Design Rationale : FIT 2099

This rationale explains the different classes that are going to be in our game. It also explains in detail what each class does and the relationship between the classes and what are the methods and attributes present and how they are manipulated to achieve the result we want..

**Door Class:**

Class Responsibility:

This class holds the ability to allow a player to pass through it only if the player has a key(an instance of the Key class) in their inventory. We decided to have unique keys for each door, therefore to enter a particular room we need its equivalent key(1 Door has 1 Key relationship)

Relationships With  Other Classes:

We observed that the Door class would require attributes and methods from the Ground class(which is a generalized class of terrain types- eg: Floor class & Wall class) as it has all of the methods(Need to Override) we require for our Door class. This way, we follow the DRY principle and eliminate the need to have the same code.

The door class will therefore extend(inherit methods and attributes) Ground class present in the game engine.

Methods/Attributes:

- Inherited attributes from Ground Class

+ Inherited Methods from Ground Class

Overridden Methods:

+canActorEnter(Actor actor) : Boolean

         If key instance is present in inventory of Door.:

                   Returns True

         Else:

                   Return False

+blocksThrownObjects() : Boolean (Made True)

Addon Methods/Attributes:

- key(Instance of Key Class)

+isUnlocked(): bool

If the door has been opened by the player before,  Return True. So it doesn’t need to be opened again

Else: Return False (Indicating the Door has **not** been unlocked yet)

**Key Class:**

Class Responsibility:

In order to obtain a key, the player must defeat an enemy. After the enemy has been knocked out, the enemy will drop a key. This key is used to open a door. Each Key, which is an object of the Key class, has a attribute called the keyID which is used to uniquely identify each key. The keyID matches with a specific doorId so that, that key can be used only to open that particular Door.

We create a new instance item “newKeyInstance”, this way we can easily give the keys to the enemy characters.

Relationships With  Other Classes:

The key class inherits from the item class because when we thought about the implementation we realised that we needed all the attributes and methods used in item class, mainly, the displayChar attribute, the allowable actions attribute, the getAllowableActions method and the newInventoryItem method. This is because when we went through the code given for the Item class, the getAllowableActions and the newInventoryItem  method already implements a way for the player or enemy to pick up an item and drop an item . This will be useful to us when player defeats an enemy (Grunt,Goon,Ninja) and drops the key into the location where the enemy was. By inheriting methods and attributes from Item class, we are yet again avoiding duplicated code(DRY principle)

The Key class has a one to one relationship with the door class and it is a dependency relationship because the door (a locked one) cannot exist without a key according to our understanding and our plans of implementation of the game.

Methods/Attributes:

- Inherited Attributes from the Item class.

+ Inherited Methods from the Item class.

Addon Methods/Attributes:

+ keyID: String

**Goon Class:**

Class Responsibility:

The Goon Class is used to implement an enemy character. The Goons can either attack the player with twice as much damage as Grunt or the Goons have a 10% chance of shouting insults at the player. The Goons also follow the player like Grunt.

Relationships With  Other Classes:

The Goon class will extend(inherit methods and attributes) Actor as it is an enemy with all required attributes of the Actor class. The Goon class also requires all the methods (like isConscious, hurt, getIntrinsicWeapon) in Actor class to be implemented properly in the game.

The Goon has two behaviors insultBehaviour and FollowBehaviour. The way FollowBehaviour is implemented for Goon is the same way that it is implemented for Grunt so it is not shown in our class diagram. However, the insultBehaviour is a new Behaviour so we showed it in our class diagram. The Goon class uses ActionFactory interface to generate actions to perform, we show this using the simple association relationship. The Goon Class also can hold a key which is shown by a simple association relationship. The Goon Class is also dependant on Action class because the Goon Class cannot exist without an Action to perform.

Methods/Attributes:

-Inherited attributes from the Actor Class

+ Inherited methods from the Actor Class

Overridden Methods:

+playTurn() - currently this method randomly chooses an Action to be performed by the user. This method should be overridden in a way that an appropriate action is performed at an appropriate time. Eg. Either follow, attack or insult the player depending on the players location.

We also remove the ability of the Goon to drop items in order to make sure he doesn’t drop the key before it is defeated.

+getIntrinsicWeapon() -  this method should be overridden so that the Intrinsic Weapon’s damage points and the name of it can be changed so that the damage of the Goon is twice of Grunt.

