Documentations & Design Rationale

Requirements

Doors and keys

Class added: Door

Roles and responsibility:

- Represents a locked door that prevent actors to pass through unless if the actor has the key to the door in the inventory, the actor can open it and go location of the door. A door can have more than one matching key and a key can only open one particular door.
- Has a field named keys of type List<Item> which will be added with the specific keys that can unlock this particular door.
- Has canActorEnter() method that will return false so that it will not give actor permission to move to the door's location in getMoveAction().
- Has blocksThrownObject() that return true since it can block thrown objects.
- Allows actor standing in front of it to performs OpenDoorAction.

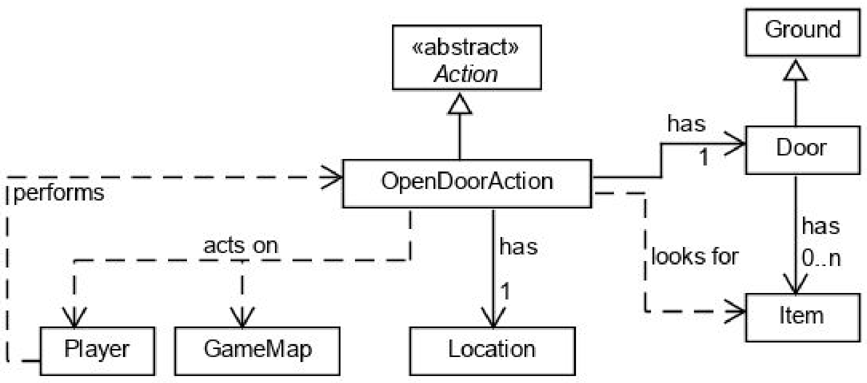
Class added: OpenDoorAction

Roles and responsibility:

- Represents an action that can be carried out by a Player that is standing in front of a door. If the Player has a matching key in its inventory, this action moves the Player to the location of the door, thus opening the door.

- In its execution, this action will search for key in the inventory of the Player that is performing the action. If key to the door is found and there is no actor blocking the door, this action moves the Player to the location of the door, otherwise the Player remains at the same location.

Interaction between classes:



- OpenDoorAction is an action to be performed by Player. It inherits the abstract class Action so that it can inherit the similar attributes and also can be processed polymorphically together with other actions in other classes.
- OpenDoorAction has a field of type Location which will be initialised with the location of the door passed into the constructor to ease the execution of this action, which is to move actor performing this action to that location. It also has a field of type Door, so that it can get the list of keys of the door when searching for key in Player’s inventory during execution.
- Door class extends Ground class because Door shares similar functionality and attributes with Ground class, so code duplication can be reduced.

- Item object involved is a key. Before opening door, OpenDoorAction will search for the key in the Player's inventory, so this action depends on the class Item. This also explains why OpenDoorAction depends on Player as the Player's inventory is required in the action.

How the required functionality is delivered:

- Key is an Item object that will be dropped from the enemies using DropItemAction. After constructing the enemy, we will add a key item to its inventory. When the enemy dies, the AttackAction class will automatically make the enemy drop all its items from its inventory. A key can only open one particular door, but there may be duplicated keys. This means that the keys dropped by two different enemies may be able to open the same matching door only.
- Door's canActorEnter() method should be overridden to return false since it is impassable unless actor performs OpenDoorAction.
- To enable the Player to pass through the door on an adjacent ground by carrying out the OpenDoorAction, we will override Door's allowableActions() so that this method returns Actions containing a OpenDoorAction instead of a empty Actions.
- OpenDoorAction's execute() method will also be overridden. This method will first get the Player's inventory using Player's getInventory() method, then iterates through it to search for the key.
    - If the key is found, create a MoveActorAction object. Then, if there is no actor at the location of the door, the location of the door will be passed into its constructor as this is the location the Player will be at after the door is opened. Then, MoveActorAction's execute() will be called to move actor through the

door. Finally, the string returned will be a message indicating that the Player successfully opened the door.

- If there is no matching key in the Player's inventory or there is an action blocking the door, a message of type string indicating that the Player fails to open the door will be returned.

New Types of Enemy: Goon

Class added: Enemy

Roles and responsibility:

- An abstract class that inherits Actor class that will be added with attributes that will be shared by all enemies so that enemies such as Grunt, Goon and Ninja can inherit from it.
- Has a field called actionFactories of type List that has elements of type ActionFactory. This is to store the behaviours of that enemy so that during its play turn, the behaviours will be process and return the action that can be performed by the enemy according to its behaviour.
- Has addBehaviour() method that adds an object of type ActionFactory to actionFactories.
- Has playTurn() method that will iterates over actionFactories and return Action that matches with that enemy's behaviour. If there is no action due to its behaviour, it will return an allowable actions of one of the actors beside it if any, otherwise it will return SkipTurnAction.
- Has getIntrinsicWeapon() to return a new IntrinsicWeapon with damage initialized to 6.

