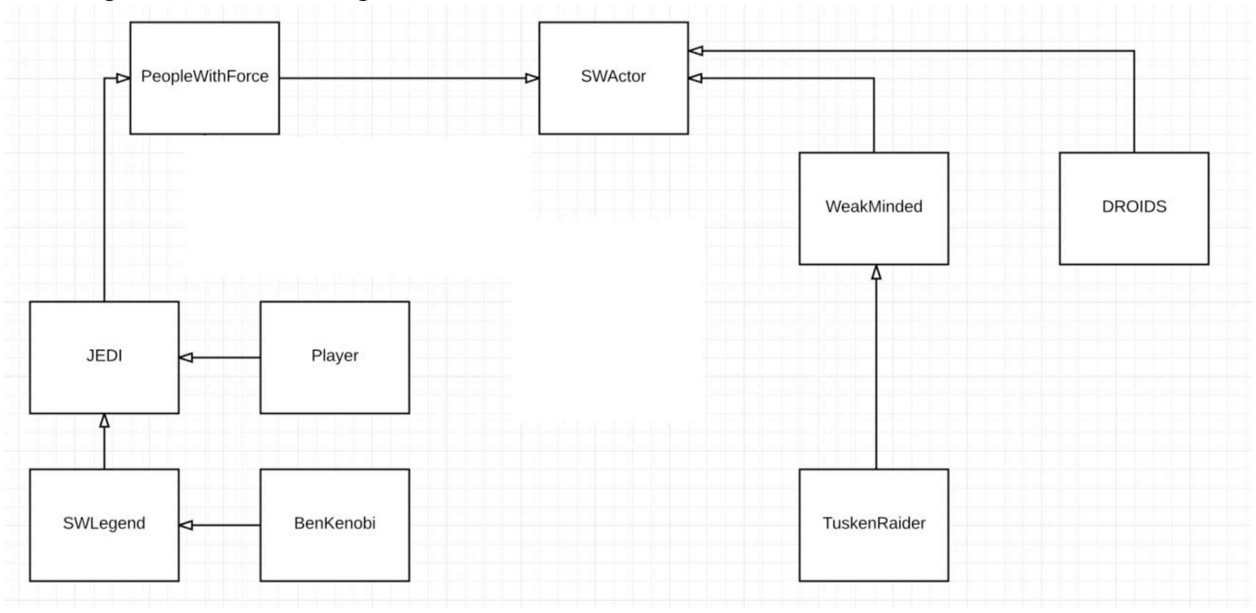
FORCE ABILITY

We will be creating new Jedi and PeopleWithForce class for extendibility. Even though these classes are basically redundant now, we are still adding these two as there might be extensions to Sith and other categories in the coming assignments. PeopleWithForce are initialised with 25 ForceAbility (more on that discussed below). Jedi are PeopleWithForce who are in Team Good while Sith are PeopleWithForce who are in Team Evil (for now, we haven't classified the team in Jedi class and it is being selected in individual players).
The design will look something like this.



- *Some people (including Ben and Luke) have the ability to use the Force.*

Force should be added under "Actions".

- *People with a little bit of Force ability can resist Jedi mind control powers.*

Characters like Ben Kenobi and Luke Skywalker will **not** have affordance to force, as they are not affected by it.

- *People with a lot of Force ability can control the weak-minded (i.e. people who can't use the Force) – that is, they can force them to attempt to move, on their next turn, in a direction of the Force-user's choice.*

Weak-minded character (e.g, Tusken Raider) on the other hand, will have affordance as they can be controlled to move by a person with lots of force.
New weak-minded character class (like SWLegends) should be created to fulfill the affordance condition.

SWActor will have ForceAbility = 0. It will have a set method to increase the force. The set method will have a condition that checks if entity belongs to weak minded. If so, it cannot be able to set the ForceAbility (ASSERTION).
0 - ForceAbility means the person has no force.

25~75 - ForceAbility refers to person with little force.
100 - ForceAbility (maximum) refers to lots of force.