**InsultBehaviour Class (Implements ActionFactory interface):**

Class Responsibility:

The InsultBehaviour class is used to print an insult only with a 10% chance at each turn. The way we decided to implement InsultBehaviour is that it has an attribute called insults which is an arraylist of strings with different insults. Every time the InsultBehaviour class is called, we will obtain a random insult by getting a random number between 0 and 1 and using it to get an insult from the arraylist of insults and print it.

Relationships With  Other Classes:

The insultBehaviour class inherits from Action class as it is an Action that can be performed by the Player and it has all the required methods for us to implement the class. The class InsultBehaviour, implements ActionFactory so that it can use the getAction() method.The getAction() method is overridden in order to return steps to achieve our behaviour., in this case throwing insults at the player. We show that InsultBehaviour implements ActionFactory by using a interface realisation arrow.

Methods/Attributes:

- Inherited Attributes from the Action Class.

+ Inherited Methods from the Action Class.

+ distance()

- insultChance: float

- Insult: String

- distanceBetweenActors: int

Overridden Methods:

+ execute() - override this method so that at each call to InsultBehaviour a random insult from the arraylist of insults is printed on the console.

+getAction() - override the method so that when the player is within the range(will be decided later) of the goon then the goon calls the execute() method which prints an insult on the console.

**Ninja Class:**

Class Responsibility:

The Ninjas have the ability to stun the player(using StunBehaviour) but there is only a 50% chance of that happening. When the player is stunned he cannot perform any actions for two turns. After throwing stun powder at the player, the ninja moves back one step.

Relationships With  Other Classes:

The relationships of Ninja Class are the same relationships of the Goon class as they are both enemies the only difference is that Ninjas don’t have FollowBehaviour but we used the code from followBehaviour to check if player is within five steps(complete stun) so we omit that relationship for Ninja. We get the location of the player and Ninja in the map

The Stun behavior of ninja allows the ninja to see if player is within 5 squares North, South, East or West of him and moves the ninja away from the player accordingly and returns the Stun behavior which is an action which implements action factory. There is only a 50 % chance of player getting stunned for two turns.

Methods/Attributes:

- Inherited Attributes from the Actor Class.

+ Inherited Methods from the Actor Class.

Overridden Methods:

+playTurn() - currently the playTurn method chooses a action randomly from a list of possible actions for the actor. We need to override it for Ninja in a way that there is only one action which is only performed if the player is within 5 squares of Ninja. There are two actions, throw stun powder and move one step back but it should be implemented as one for the sake of simplicity.

**StunBehavior Class: (Implements ActionFactory interface)**

Class Responsibility:

The stunBehaviour class is used to stun player with only 50% chance of stunning the player. The way we decided to implement stunBehaviour is, there is another class called stunPowder which is a weaponItem object which is present in the inventory of Ninja. Each time the stunBehaviour class is called it checks the Ninjas inventory for stunPowder and stuns the player for two turns but the probability of that happening is only 50%

Relationships With  Other Classes:

The stunBehaviour class has the same relationships with the classes from the engine as the insultBehaviour class so we will not go into detail about it. However, we are explaining the relationship of the stunBehaviour class with non-engine classes below.

The stunBehaviour class is dependant on the stunPowder class because if Ninja doesn't have stunPowder he cannot perform stunBehaviour. This is shown in the diagram as a dependency relationship with a normal open arrow with dashed lines. Ninja has stunBehaviour which is shown as a simple association.

Methods/Attributes:

- Inherited Attributes from the Action Class.

+ Inherited Methods from the Action Class.

Overridden Methods:

+execute() - this is overridden in a way to show a description that the Ninja stuns the player.

+getAction() - this is overridden in a way that the methods gets the allowable actions of the player and sets it to null for two turns.

**Stun Powder Class(Implemented but did not have enough time to use it)**

Class Responsibility:

The objects of the Stun Powder class are held in the the Ninjas inventory. The Stun Powder has no damage but it Stuns player for two turns.

We create a new weaponIteminstance of this item “newInventoryStunPowderInstance”, this way we can easily give the weapon to the enemy characters.