Class modified: Grunt

- Since now we have an Enemy abstract class and Grunt is an enemy to Player, Grunt class can be modified to inherit Enemy instead of Actor.

Class added: Goon

Roles and responsibility:

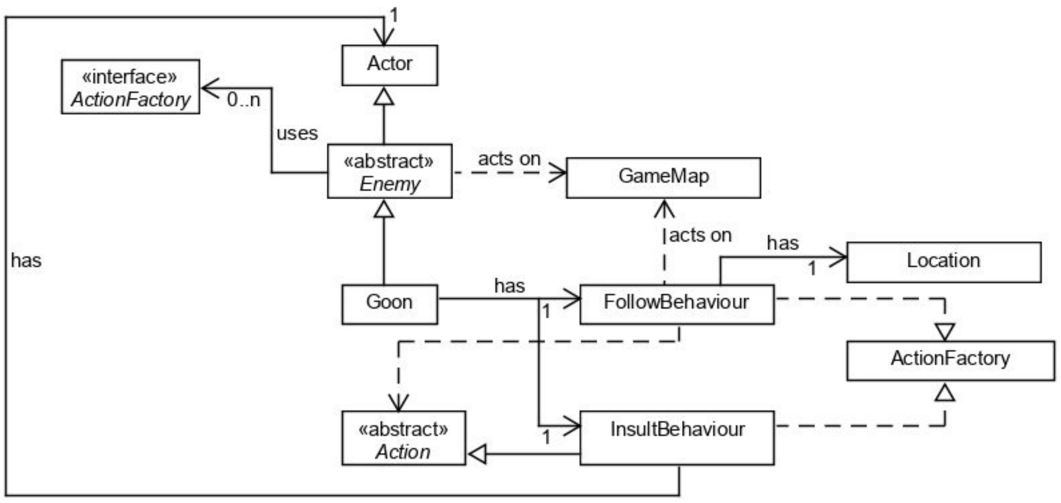
- Represents a Goon that is able to follow and insult the Player. It is also able to kick other Actor for 12 damage(twice the damage of Grunt).
- Has a field named actionFactories of type List<ActionFactory> which will be added with FollowBehaviour and InsultBehaviour.
- Has a field named inventory of type List<Item> which will be added with keys to be dropped when defeated.
- Allows actors on adjacent ground to perform AttackAction on it.
- Can perform AttackAction, SkipTurnAction and MoveActorAction(in FollowBehaviour).

Class added: InsultBehaviour

Roles and responsibility:

- Represents a behaviour to be added to Goon's actionFactories and an action that has 10% chance of insulting the Player no matter how far the Player is.
- Has a field of type List<String> that stores different insults to be shouted.
- Has a field of type Actor that stores the reference to the target that Goon will insult at.
- Has a method, getAction() that returns the action to be performed by Goon due to this behaviour, which is either null (90%) or the InsultBehaviour object itself (10%) because the execution of the action is included in this class.
- In the execution of the action that insults Player, a random insult from the list will be returned to be printed to the console.

Interaction between classes:



- We will be having more than one type of Actor that can be considered as Enemy to Player. Those actors may have behaviours so they need to have attributes such as actionFactories of type List<ActionFactory>, addBehaviour(). Therefore, instead of making them inheriting Actor and repeat code to add those attributes in all these actor classes, we decided to create an abstract class called Enemy that inherit Actor. It is abstract because it does not represent a real enemy in the game so it will never have to be instantiated. It only contains similar attributes of enemies mentioned above so that classes such as Goon and Grunt can inherit from Enemy and get all the similar attributes. Enemy's both addBehaviour() and playTurn() will contain code similar to the code given in the class Grunt.
- Since the Enemy class will have an instance variable called actionFactories of type List<ActionFactory>, we say that Enemy uses ActionFactory.
- Since InsultBehaviour class will have an instance variable of type Actor to store the reference to the target that Goon will insult at, we say that InsultBehaviour has a Actor.