So how is it all going to work

- We need a new “ForceInformation” and “ForceNieghbours” behaviours. “ForceInformation class is just like “AttackInformation” class where it has attributes “entity and affordance”. So for actors who can control the weak-minded (Ben Kenobi), inside their act method, we will have a “ForceNeighbours” method being called. This will return the actors who have force affordance attached to it and who can be controlled by people who have a lot of force (e.g, ben kenobi). A random actor may be returned if there are more than one actors in the same location who have force affordance (just like attack).

- We will add a “act()” method inside the “Force” action. This act() method will allow the actor who is using the force to move the target. So in the act() method, the target will have its move affordance removed and a new move affordance will be added to the target which will cause it to move in the specified direction of the “Player” who is using the force.
  The possible set of directions that are possible for the “target: to be moved to can be checked by looping through all the possible compass bearings and choosing anyone at random.

LIGHTSABRES

- *Anybody can pick a lightsabre up, but only people with a lot of Force ability can wield one and use it as a weapon*

Modification:

The weapon capability has to be removed from lightsaber first. By default, the lightsabre will have no capability.

The method in SWActor which is “setitemcarried” takes the actor as argument is used to carry/take an entity. Over there we need to put a condition to check if ForceAbility is equal to 100 (lots of force). If so, set the lightsabre as the item of the actor with weapon capability. Or else only add the lightsabre to the item carried of the actor without weapon capability.

BEN KENOBI

- *Ben can train Luke. Training Luke has the effect of raising his force ability to the extent that Luke can wield a lightsabre.*

Player(Luke) will start with 25 ForceAbility (Training and levels mentioned later), while SWLegends (including Ben Kenobi) will start with 100 (maximum) ForceAbility.
Each time Luke is trained, his ForceAbility will increase by 25 upto a maximum of 100 (reaching maximum) and being able to wield a lightsabre.

Train class should be added in Actions. Only Luke will have Train Affordance.
Train class will have a canDo() function which will check if player "isDead()" and if Ben Kenobi present in the same location as Luke (using the EntityManager). If so, only then the action will pass on to the "getuserdecision()" method in the "SWGridController" class. So when luke is at some location, normal user move commands will pop up on the screen but the "train" option will only pop up only when their is Ben kenobi at the same location. If the Player (Luke) enters the train option, then the action will be accordingly scheduled in the player 'Luke's' scheduler.

Inside the train action, there will be an act method which will increase Luke's force ability by 25 if and only if the force ability is not already 100. If already 100, then a message can be shown, 'Already trained till max". Also whenever the force ability increases, we need to check if Luke has a lightsabre with him. If yes then check if Luke has maximum force ability. If yes, then add weapons capability to Luke's lightsabre.

<u>**DROIDS**</u>

New class Droids will be added under SWActor. It is not being under Entities because they cannot move. For logical reason, we will add a hasMind boolean in SWActors which will be True but will be set to False for Droids.

- *Droids can't use the Force.*

Since a droid does not have a mind to begin with, it will not be able to use force. It will not be affected by force either as it has no Force affordance

- *Droids have owners.*

Droids will have a constructor which will have an actor attached to it. An actor will now have two constructor, one without a droid and one with a droid. So an actor will have a droid instance variable and a boolean if it is owning any droid at the moment or not. So a droid object will be passed to an actor's constructor if needed.

- *Droids follow their owners if they are able to do so.*

When an actor who owns a droid moves somewhere, inside the 'Move' action class, the act() method inside the move class will be modified so that when an actor is scheduled to move, it checks if the actor has a droid owned. If yes, then the actor moves and also the droid will be moved in the same direction by using the "moveEnitity()" method in the SWWorld class.

There will be some checking need to done.
For example, if the droid is immobile or not which will then cause the droid to stay at the same place and the owner moves on. The owner may still own the droid.

- *Droids lose health when they try to move in Badlands.*

A checking needs to be added in the act() method of Move to check the location of the droid using whereis() method from SWWorld. If returns true, hit points will be deducted by a certain number each turn that it remains there.

- *Droids regain health if they use oil, or if somebody else uses oil on them.*

Regarding droids repairing themselves, this is not yet implemented. For that we will need a separate act() inside droid which can be activated for example if its owner dies and the droid is able to move on its own.

A new capability 'Droid Usable' can be introduced.

