Ben Kenobi:

* For Ben Kenobi requirement, I figured I’d make a new affordance called “Train”.
* Extends SWAffordance implements SWActionInterface
* This affordance will be attached to SWActors that can train or be trained, in this case attached to only Ben Kenobi and Player (Luke).
* The behavior of Train will be very similar to Attack, with the difference being Train affects Force while Attack affects Hitpoints.
* Allows an entity with Train affordance and the force ability to train another entity which has the Train affordance and no force ability
* Training can only happen if both actors are in the same team
* An entity can train only if it has a higher Force than the other
* After training, the actor that is trained will have enough Force to wield the lightsaber
* This action will take up 1 tick (similar to all other actions)
* For AI (non-human controlled), there will be another new class “TrainNeighbours” which will return an AttackInformation object specifying which entity to be trained. TrainNeighbours is very similar to AttackNeighbours, and returns AttackInformation because AttackInformation class consists of entity and affordance so I just reuse it.
* Advantages:
* Opens development of Force training for the future
* Similar behavior to Attack so not much code is going to be added
* Disadvantages:
* Will have to add new functions to Actors that has Train affordance
* Scheduling might get a little bit messy, training and attack priority will need to be considered

Droid:

* Make a new SWActor “Droid”
* Extends SWActor
* Droids are neutral team and will have their Attack affordance removed from it when initialized
* New attributes:
* private SWActor owner: stores the owner of droid / who droid needs to follow
* private Boolean autoPilot: specifies if it is following owner / moving in random direction
* private Follow path: the path that droid needs to go to follow owner
* New methods:
* void toggleAutoPilot(): on and off for autoPilot
* boolean getAutoPilot(): returns autoPilot value
* boolean isImmobile(): returns True if owner is null or HP is less than 0, and False otherwise
* string evaluateLocation(): evaluates the location the droid is in and drops hp accordingly
* These new attributes and methods should fulfill all the requirements for the assignment. However, there is a new class “Follow” that I have to elaborate to explain the movement of the droid.
* The new class “Follow” is a behavior like Patrol from BenKenobi.
* It has 2 attributes in it: droid and owner. Initialized when the constructor is called
* Methods for Follow:
* boolean ownerIsNearby(): searches for location of droid and owner then evaluates if the location of owner is one move away from droid according to the CompassBearings.
* Direction ownerDirection(): finds out which direction the owner is from droid. This method can only be run when ownerIsNearby() is true.
* Direction randomDirection(): returns a random CompassBearing.
* The flow of Follow event is explained in the sequence diagram.
* Advantages:
* Can have a lot of droids in the game and each droid will have an actor.
* Follow behavior can be used by other characters in future development by changing droid into SWActor.
* Disadvantages:
* Droid and Follow has a lot of dependencies, especially Follow also has dependency on SWWorld and EntityManager.

Force:

* A new class called Force
* Force object will be created throughout all SWActor class
* There will be a forceInIt Boolean variable to indicate whether the following SWActor has force or not
* The class will also contain method hasForce to check whether certain actor has force or not
* Advantage:

1. Easier to implement new capabilities to the force class.
2. Makes the structure tidier.

* Disadvantages:

1. More complicated than adding a force Boolean variable to the SWActor

LightSaber:

* New method called checkForce to check whether SWActors that wields it has force.
* If the SWActor does not have force, it will return “null” instead off “Capability.Weapon” when “Attack” class check its Capability so it is not treated as weapon by “Attack” Class.
* Advantage:

1. Not overly complicated
2. Short and compact code

* Disadvantages:

1. Will need to access the SWActor that hold this item.(might ve complicated)

Leave Affordance:

* Reversing all that TakeAffordance do
* Have all method and variable that ”takeAffordance” has, but have reversed effect
* canDo method check whether there is item at hand
* “act” method remove ItemCarried and add item to EntityManager
* getDescription method return string “leave” with item description
* Advantage:

1. Simple to implement

* Disadvantage:

1. Similar to TakeAffordance