Design Rationale

Class Diagram

We are explaining the class diagram in parts in which the work was divided in the assignment. This makes it easier to see the new classes we created at each stage and why it was created and the use for it.

Doors and Keys

The new classes that were added in this section is *Door, UnlockDoorAction and Key*. *Door* inherits *Ground* because a door is part of the ground, the door will act as an obstacle. *UnlockDoorAction* is also one of the subclasses of *Action,* A *Door* will have and instance of *UnlockDoorAction* for interaction, which means it has a one-to-one multiplicity. A *door* can only be interacted by an *UnlockDoorAction*. As for the *Key* class, it inherits the *Item* class, because a key is an item. When a player uses the *UnlockDoorAction* on a *Door*, the player will have to get the key and check if the *Key* item is in the player's inventory.

New types of enemy

As for this section, the new classes that were created is *Goon, Ninja, StunAction, InsultBehaviour*, and *MoveBehaviour*. The new enemies will inherit from *Grunt*, because *Grunt* is the only class which we can alter, since we are not supposed to touch the codes in the engine. *Goon* has the same behavior which is *FollowBehaviour* so not much to change, just need to use the *addBehaviour* function in *Grunt* to add *InsultBehaviour*. *Ninja* has a new trait as well, which is only *MoveBehaviour*. Both of these new traits, *InsultBehaviour* and *MoveBehaviour* will extend an interface called *ActionFactory*.

We made this classes because we are following the current design of the codes, which is the *FollowBehaviour*. The class *StunAction* is a subclass of *Action,* and has a dependency

on *SkipTurnAction.* This part here is for the *Ninja*, because when the player is stunned, the player would have to skip 2 game turns, which also means that *StunAction* will have 2 instances of *SkipTurnAction*.

Q

The new classes that were added for this part are *Q, RocketBody, GiveAction* and *TalkAction*. We modified Q to inherit from the class *Grunt*. This is because enemies as well as other not hostile characters are non-player characters. By inheriting *Grunt*, this prevents Q from doing unnecessary actions. Since Q is supposed to supply the player with a rocket body, Q has an instance of *RocketBody* class. Q is also supposed to give the rocket body to the player in exchange for the rocket plans, so Q would have the *giveAction* which is inherited from the *Action* class. The multiplicity of this is one to one because Q can only have one rocket body and also give one rocket body to the player. Q also talks to the player, so Q has the TalkAction which is also inherited from the *Action* class.

Miniboss: Doctor Maybe

The new classes created here are *Miniboss* and *RocketEngine*. Since Doctor Maybe is also an enemy, he inherits from *Grunt*. This ensures that the DRY (Don't Repeat Yourself) principle is adhered to since *Grunt* is also an enemy. Doctor Maybe has his own class because it inherits and overrides methods from the superclass. Doctor Maybe also has less hit points and does half the damage.

For example, the method playTurn will need to be overridden since Doctor Maybe needs to be placed in a locked room and has different actions. Doctor Maybe also holds a rocket engine and drops it when he becomes unconscious. The multiplicity of this is one to one because a miniboss can only hold one rocket engine. Since a rocket engine is an item, it inherits from the class *Item*.

Building a Rocket

The new classes here are *RocketEngine, RocketPlan, RocketBody, RocketPad and BuildPlaceRocketAction.* In order for a player to build a rocket, a player needs to have the rocket body and rocket engine. Since these are both items, they inherit from the *Item* class. The items are built with the *BuildPlaceRocketAction* class which will build and place a rocket from the items that the player has at the *Ground* rocket pad. The class *BuildPlaceRocketAction* inherits from class *Action* as it is an action. Since *BuildPlaceRocketAction* requires both items, the multiplicity is one to many.

Since the initial class diagram we were given had all the different type of actions inheriting class *Action,* all the newly added actions are also made as new classes so that there is consistency with the design. Inheriting also prevents repeated code as the subclasses can inherit methods from the superclass. Class *RocketPad* exists so that the rocket can be placed there after it has been built. The multiplicity of this is one to one as there can only be one rocket built and one rocket placed at the rocket pad. The rocket pad will also inherit from *Ground*.

Classes we added in Assignment 2

The new class we made is the *PlayerStunable,* which inherits *Player*. The reason why we did this was because *Player* is in the engine package which we are not allowed to touch, but for a *Player* to be stunned we need to assigned that state to the *Player*. Therefore, by creating this class the *Player* and overriding some of the methods, the *Player* can be stunned by the *Ninja.*

Sequence Diagram

For our interaction diagram, we used the sequence diagram to show a complex interaction which is when the player attacks a Goon. The diagram is very detailed so that all the interactions are shown intricately for our own understanding. For the player to attack a goon, the player would have to call the execute method from *AttackAction* class. To attack, the player needs a weapon, for this to happen the player needs to check their inventory if there are items that could be a weapon, if there is none, it will create an instance of *IntrinsicWeapon*, and return it back to the *AttackAction* execute method.

Once, we have the weapon that the player would use to attack. We would need to get details of the weapon, like the damage value and description of what it does. When the damage value is returned, it will hurt the Goon by deducting its hitpoints, which is the health of the Goon. If the Goon is not conscious, it will drop all the items which is in its inventory. For this to happen, the Goon becomes an item, which is an unconscious Goon and we would need to get the location of where the Goon is to replace the Goon with the unconscious Goon item.

After that, we would get the list of items that is in the inventory of the Goon and check every item in the list if the item has a *DropItemAction* instance. If it does then the item is dropped. After all the items are dropped, the Goon will be removed from the map.