'OIL CAN' entity will extend from 'SWEntity'.
New entity 'OIL CAN' can be made where each 'OIL CAN' will have a 'Droid Usable' capability.
A new action is introduced 'UseOnDroid'.
So the 'OIL CAN' entity will also have a 'UseOnDroid' affordance.

So the canDo() method in the 'USE ON DROID' will check if the item carried by the actor is an OIL CAN and also check if there is a droid in the same location. If yes, it is checked whether the droid is not immobile or not, if so, true is returned. Then a command for that for example 'Use Oil on Droid' ,if the command is pressed, the a 'USE ON DROID' action is scheduled which has an act() which will be called by the scheduler.

Inside the act(), a random local (same location as the actor) droid will be selected and their hit points will be increased by a certain amount.

- *Droids don't die, but they become immobile when their health runs out.*

Like mentioned earlier, checking will be done in the act() method of the Move class. So when the actor is trying to move, it will be checked if the droid it owns (if it owns) has hit point which is less than or equal to zero. If yes, then the actor continues his movement in the specified direction leaving the droid on the map where it is.

A new instance boolean "immobile" can be added to the Droids which shows True or False depending on the HP.

Also for now we are assuming the droid does not move on its own. But when possible, all this can be achieved by specifying what the droid will be able to do in its 'act()' method. So for now, the droids act() method does not do anything. Everything goes through its owners.

- *Immobile droids can be disassembled into droid parts. Some people know how to repair immobile droids.*

New class Droid Parts will be added. It will extend from SWEntities. It will have DROID USABLE Capability and USE ON DROID affordance. It will also have an 'TAKE' affordance.
New Action "DISASSEMBLE" will also be introduced.
This will be added to the droids as their affordance.

In "DISASSEMBLE"'s canDo() method there will be checking done to see if local immobile droids present (For now assuming that everyone can repair/disassemble a droid). If so, returns True. Command on display shown.
If command chosen, job is scheduled. Then the act() method in the DISASSEMBLE class will be called. If more than one immobile droid present, one is randomly chosen. The chosen droid is then removed from the map using removeEntity in the entity manager and on the same location a new droid part instance is created and is set inside the EntityManager.

For future extension, not everyone will be able to disassemble a droid. So for that, a new capability may be introduced like 'REPAIR' which can be added to certain actors. Inside the canDo() method of the 'DISASSEMBLE' class we can check if actors trying to disassemble have the 'REPAIR' capability.

- *Immobile droids can be repaired by using droid parts on them, which uses up the droid parts.*

So the canDo() method in the 'USE ON DROID' will have further checking now (extending from the OIL CAN part) which will check if the item carried by the actor is an DROID PARTS and also check if there is an immobile droid in the same location. If yes, true is returned. Then a command for that (e.g. 'USE OIL ON DROID') is displayed ,if the command is pressed, the a 'USE ON DROID' action is scheduled which has an act() which will be called by the scheduler.

Inside the act() method of 'USE ON DROIDS', first we will set the item carried by the actor to null to provide the illusion that the droid parts has been used on the immobile droid. A random local (same location as the actor) immobile droid will be selected and its "IMMOBILE" boolean will be set to False. HP of that droid will be set to half of maximum.

HEALING

- *Drinking from a canteen should heal the drinker a little bit.*

Canteen will have an instance named HealPoints which contains an integer which determines by how much the Player will be healed.

A new "DRINK" affordance can made in the actions folder. All entities that have the 'FILLABLE' capability will also have a 'DRINKABLE' capability. The "DRINK" class will have a canDo() method , it will see what item the actor has and will see if the item has "DRINKABLE" capability. If yes also it is checked if the 'target(e.g. canteen)" is not empty. If these conditions are fulfilled, true is returned and then this action can be added to the command line of Luke. If the user selects the option to drink, the drink action will be scheduled to the scheduler. The act method in the drink action will be called. Then the hit points of the actor will be increased. For this to be achieved, we need to create a new method inside SWActor setHitPoints() which will set the SWActor's Hit Points. And by retrieving the original Hit Points from the Player using the getHitPoints(), we will set the new Hit Points by calling the method the Player setHitPoint(hitpoints + canteen.HealPoints).Then a new method in the canteen class called 'ReduceLeve()' which will reduce the level of the canteen by 1.