Relationships With  Other Classes:

The Stun Behaviour Class is dependant on the Stun Powder class because the Ninja cannot perform Stun Behaviour without the Stun Powder. The Ninja holds instances of the Stun Powder class in its inventory which is shown by an simple association relationship.

Methods/Attributes:

- Inherited Attributes from the WeaponItem Class.

+ Inherited Methods from the WeaponItem Class.

Overridden Methods:

+execute() - this is overridden in a way to show a description that the Ninja stuns the player.

+getAction() - this is overridden in a way that the methods gets the allowable actions of the player and sets it to null for two turns.

**Q(NPC) Class:**

Class Responsibility:

The Q actor is a NPC (Non-Player Character). The Q class is an important class as it communicates with player(Talk Behaviour) and it gives the Rocket Body in turn for the Rocket Plans. The Other Actor uses give plans action to exchange RocketPlans for RocketBody.

Relationships With  Other Classes:

The Q class inherits from Actor class because it is also a character with hitpoints, a display character and all the other attributes any other actor has. It also uses the methods of the actor class therefore it inherits from Actor class. The givePlansAction class is dependent on the Q class because the givePlansAction cannot take place unless the Q is there to receive the plans. Q holds the Rocket Body which is shown with a simple association relationship. The Talk Behaviour is used by Q and their relationship is shown with a simple association. The Q class uses Action Factory to generate a list of Actions to be performed. We gave Q a very high hitpoint in order to prevent him from being killed in the game.

Methods/Attributes:

- Inherited Attributes from the Actor Class.

+ Inherited Methods from the Actor Class.

Overridden Methods:

+playTurn() - this method is overridden to remove AttackAction and PickupItemAction from Q.

+ getAllowableActions() - this method is overridden in. a way that if the player has the rocketPlans in his inventory, we will add the two behaviours, (TalkBehaviour and GivePlansAction) to be executed. Otherwise only TalkBehaviour will be executed with Q saying “I can give you something that will help, but I’m going to need the plans. “

**TalkBehaviour Class:**

Class Responsibility:

This class is used to allow the interaction between Q and the player. Q should be able to talk to player when it comes into contact with the player. If Q finds the RocketPlans in players inventory, he will promt the player to give it to him in exchange for the RocketBody by saying “Hand them over, I don’t have all day!”, if the player does not have the RocketPlans, Q will say “I can give you something that will help, but I’m going to need the plans.” and the player should move on and get the RocketPlans.

Relationships With  Other Classes:

It is the similar to insult behaviour except for the fact that the behaviour is specific to just Q.

Methods/Attributes:

- Inherited Attributes from the Action Class.

+ Inherited Methods from the Action Class.

Overridden Methods:

+execute() - This is overridden to show Q interacting with Player

+getAction() - his is overridden in a way that the methods gets the allowable actions of the player and gives player the option to give the RocketPlans to Q if he has already collected it.

**Doctor(Doctor Maybe) Class:**

Class Responsibility:

The Doctor Maybe character is an enemy character who is like the Mini Boss of the game. The role of the Doctor is to be placed inside a locked door whose key can be obtained by fighting a enemy. The Doctor can also fight but is pretty weak (half hit points and damage of grunt). The Doctor holds the Rocket Engine.

Relationships With  Other Classes:

The Doctor also inherits from actor class like any other enemy because all the attributes of actor class are vital when implementing a character in the game. The Doctor has a simple association relationship with Rocket Engine because it is held in the inventory of the Doctor. The Doctor is placed into a Locked Room in the application class which initializes the map.

Methods/Attributes:

- Inherited Attributes from the Actor Class.

+ Inherited Methods from the Actor Class.

**RocketPlans Class:**

Class Responsibility:

The RocketPlans is an object of RocketPlans class, which needs to be given to Q(the Non-Player Character) in order to obtain the “Rocket Body” object.

Relationships With  Other Classes:

The RocketPlans class inherits from the item class because when we thought about the implementation we realised that we needed all the attributes and methods used in item class. This makes it very similar to the key class(with regards to methods such as getAllowableActions, newInventoryItem method etc.) except for the fact that the plans will be held inside of a room. This will be manually positioned in the application class of the game which is used to initialize the characters/terrains of the game. We decided to do it this way because we can make changes to any code in the Game Package and cannot make changes to the Engine Package.

