**Assignment 3 Rationale for new features**

**Feature 1 Reservior**

**Design choice:**

To realize the leave affordance feature, the Reservior class is modified in the following ways:

* Add the inherited attribute hitpoints and set the default int attribute hitpoints to 40 as the default hitpoints are supposed to be 40.
* Add a new Boolean method descriptionChanged() using the methods getLongDescription () and getShortDescription () which returns true if the descriptions has been changed, False otherwise.
* Add the method takeDamange() which overrides the method in super class.

The method takes in an int paremeter which indicates how much the damage actually is and calls the method in the super class SWEntity which will set the hitpoints to hippoints-damage.

After that, the method descriptionChanged() is called to check if the shortDescription and the longDescription has already been changed. If false and the hitpoints is dropped below 0 or 20 which combined indicating that the hitpionts is changed for the first time therefore the shortDescriptionand and longDescription needed be changed as well as the symbol by calling the setShortDescription(),setLongDescription() and setSymbol().

This design relates to several OO design principles:

* Group all related classes together into a package.

Since we group the entity Reservior into the entity package, and group the throw action into action package.

* Avoid excessive use of literals.

In the takeDamage() method, we set hitpoints threhold extents as static final int attirbutes instead of using figures to represent them.

* Classes should be responsible for their own properties.

As Reservoir class should only in charge of the Reservoir entity.

**Advantages:**

* Easy to implement
* Without the need to change any other class than Reservior, an implementation without any unnecessary dependencies.
* Following DRY principle
* Follow all other princinples mentioned above.

**Disadvantages**:

* There are limitations due to the implementation, for example the Reservoir can’t be damaged after their hitpoints are below 0.
* This implementation still remains very trivial, It can’t do much except change the symbol or the description. It might be troublesome to scale this class in the future.

**Feauture 2 Grenade:**

**Design choice:**

To realise this feature, we need two new classes: Grenade added in package starwars.entities, and Throw in starwars.actions.

For Throw class, it inherits from SWAffordance and implements SWActionInterface, which allows it to overide some classes(canDo(),act() ) from the interface and also overides an abstract class from its ancestor class (getDescription()).

The constructor of the throw class should create an instance of throw affordance.

For the canDo(a::SWActor) method, it specify the only item(entity) carried by actor can perform this affordance is grenade, so it check whether the item carried is grenade or not.

For the act(a::SWActor) method, it will be implemented in detail as following steps:

1. It will check if the item carried by actor is grenade or not. If yes, then it will first collect all the directions in enum class compassBearing().
2. We should get the location of actor a by calling the function whereis(target) from entitymanger.
3. Then, we should get all the entities in that location(a included) by passing the location of actor a to the function contnets() from entitymanager.
4. Since the contents() will return a list containing all the entities in that location, so we can go through each entity in that list and check if there is any entities except actor a in the same location where the grenade is thrown, if there are some, then we will set those entities to take damage 20. In here, instead of using a figure 20, we will create a static final int attribute maxExplosion = 20, to represent this extent of damage to reduce the dependencies on this figure, and it is also related to a design principle: Avoid excessive use of literals.
5. Next, we will loop through the directions we collect in 1, and for each direction, we use getNeighbour() from Location class to get the neighbouring location in that speficied direction(i.e: can be reach in one step from the location where the grenade is thrown). From this neighbouring location, we can use contents() again to get a list containning all the entities in this location, and loop through the list to cause damage for every entity in this location, in here we will set the damage extent as 10, which will also be represented as a static final int attirbute intermediateExplosion = 10.

In addition, we also use getNeighbour().getNeighbour() for each direction, to get its neighbouring location of neighbouring location(i.e.: can be reach in two steps from the location where the grenade is thrown). From this location, we can use contents() again to get a list containning all the entities in this location, and loop through the list to caiuse damage for every entiity in this locatiion, in here we will set the damage extent as 5, which will be represented as a static final int attribute minExplosion = 5 as well.

1. Since after throwing the grenade, the grenade should be destroyed and disppear in the world, so we remove the grenade from the world, by SWAction.getEntityManager().remove(target).

Also, we set the item carried by the actor to null correspondingly.

Finally, we need to override the getDescription() to return the relative description of the throw action for display.

For Grenade class, it is similar to other entities like blaster or lightsaber, it inherits from SWEntity and it needs to overide some abstract methods (getshortDescription(), getlongDescription()) from that its ancester class implment abstract class. Also, the grenade class needs to add two affordances: Take and Throw, which allows the grenede perform its natural functionalities. (can be picked up and throw to a position and casue damage.)

