Design Rationale

Leave Affordance

We added a Leave class which extends the SWAffordance class in package Actions. We did it that way because for all actors in the SWActor class, they can call the class Leave to leave the entity (including lightsaber, blaster and canteen) on the location where the actor is at.

In the class, there will be a constructor Leave() which initialize the messageRenderer and give Leave a priority of 1 by calling the super class constructor.

Also, we will have a method called canDo(actorname) which checks if the actor can leave the entity or not. First it will call the getItemCarried() method to see if the actor is carrying any entity. The getItemCarried () method will return either null or the entity name, in which we can use to check if the actor can leave the entity or not. Then a true or false will be returned.

Afterwards, the act() method will be called and it will add the entity back into the EntityManager as the actor is no longer holder the entity.

Finally, method getDescription() will return a brief statement of the action.

Advantages:

- It is easy to implement since the Leave class is similar to the Take class

Disadvantages:

- Adding a new class instead of just adding methods in the code will complicate the code

Force Ability

We added a forceLevel attribute as an integer in SWActor Class instead of adding a new class because it is way simpler and way less redundant code. We wanted to match the DRY principle. The forceLevel will be set between 0 - 100, 0 means does not have force while 100 means it has maximum force.

The first method will be getForce(). Calling this method will return the force level of a particular actor of Actor class. Not all actor can have the ability to use the Force.

The second method will be setForce(forceLevel). Calling this method will set the force level of the particular actor of the Actor class. It can be used for increasing the force level and also decreasing the force level.

The last method will be hasForce(). Calling this method will return a true or false depending whether the actor has the ability to use force or not. (Checking if forceLevel of a particular actor is 0 or not)

Advantage:

- Arithmetic Calculation can be made as integer is used
- Can distinguish between multiple levels of force ability
- Can extend to further implementations and new methods later on
- Easy implementation and keeping the code DRY
- Little repeat code

Disadvantage:

- Further implementations might not be as well as making it as a class

Ben Kenobi

We added a Train class so that when Ben and Luke are in the same location, Ben can train Luke and increase his force level to the extent that he can wield a lightsaber.

In the Train class, when the constructor Train() is called, it will first call the method getLocation() to get the current location of the player of Player class, which is Luke.

After getting the location of Luke, it will then call the getLocation() method on Ben to get the current location of Ben of BenKenobi class.

If the location of Luke and Ben are the same, it will then call the setForce(forceLevel) method from the Force class and increase the force level of Luke.

Advantages:

- More methods can be extended to this class later on as there might be more methods that Ben and Luke can do together besides training
- There might be more players later on and different player might interact with each other using this class if they are both on the same location

Disadvantages:

- Adding a new class instead of just adding a method in the BenKenobi and Player class will complex the code

LightSaber

Since only people with a lot of Force ability (though yet unspecified how much) can wield a lightsaber and use it as a weapon, we have modified the LightSaber class as well as the Attack class.

In the class, the constructor LightSaber() remains mostly unchanged except for adding a forceRequirement attribute which will be how much forceLevel an SWActor has to have before being able to use this as a weapon. In this new implementation, the forceRequirement from the Lightsaber class and the forceLevel from the Actor class will be called in the Attack class.

This is done by calling getForceLevel() on the actor class and checking if it is at least equal to the forceRequirement determined by calling Lightsaber.getForceRequirement(). if forceLevel is insufficient, a message will be printed stating that the attack was unsuccessful and nothing further will happen.

Advantages:

- Changes are easy to implement; only required modification within Lightsaber class attributes and some parts of Attack class.

Disadvantages:

- Attack class has code that only specifically applies when Lightsaber weapon is in question.

Droids

A Droid class is added which will extend the SWActor class, to which we will store an attribute called droidOwner which will be set to null upon initialization by default. The class Droid has the following methods:

- setOwner(droidOwner), to set the owner of a droid to droidOwner
- getOwner(), to return the droidOwner of a droid
- getLocation() which returns the locations of droid and droidOwner (droidLocation, ownerLocation) by calling the whereIs(droid) and whereIs(droid.getOwner()).

The Droid instance is created by Droid() to create instances of the Droid with the prescribed behaviours. A Droid instance is added to which will store an attribute called droidOwner which will be set to null upon initialization by default. The droidOwner attribute can be set by calling droid.setOwner(droidOwner), which can then be retrieved by droid.getOwner().

The location of droid and owner (respectively, droidLocation and ownerLocation) can be retrieved by calling the method droid.getLocation() and droid.getOwner().getLocation(). This method will technically be added within the SWActor class but this does not make a difference because the Droid class will be an extension on the SWActor class. By comparing the return values droidLocation and ownerLocation, we can determine whether droid and its owner are in the same location. If droidLocation and ownerLocation is the same, no further action is taken.