- *Droids regain health if they use an oil can, or if somebody else uses an oil can on them.*

--MENTIONED IN THE DROID SECTION--

- *This should not deplete the oil completely; unlike the canteen, there's no way to refill the oil can.*

For refilling/filling of canteen

In player's act() method, all actions that are possible are iterated from a list. From there, by a canDo() method, it is checked whether the action is possible. Canteen has a fill affordance.

In the Fill class, the canDo() method takes actor as an argument and checks if the itemCarried by actor is a canteen or not. If it is a canteen, it checks whether if it has a capability of Fillable; finally checks if the canteen is not full. If the above conditions are met, then only the method returns true. Then the action is displayed as an option in the command line.

When the command is chosen to fill, the scheduler will take the job and when it is executed it will go to the fill() method of canteen, where it will make the level of canteen equal to capacity (maximum).

The modified class diagrams have been submitted separately.

Below are the new classes with their responsibilities.

Quite a few changes have been made to the existing classes, but these are not shown below. This is because they have been covered above point-wise in great detail.

The places where we have kept the variables spot in the class diagram blank is because the there are no new variables for the newly introduced class. It will use the variables from their extended classes if needed. Also the methods for these classes are actually common methods which we just need to override to suit our purposes.

DECIDER CLASS FOR WIN/LOSE (HAS BEEN MENTIONED IN JAVADOC AS WELL)

```
/**
 * This class provides a mechanism for a client to know if their game
is over or not. This class is used in
 * relation with the GameOver class. The decider constructor takes a
GameOver object as parameter in constructor.
 * A client can actually make a decider for their own game and extend
this class. Also they can make their own GameOver
 * and pass it to their decider.
 *
 * This class has a canDecide method which the client can loop on
until the decider can decide if the game is over or
 * not. If decider can decide, it will return true else it returns
false. It decides by using the win() and lose()
 * method in the gameOver class.
 *
 * This class help's the client to easily develop a win and lose
situation for any game. All the client needs to do
 * is implement the win and lose methods and use the decider class to
know when the game is decided or not.
 *
 * To make full use of this class, the client needs to use this class
in relation with the GameOver class.
 * @see {@link edu.monash.fit2099.simulator.space.GameOver}
 */
```

GameOver CLASS FOR WIN/LOSE (HAS BEEN MENTIONED IN JAVADOC AS WELL)

```
/**
 * This class basically gives a mechanism so that when each client
game uses this engine, they can extend or use this
 * class to write ways in which their game is won or lost. By default
the win() and lose() method returns false.
 * If any client game wants to write their own code for how their game
will be won or lost, they need to create a new
 * class which extends this class and override the win() and lose()
methods. Also they need to specify the reason for
 * the game over .
 *
 * To make full use of this class, the client needs to use this class
in relation with the Decider class.
 * @see {@link edu.monash.fit2099.simulator.space.Decider}
 */
```

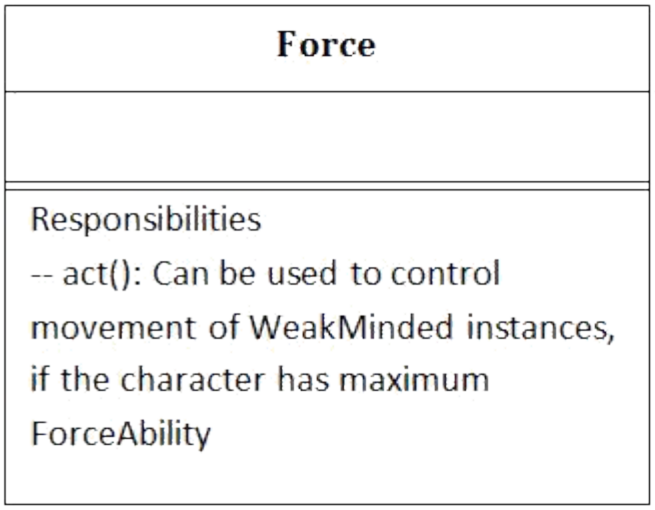
Droids

-Immobile: Boolean
-hasMind: Boolean

Responsibilities
-- Follows owner
-- Does not die when Hit Points reach 0, becomes immobile (means stays on the same place on the map)
-- Can be made mobile again using DroidParts
--Immobile droids can be disassembled into DroidParts

