**Rationale**

* **Feature 1 Leave Affordance**

**Design choice:**

To realize the leave affordance feature, a new action class Leave is introduced to the system. Leave will inherit all the methods of SWAffordance which includes a constructor,a canDo(), an act() and a getDescription() method. The constructor creates an instance of Leave Affordance, the canDo() method checks if the actor is carrying an item because only actors currently carrying an item can do the action Leave, the act() method first check if the the item currently carried is of type SWEntityInterface, then set the item to the location where the actor is currently at and set the itemCarried to null because after the Leave action, the actor should not be carrying anything. An item that is left by some actor shouldn’t be able to left again because it is currently on the map instead of holding by an actor. Therefore the Leave Affordance needs be removed from the set of Affordance that the item has and the Take Affordance needs to be added because the item left should be able to be picked again.

Changes are made in existing Take Affordance as well, in the act() method, since after the Take action, the item should be able to be left therefore a new Leave Affordance instance is added to the item.

**Reason:**

The reason why Leave is treated as affordance and added to the action package is because when Leave occurs, the item is being left instead of the item leaving something, this indicates that this is in fact an affordance. Hence the Leave class needs to inherit from its parent SWAffordance. Since in real life only item being carried could be left, Leave will need to check if an item could be left by checking if the actor is currently holding anything. Only when the item is being carried can it be left. In reality, an item can’t be picked and left simultaneously therefore the Take and Leave affordance could not exist at the same time, hence the Take affordance instance is removed and the Leave affordance instance is created in Take class and vise versa.

**Advantages:**

1. Easy to implement
2. Without the need to change any other class than Take.
3. Following DRY principle

**Disadvantages**:

1. Increased dependency on engine code

* **Feature 2 Force ability**

**Design choice:**

To realize the Force ability feature, a new enum class is introduced to the system. Inside the enum class, there are three possible choices to choose which includes Force.None indicating that no force is possesse, Force.Weak indicating that some force is possessed but not enough to be used and Force. Force.Strong indicating that lots of force is possessed and could be used. Since all actor has one of the three force state therefore the new force attribute with default Force.None is added to the abstract class SWActor. An accessor getForceState() and a settor setForceState() is added to the SWActor as well. According to the spec, Luke(player ) should be at Force.Weak and Ben should be at Force.Strong and TuskenRaider should be at Force.Strong as well.

Therefore the attribute value is changed accordingly in their separate classes.

**Reason:**

The reason why Force is being implemented as a enum class instead of a attribute like hit points is because if it is like hit points then it will require a numerical threshold value to distinguish the different states which at the moment in not sure of. That will cause design problems in the future, therefore force ability is being implemented as enum class. This way, no disputes will be brought and the requirement is satisfied.

**Advantages:**

1. Easy to implement
2. Without the need to change any other class than SWActor
3. Following DRY principle

**Disadvantages**:

1. Loss of accuracy for force because the implementation only allows different levels instead of exact values

* **Feature 3 Lightsaber**

**Design choice:**

To realize the Lightsaber feature, The Attack class is modified, the idea is to check if the actor is carrying a lightsaber by using a Boolean variable isLightSaber, to check if the actor has the Force.Strong state by using another Boolean variable canWieldLightSaber because only actors with Force.Strong are able to wield to wield a Lightsaber. Then the attack chould be categorized into following:

1. If both the Boolean variables are true, this indicates the actor is carrying a lightsaber and is able to wield so attack will be using the itemcarried
2. If isLightSaber is false, this indicates the actor is not carring lighsaber this will just remain unchanged
3. If isLightSaber is true and canWieldLightSaber is force, the attack will be using barehands

**Reason:**

The reason why class Attack is changed is because of the Don’t repeat yourself principle. Since the force only decides if the actor’s attack action, therefore only class Attack is required to be modified.

Inside attack, it is only required to decide which attack situation the actor is in therefore the two Boolean variable to help dicide the situations and then proceed with the attack.

**Advantages:**

1. Easy to implement
2. Without the need to change any other class than Attack
3. Following DRY principle

**Disadvantages**:

1. Increased dependency on SWActor class due to the added Boolean attributes
2. Complexity of Attack class increases

* **Feature 4 Ben Kenobi**

**Design choice:**

Create a new class train in the package starwars.entities.actors.behaviors.

As Luke can only be trained by Ben when they are in the same location, thus we must check this condition before we set Luke force ability to strong. And this can be shown in detailed as following:

1. Firstly, we should get the location of Ben by calling the function whereIs(Ben).
2. Then, we should get all the entities in that location (Ben included) by passing the location of Ben to the function contents(loc).
3. Since the contents(loc) will return a list containning all the entites in that location, so we need to go through each entity in that list and check that if there is a Player(Luke) instance, also the entity we check should exclude Ben himself. If a Player object has been founded, then we know Ben and Luke are in the same location, and we can set Luke’s force ability to Strong so that he can wield a lightsaber. (This step should only happen when Luke’s force state is not strong, because there is no need to train Luke when he is already with a strong force.)