- Goon is able to follow and insult the Player, so it has FollowBehaviour and InsultBehaviour. All behaviour classes will implement ActionFactory as they all must implement the getAction() method. For insultBehaviour, it also inherits Action class to achieve a simpler implementation (refer next section).
- Both Enemy and FollowBehaviour act on GameMap because Enemy's playTurn() method and FollowerBehaviour's getAction() method have parameter of type GameMap. This is because in FollowBehaviour we have to know where the target is on the map to move the actor closer to the target and in Enemy's playTurn(), the map is used to determine if there is an adjacent actor beside the enemy so that appropriate actions will be returned.
- FollowBehaviour has a field of type Location to store the reference to the previous location of the target.
- FollowBehaviour also depends on action as in its getAction(), it will return MoveActorAction if there is an exit that is closer to the target.
- Goon can also perform MoveActorAction and AttackAction but these action classes is not shown in our class diagram because this is already included in the class diagram of engine package given.

How the required functionality is delivered:

- Enemy's default damage is set to 6 in its getIntrinsicWeapon(). Grunt will not override this method, means having the default damage.
- Goon's damage can be set by overriding its getIntrinsicWeapon() to return a new IntrinsicWeapon with twice the damage of its super class. Since Grunt is having the default super class's damage, by doing so, Goon can have twice the damage of Grunt.

- It is also stated that it has follow behaviour and insult behaviour. Thus, we plan to create a InsultBehaviour class that will extend Action and implement ActionFactory. The reason is that we want InsultBehaviour to have the getAction() method, most importantly, to be processed polymorphically as an ActionFactory type in other classes. However, this behaviour is supposed to cause a string to be printed out to the console but the return type of getAction() is Action. One way of implementing this is to add a InsultAction class and return it in InsultBehaviour's getAction(). However, this is redundant because a better way is that, make InsultBehaviour to able to be referenced by both ActionFactory type and Action type using abstract class together with interface. If we do so, no extra class is required. In this case, we will override getAction() to return the InsultBehaviour object itself by using the keyword this so that execute() method can be called on itself later in World's processActorTurn() method. So, execute() method from its parent class, Action, will also be overridden to return an insult of type String to World's processActorTurn() to be printed to the console.
- In order to add randomness, Math.random will be used so that there is only 10% chance of returning the InsultBehaviour object so that it can then be executed and print an insult, otherwise return null. When null is returned to Enemy's playTurn(), InsultBehaviour action will not be passed to World's processActorTurn to be executed. In the execute() method in InsultBehaviour class, we will randomly pick and return one of the insult from InsultBehaviour's instance variable of type List<String> that stores a list of insults.

- In the Goon’s constructor, an InsultBehaviour instance and a FollowBehaviour instance will be created and added to Goon’s actionFactories using addBehavior() method.

New Types of Enemy: Ninja

Class added: Ninja

Roles and responsibility:

- Represent a Ninja that is able to stun Player for two rounds, then move one space away from Player if Player is within 5 squares of them. A Ninja will not be able to do damage to any actor.
- Has a field named actionFactories of type List<ActionFactory> which will be added with StunAction.
- Has a field named inventory of type List<Item> which will be added with keys to be dropped when defeated.
- Allows actors on adjacent ground to perform AttackAction on it.
- Can perform StunAction, SkipTurnAction and MoveActorAction(in StunAction).

Class added: StunAction

Roles and responsibility:

- Represent a behaviour to be added to Ninja's actionFactories and an action that has 50% chance of stunning Player for two rounds. If the Player is successfully stunned, the Ninja will move one space away from the Player.
- Has a field of type Actor to store the target object so this instance variable will be initialised with the Player object during the construction of StunAction.
- Has a field of type StunnedPlayer called stunnedPlayer to store the reference to a StunnedPlayer which will replace Player in the game world when player is stunned.

- Has a field named stunStatus of type int to act as a counter to keep track of how many rounds left the Player needs to be stunned before being able to perform any actions.
- Has a field called world of type World to store the reference of the World created in Application that all the actors is in so the World's player can be changed from Player to StunnedPlayer when player is stunned and changed from StunnedPlayer back to Player when player is freed from stun.
- Has a method, getAction() that returns the StunAction object itself if the Player is 5 squares away of the actor having this behaviour and there is no any Ground object blocking, or player is currently stunned.
- The execution of the action that stuns Player involves:
    1. Try to stun the Player, which ensures that Player only can perform SkipTurnAction for the following rounds. If the stun is successful, move the Ninja one space from the Player.
    2. Controlling the stunStatus counter.
    3. Free the Player from being stunned and ensure the Player has the correct value of hit points after being freed.