The reason we decided to inherit from the item class is the same reason we decided to inherit the item class for key class. There are attributes and methods present which already implement a way in which player can pickup or drop an item.

Methods/Attributes:

- Inherited Attributes from the Item Class.

+ Inherited Methods from the Item Class.

**RocketEngine Class:**

Class Responsibility:

The RocketEngine which is an object of the RocketEngine class is initially given to the doctor(stored in the Doctors Inventory) at the start of the game. It is used to build the Rocket.

We create a new instance item “newRocketEngineInstance”, this way we can easily give the Rocket engine to doctor maybe

Relationships With  Other Classes:

The RocketEngine class inherits from the item class because when we thought about the implementation we realised that we needed all the attributes and methods used in item class. This makes it very similar to the RocketPlans class(with regards to methods such as getAllowableActions, newInventoryItem method etc.). Doctor Maybe must hold the RocketEngine and it is shown by a simple association relationship.

Methods/Attributes:

- Inherited Attributes from the Item Class.

+ Inherited Methods from the Item Class.

**RocketBody:**

Class Responsibility:

The RocketBody which is an object of the RocketBody class is initially given to Q(stored in Q’s Inventory) at the start of the game. It is a part used to build rocket.

Relationships With  Other Classes:

The RocketBody Class inherits  from the Item class as it has all the attributes and method that we require to implement the RocketBody Class. This is properly explained in all other classes which inherit from the Item class as well.

Methods/Attributes:

- Inherited Attributes from the Item Class.

+ Inherited Methods from the Item Class.

**RocketPad Class:**

Class Responsibility:  
This class is used to build the rocket! This is done when the player has both the Rocket Engine and the Rocket Body in his inventory upon entering the RocketPad which is a location in the map. The RocketPad is initialized at the start of the game by giving it a location.

Relationships With  Other Classes:

The RocketPad class inherits from the Ground class. We Overwrite allowableActions to create a new action of BuildingRocketAction, this is where the check for the parts takes place and where we can conclude the game.

Methods/Attributes:

- Inherited Attributes from the Ground Class.

+ Inherited Methods from the Ground Class.

**BuildingRocketAction Class:**

Class Responsibility: (Updated)

This class holds the ability to build a rocket on the current location of the RocketPad. Several parameters are passed into this class in order to effectively build a rocket(which has a parameter to know where to go to, i.e the MoonBase, and the enemy, who will be given the skill to move on the Moon only once the player gets there)

Steps:

We first go through the actors inventory and check if he has the RocketEngine and the RocketBody, and for each part found we increment the count of items and remove it from the inventory(As there can only be one instance of RocketEngine and RocketBody)

Finally, if the conditions are met, we will add a new Rocket instance.

The class will then return the appropriate message

Relationships With  Other Classes:

The only Relationship Rocket has, is with Rocket Body and Rocket Engine. It is a aggregation Relationship because both Rocket Body and Rocket Engine both combine to make the rocket. However, the Rocket Body and Rocket Engine can exists even if the Rocket didn’t exist therefore we decided to go with a aggregation relationship

Methods/Attributes:

-numberOfItems

**newPlayer: (Inherits Player)**

Class Responsibility:

This class is for the player. Since we cannot change engine code we decided to make a new class for player which inherits from the player class in the engine. We have an boolean attribute (isStun) to check if the player is stunned by Ninja and and we have a boolean attribute (remainingOxygen) for remaining oxygen and a Location attribute (RocketPadLocation) to store the location of where the player should be transported to if the oxygen is depleted.

On every playturn of the player we check if player has OXYGEN\_SKILL . If true, it means player is on the Moon and we increment the count for the Oxygen Tank. If the count is 10 we remove the tank. We check if there are remaining oxygen tanks and set the boolean remainingOxygen. If there is no remaining oxygen a SafetyTransportAction is returned. If stun isStun true a SkipTurnAction is returned. Else it returns default actions.

**Design Principles Followed**

There were quite a few design principles we thought about when we were implementing the classes. First of all, if you see our class diagrams you can see that we have used the generalised classes in the Engine to create new classes to represent characters, Items and Actions of these characters. This is to not repeat code and to follow the DRY principle.

Secondly, the use of interfaces reduces the dependencies between the classes because it uses a common method which can be implemented by a lot of different classes thereby reducing direct contact between classes. It also reduces the repetition of code.