Otherwise, we check if droidOwner is in any adjacent location from droidLocation by calling droidLocation.getNeighbours(Direction), trying all directions until a location is found that is the same as ownerLocation as determined above. If such a location is found, the droid is moved to that location by calling the Move(Direction), where Direction is the direction of movement where droidLocation == ownerLocation.

On the other hand, if no such location is found, Move(Direction) would be called where Direction would be a randomly determined direction.

Finally, we check if droidLocation falls within the coordinates of the Badlands, and if it is we call droid.takeDamage()

Advantages:

- The creation of Droid class can allow for more unique actions or behaviours to be implemented within Droid type objects.
- Dependencies on Droid can be easily implemented and extended to work with other object classes if necessary, e.g. Attack, Blaster, Take

Disadvantages:

- Creation of new classes may be overcomplicated if there are not many additional uses/implementations for it.

Sandcrawler

When implementing the sandcrawler features, we added 5 new classes. They are Exit, Enter, Door, Sandcrawler and SWSandcrawlerWorld.

First of all, for Door, it is a class in starwars.entities package which inherit from SWEntity class. The class will call the super constructor of the superclass and make a new Entity of Door. The door class will have Exit affordance which will be used for exiting the SWSandcrawlerWorld. Also, the door has a capability of DOOR instead of other capabilities.

For SWSandcrawlerWorld, it is class in the starwars package which inherits from SWWorld. We have changed the SWWorld and SWGrid constructor by adding two coordinates (x,y). So whenever we want to make a new world, we can just add two parameters for the length and width of the grid and call the super class constructor. Also, this class has an initializeWorld method that initialize the innerworld of the sandcrawler. It will make a new door object while putting it inside the (0,0) coordinate of the grid and gives description to the grids in the world.

For Sandcrawler, this class is in starwars.entities.actors package in which it inherits the SWActor class. By calling the constructor, it will make a new actor in the SWWorld and make another world of class SWSandcrawlerWorld. The sandcrawler moves in the same pattern as Ben Kenobi every two turn.

When the sandcrawler is in the same location as the droid, the sandcrawler will check if the droid can be scavenged (Will be discussed below why we need to check if the droid can be scavenged) and if it is true then we will remove the droid from the current world entitymanager and set its' new location in the SWSandcrawlerWorld entitymanager.

When an actor of class SWActor is in the same location of the sandcrawler, the actor can choose to enter the sandcrawler by Enter Affordance. Enter is a class that inherits SWAffordance in starwars.actions package. If the actor is not dead and his/her force level is not 0, the Enter class will call the enter method in Sandcrawler which does the same thing as it did to the droid as stated above, set the messageRenderer to the innerWorld and also reset the move command for the actor. If the actor is luke, we will set the scheduler to the SWSandcrawlerWorld's scheduler and then initialize the world while rendering the uiController and ticking the scheduler.

We have also changed the Droid class so that when the droid is inside the SWSandcrawlerWorld, it cannot move. Also, if the owner is in the same location as the droid, the droid will exit the Sandcrawler the next turn.

When the droid owner wants to exit the sandcrawler, the owner needs to go back to the door's position to exit. The exit affordance does not need to check if the owner has not or not because when the owner enters the sandcrawler, we have already checked it. It will then call the exit method in class Sandcrawler, remove the actor in the SWSandcrawlerWorld and set the location to the current location of the sandcrawler in SWWorld. Also, if the owner is of class Player, we will change the ui to print the SWWorld instead of the SWSandcrawlerWorld by calling the SWGridController constructor and resetting the messageRenderer to the SWWorld one. If the owner is back in SWWorld, we will set the droid can be scavenged by the sandcrawler.

Coming back to the question above, why do we need to check if the droid can be scavenged or not? It is because when the droid owner saved his/her droid, the droid will automatically go out of the sandcrawler and the next turn, the sandcrawler can scavenge the droid again back into the sandcrawler. Therefore, we implemented a Boolean attribute that indicates if the droid can be scavenged or not. If the owner and the droid are both in SWWorld, the droid can be scavenged but if only the droid is in the SWWorld, it cannot.

Advantage:

- Easier to implement
- Further implementations or add-on can be easily implemented as we are using different classes for different jobs in the code
- Encapsulate things that must depend on each other inside the encapsulation boundary
- Minimize dependencies that cross encapsulation boundaries

Disadvantage:

- The complexity of the code is high