to return a Boolean value that represents if its owner is in any neighbouring location.
  3. **ifDirectionAvailable(compassBearing c)**, which will call the function **hasExit()** in Location to determine if there is a valid direction in which the droid can move. The droid will then act based on the available directions.
  4. **takeDamageInBadlands()**, which decreases the *Droid's* health points when walking through the Badlands.
  5. **checkIfMobile(int health),** which will check if the health points of the droid is positive, to determine if the Move() function is called in the following code.

  There will also be a new action called *Own*, which extends *Affordance* in starwars.actions package, to help define the relationship between a *Droid* and its *Owner*.