**Reason:**

In star war, Ben can train Luke to raise his force to the extend that he can wield the lightsaber, and trainning is like a behavior of actor(Ben), so this is the reason why the train class should be a type of behavior class inside the package mentioned above. Also, trainning is an action that Ben do for the Luke, and thus we should put trainning behavior inside the act() of Ben.

This design relates to several OO design principles:

1. Classes should be responsible for their own properties. As the train class should only in charge of the trainning behavior of Ben.
2. Group all related classes together into a package.

Since we mentioned before, train is a behavior of Ben, so we add to the behavior package.

1. Reduece depencies as much as possible. The train class only depend the actors that relate to the trainning, and no other classes should depend on this train class.
2. Don’t repeat yourself. Because all the class and attribute inside the train class are necessary, and therefore no redundant code included inside the class.

**Advantage**:

1. Train class only responsible for its own properties.
2. No reduncdancy inside the train class.
3. No other classes depend on the train class except the actors that relate to the trainning.
4. Follow the dry principle

**Disadvantage**:

1. Serval functions from engine code are used, so the train class is dependent a lot on engine code.

* **Feature 5 Droids**

**Design choice:**

To add this new feature to the system, we decide to create a new Droid class in starwars.entities.actors package.

From the requirement of the new feature, we should never touch the forcestate attribute in SWActor even the Droid class inherits from SWActor, and we just leave the forstate’s value for droid remain none. In addition, two attributes are essential for this class, one is (SWActor) owner, and the other one is entitymanager (EntityManger<SWEntityInterface, SWLocation> em). Owner is used to indicate each droid’s owner, and the entitymanager can allow us easily to manipulate on association of both entity and their relavant location.

Firstly, we decide to randomly pick an actor as its owner from the SWWorld by a setter(setOwner()). For the droid that has its owner, it should have two main behaviors as shown below:

1. To realize the first behavior, the follow behavior:

we should compare droid’s position and its owner position by using the whereis() function through the entitymanager, if they are in the same location, then droid will stand still. Also, the droid will follow its owner when droid’s owner is in a neighbouring position, which can use a flag (boolean isNear, initially set as False) to indicate whether the droid is neighbor to its owner. If droid is near to its owner, then we set the flag to be true. Otherwise, we remain the False flag. To check whether the droid is neighbor to its owner, we can use the getNeighbor () function through entity manager, and we pass all the possible neighbor directions from the enum class CompassBearing in the Grid class. If there is one neighbor direction match the position of owner, then we let the droid move to that position by creating a new move instance and add it as a new event to the queue of droid’s events through the schdule() method inside the Scheduler class.

1. To realize the second behavior, the droid that cannot find its owner in its current or neighbouring locations:

we face the difficulity to get a random direction, as we can only get a random number straightly. Therefore, we create a RandomDirection class to solve this problem. In this class, we have a convertNumCompass method that take a integer as an argument. The method return type is CompassBearing, as we want to return a direction eventually. Inside the method, a switch statement is used to convert each integer into a specific direction (e.g,: case 1 represent North, case 2 represent Northeast, etc), and we have 8 possible directions.

Next, we can get the random number by Random class from Java.util. Since there are 8 possible directions, so we need 8 random number (1-8) by the nextInt function (nextInt(8) + 1). Next, we need to convert each random number to be a direction out of 8 possible directions. This can be done by passing each random number to the convertNumCompassing method inside our RandomDirection class.

Initially, the droid pick a random direction by above method, and keep moving in that position until it finds its owner. The seesExit from the entitymanger should be useful at this stage, as it can ensure that we always move to position that has a neighbouring location. If the droid can no longer move in its current direction, then we randomly pick a direction again and move in this direction. We repeat this process until the droid find its owner.

For the droid withou owner, it should not move, and we should use return to stop its action.

Next, since the droid will lose health when they move in badlands. We firsly need to override the takedamage in the SWActor class, to set the damage taken by droid. Then, we need to know whether the droid has moved in the badland or not, which can be implemented by a new method that pass a location of a droid as an argument and check the symbol at that position is “b” or not through the getSymbol() method in SWLocation class. If the character that method return is “b” then we know the droid has moved in the badland and it will lose hitpoints correspondingly by using the takedamage() method above. (we assume it will lose two hitpoints in each time it move in badland.).

Finally, we set the droid to stand still when its health runs out.

**Reason**:

To introduce a RandomDirection class can reduce the redundant code when we need the random direction multiple times, which is related to the Dry principle: Don’t repeat yourself.

To introduce two behaviors can make the different behaviors of Droid can only reponsible for its own properties. (e.g.: the follow behavior can only in charge of how the droid’s following its owner).

This is related to the the OO design principle: Classes should be responsible for their own properties.

**Advantage**:

1.Follow the dry principle.

2. Even it is relatively complex to implement compare to the last few features, the task that each class in charge is still clear.

3. Minimise the dependencies of other class on this class

**Diadvantage**:

1. Relatively complex to implement compare to the last few features.