OilCan

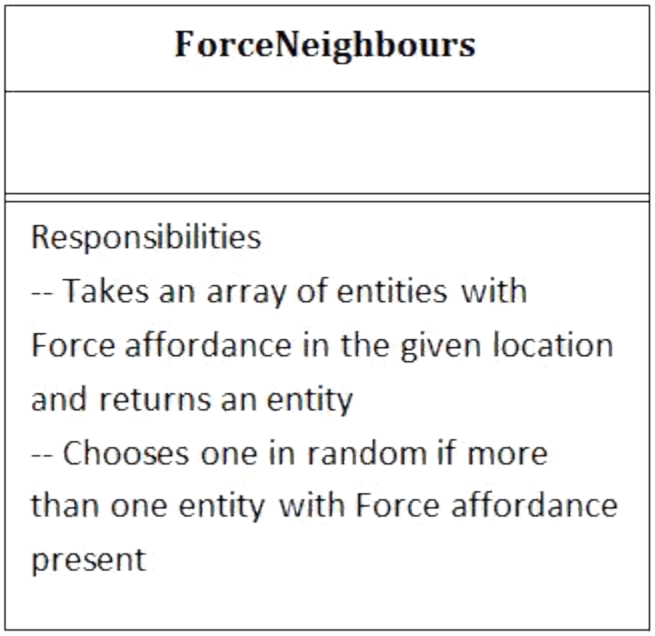
Responsibilities
-- Used on droids by a player to regenerate droid’s hit points



Force

Responsibilities

-- act(): Can be used to control movement of WeakMinded instances, if the character has maximum ForceAbility



ForceNeighbours

Responsibilities

-- Takes an array of entities with Force affordance in the given location and returns an entity

-- Chooses one in random if more than one entity with Force affordance present

DroidParts
Responsibilities: --This is used to fix the immobile droids. It will have a new affordance attached to it called ‘UseOnDroid’ and a new capability ‘DroidUsable’. --This helps us to identify which items can be used on droids.

WeakMindedOrganics
-forceability: int
Responsibilities: --This class extends the SWActor class. WeakMindedOrganics have force ability 0. They can be controlled by people who have lot of force --From default, they have a ‘Force’ affordance attached as they can be controlled.

ForceInformation

-entity: SWEntityInterface

-affordance: Affordance

Responsibilities:

--This class is responsible for storing objects which will generally be actors who have a force affordance attached to it.

-- This class is used by the 'ForceNeighbours' class

UseOnDroids

Responsibilities:

--This will be attached to all entities who have an 'DroidUsable' capability like 'OIL'

--They will have an act () method which will remove the droid parts from map and increase the droid health. The act() method will take a SWActor who is carrying the parts.

DRINK

Responsibilities:

--This class is responsible for allowing an actor to use the drink action which is attached to entities like canteen.

--act () method in the DRINK class will check if the canteen is not empty and then increase the hit points of the actor trying to drink.

--new reduceLevel () method inside the canteen class will be called to reduce the level of the canteen.

Train

Responsibilities:

--This affordance can be attached to Luke so that he can be trained by Ben Kenobi.

--There will be a act() method which will increase Luke's forceability by 25 if and only if it's not already 100.

--if 100, then if Luke is carrying an light-sabre, add WEAPON capability to it.

Disassemble

Responsibilities:

--This will be added to all the droids as their affordance.

--They will have an act () method which will remove the droid from the map and put droid parts in that location.

--For now all actors can disassemble but there can be an extension adding 'REPAIR' capability to certain actors.

PeopleWithForce

forceability: int

Responsibilities:

--This class will extend the SWActor class. Classes which extend this class will have an initial force ability of 25. This indicates they at least have little force to start with.

AdmiralAckbar
-admiral: AdmiralAckbar
Responsibilities -- Extends SWLegend -- Has a 10% probability of saying “it’s a trap” --Only one AdmiralAckbar can be created like all SWLegends

C3PO
-C3POquotes: ArrayList<String>
Responsibilities -- Extends Droids -- It doesn’t move and stays in the same location. --It has a 10% chance of saying something at each turn.