**Reason:**

A grenade is like a weapon like entity such as blaster and lightsaber, but with different types of use, which is the reason why the Grenade class should be a type of entity class inside the package starwars.entities. A throw is like a action that taken by actors such as leave and take, so we decide to group it with those actions together into the starwars.actions. Since the grenade is the only entity can perform throw action, so we specifically check the item carried by actor is Grenade to perform throw.

This design relates to several OO design principles:

1. Group all related classes together into a package.

Since we group the entity grenade into the entity package, and group the throw action into action package.

1. Avoid excessive use of literals.

In the steps of 4,5,6 explanations for act() method, we set damage extents as static final int attirbutes instead of using figures to represent them.

1. Classes should be responsible for their own properties.

As grenade class should only in charge of the grenade entity, similarly, for the throw class, which should only deal with the throw action.

1. Reduce depencies as much as possible.

The throw action only depends the item that carried by actor is grenade, and no other classes should be depended by the throw class. Similary, the grenade only need to depend on two affordance classes:take and throw, to perform its natural functionalities and no other classes should be depended by grenade class.

1. Don’t repeat yourself.

Because all the methods and attributes in both grenade and throw classess are necessary, which means there is no redundant code inside both classes.

**Advantage:**

1. Minimise the dependencies on other class for both grenade and throw classes, also minimse the dependencies on the figure, by using the static final int attributes.
2. Follow the dry principle.
3. Follow all other princinples mentioned above.
4. It is easy to implement.

**Disadavantage:**

1. Serval functions from engine code are used, especially the throw class, so this will increase the depencies on engine code for throw class.

**Feauture 3: JAWA Sandcrawler**

**Design choice**:

To realize this feature, we need to modify three classes: SWWorld, SWGridTextInterface, SWGridController.

* For SWWorld class, we add a new attribute interiorGrid with type SWGrid, and initialize this attribute in the constructor of SWWorld class. In addition, we add a getter for this attribute and add two getters to get the height and the width of this attribute(InteriorGrid). Also, we set a door symbol inside this interiorGrid in coordinates (4,3).
* For SWGridTextInterface class, we have added a static SWGrid attribute called interiorGrid, and initialize it in the constructor of this class. Moreover, we add a displaymapgenral() method which renders the map into a text format for later output.

The displayMap() has been modified to collect the widths and heights of two grids and display them by calling and passing them into the displaymapgenral() method.

* For SWGridController class, we add a new attribute interiorGrid with type SWGrid, and initialize this attribute in the constructor of SWGridController class.

Then, we create a new class Sandcrawler in the package:starwars.entities.actors. There are 5 attributes in the class:

1. Decisionarray: decision array used alternate between two scenarios.
2. Decision1: decision int attribute representing first out of two turns
3. Decison2: decision int attribute representing second out of two turns
4. Door\_x: x coordinate of the door
5. Door\_y: y coordinate of the door
6. Path: decision Type Patrol attribute used to store the movements that Sandcrawler follows
7. InteriorGrid: SWGrid type attribute representing the interior grid inside the </code>Sandcrawler</code>

Also, the class has its own constructor and 3 other methods:

Sandcrawler():Construtor which creates and initialize an Sandcrawler object.

act():This method allow the </code>Sandcrawler</code> to act as per specification

first, the method will check the location of which that Sandcrawler is in, if a Droid happens to be in the same location, the Droid will be taken into the interior grid by calling the setInterior method. Then the method will check if a SWActor who could use force is in that location, if there is, the SWActor will  be taken into the interoir grid by calling the setInterior method. After that, the method will collect all the SWEntityInterface entities that is the same location as the door symble'd' if there is SWActor with enough force in it , the actor will be taken out of the interior grid and put back into the original grid. After all that, the method will check if it is still alive, if Sandcrawler is, then move otherwise stay by calling the method alternateDecisionArray.

setInterior(): This method will take the input SWEntityInterface object out of the original grid and set the input SWEntityInterface object into the interior grid.

alternateDecisionArray(): This method keeps the decision array at size two and alternate between the two decision integers.

**Advanatge**:

This design relates to several OO design principles:

* Group all related classes together into a package.

Since we group the actor sandcrawler into the actors package.

* Classes should be responsible for their own properties.

The sandcrawler class is only in charge of sandcrawler actor.

* Reduce depencies as much as possible.
* Avoid excessive use of literals.

Using constants to represent the actual figures. (e.g.:decision1,decison2, door\_x, door\_y)

* Follow the dry principle.
* It is easy to implement.

**Disadavantage:**

* The coordinates of the door has to be physically documented inside the Sandcrawler class which makes it somewhat hard to change in the future
* This implementation can’t allow a SWActor to be not taken in when it is having an encounter with Sandcrawler. Therefore there won’t be an option for the SWActor if it wants to be taken into or not.