PrincessLeia

-leia: PrincessLeia

Responsibilities

-- Extends SWLegend

-- It doesn’t move and stays in the same location.

--It follows luke after luke is in the same location

R2D2

-path: Patrol

Responsibilities

-- Extends Droids

-- It moves 5 steps east and 5 steps west

--It has a internal oil reservoir and can heal other droids

--Also can repair droids with droid parts

Stormtrooper

Responsibilities
-- Extends WeakMinded
-- It has 5% chance for calling for back up

--It has a blaster

--Moves randomly

--Tendency to miss the target when attacking

CallForBackup

Responsibilities
-- Extends SWAffordance
-- target is a Stormtrooper which has called for backup

--A new Stormtrooper is created in the same location

FLY

Responsibilities

-- Extends SWAffordance

-- Asks for user input for which grid to fly to

--All actors following luke move to the new grid.

ForceChoke

Responsibilities

-- Extends SWAffordance

-- All actors except droids can be force choked by darth vader

--targets hitpoints reduce by 50.

SWDecider

Responsibilities

-- Extends Decider

-- It takes a game over object in its constructor .

--This class decides when the star wars game is over

SWGameOver

-world: SWWorld

Responsibilities

-- Extends GameOver

-- It has two methods win and lose

--If both win and lose returns false, the star wars game is still not over

Decider

-gameover: Gameover

Responsibilities
-- It has a canDecide() method which returns true when a game can be decided

--Uses the game over win() and lose() method to decide the state of any game.

GameOver

-ReasonForGameOverL String

-world: SWWorld

Responsibilities
-- It has a win() and lose() method which by default returns false.

--Any game who needs to write its own win and lose should extend this class and override.

MilleniumFalcon

Responsibilities

-- Extends SWEntity

-- It is used to fly to different grids.

--it has a “FLY” affordance attached to it.

DarthVader

-darthvader: DarthVader

Responsibilities
-- Extends SWLegends
-- It tries to convince luke.

--Also if not convinced, it attacks luke

--Also it can force choke other actors

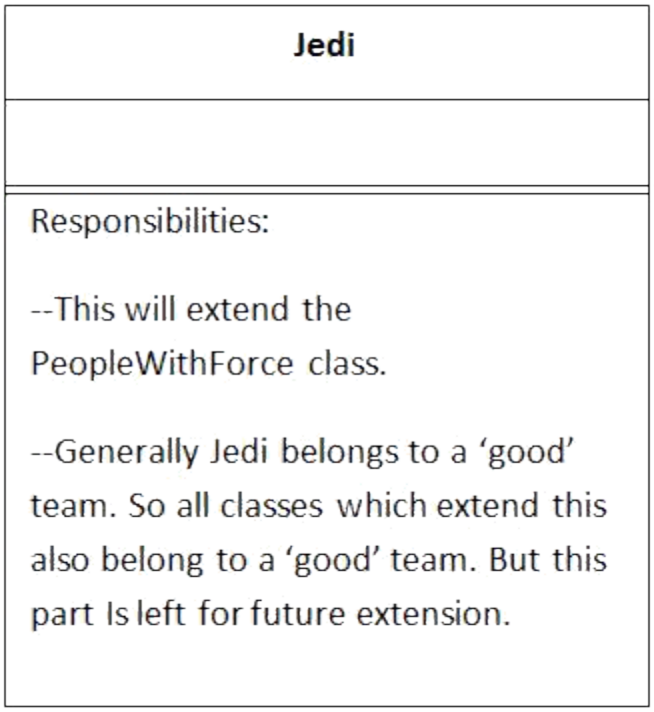
moves randomly.

MonMothma

-hasSaid: Boolean

Responsibilities
-- Extends SWLegends
-- It doesn't move and stays in the same location.

--It says a quote if at yavinIV luke lands without princess leia and r2d2.



Jedi

Responsibilities:

--This will extend the PeopleWithForce class.

--Generally Jedi belongs to a ‘good’ team. So all classes which extend this also belong to a ‘good’ team. But this part Is left for future extension.