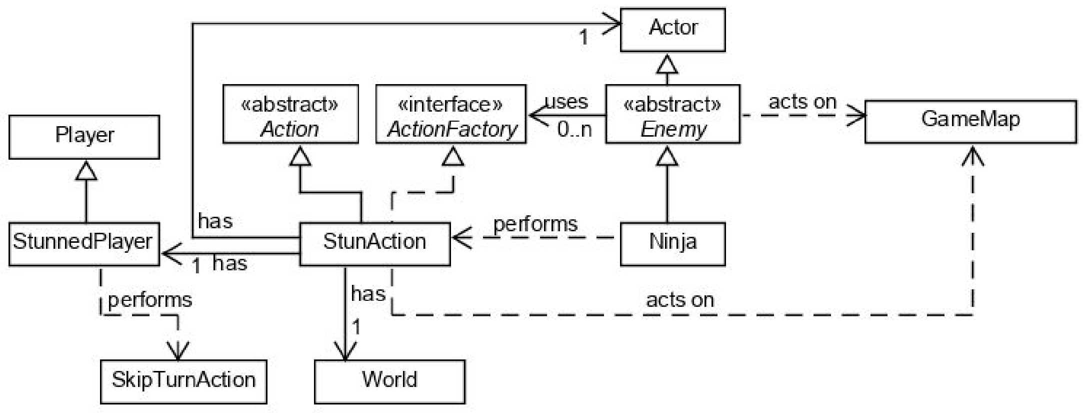
Class added: StunnedPlayer

Roles and responsibility:

- Represent a Player that is currently being stunned by Ninja who is unable to perform any action other than skipping its own turn. This player cannot be stunned anymore.
- Has the same name, display character and priority as Player so that it can be an actor in the game in place of the actual Player when Player is stunned.

- Has high hit points so that it will not be knocked out when attacked by other enemies when player is stunned and ends the game inappropriately.
- Since StunnedPlayer acts as a substitute of Player, it has a method getDeductedHP() that returns the damage caused by other enemies to the Player when they attack Player during StunnedPlayer's lifetime, which is the 2 rounds that the Player is stunned.
- Allows actors on adjacent ground to perform AttackAction on it.
- Performs SkipTurnAction, so it has a playTurn() method that always return menu with only SkipTurnAction.

Interaction between classes:



- Ninja is an enemy to Player, similarly to Grunt and Goon, we make it a subclass of Enemy. A Ninja is able to perform StunAction on Player.
- StunAction is an action that requires to be executed, so it inherits Action. It also implements ActionFactory in order to ease the delivery of the functionality (refer next section). We will give StunAction an instance variable of type Actor which is the target of this action. StunAction will also have an instance variable of type StunnedPlayer which is also a possible target of StunAction when the actual Player is

stunned. StunAction will also have a field of type World to store the reference of the World created in Application that all the actors is in so the World's player can be changed from Player to StunnedPlayer when player is stunned and changed from StunnedPlayer back to Player when player is freed from stun. In StunAction's execute() method, if Player is successfully stunned, a StunnedPlayer will replace the Player for the next two rounds in World.

- StunAction acts on GameMap because in StunAction, we need to determine the location of player/stunnedPlayer in order to add/remove it from the map whenever the player is stunned or freed from stun. Besides that, the map is also used to check if the distance between the actor performing this action is within 5 squares of the target before executing this action.
- Since a StunnedPlayer acts as a substitute of a Player when the Player is stunned, it inherits Player. A StunnedPlayer will only be able to wait during its turn, so it performs SkipTurnAction.
- Ninja can either perform StunAction or SkipTurnAction.

How the required functionality is delivered:

- Ninja can perform StunAction which inherits Action abstract class and also implement ActionFactory. The reason of implementing the interface ActionFactory is that, in World's processActorTurn(), we only get allowable actions from actors on adjacent ground but we wish to make StunAction an allowable actions when Player is within 5 squares. Moreover, if StunAction only inherit Action, it has to be added to Player's getAllowableAction(), but we want to make it an exclusive action to Ninja, which means that we have to check whether actor is a Ninja, then only we can add

StunAction to Player's getAllowableAction(). Therefore, by making StunAction implement ActionFactory, we can avoid the fuss as we can add StunAction to Ninja as a behaviour in Ninja's constructor.

- In StunAction, we will have a field "stunStatus" of type int to act as a counter to keep track of how many rounds left the Player needs to be stunned before being able to perform any actions (eg, when stunStatus == 2, this means that the Player can still be stunned for 2 rounds). We will override method getAction() so that it returns StunAction if the Player is within 5 squares or stunStatus > 0 (stunStatus > 0 indicates that the player is currently being stunned, so Ninja must perform StunAction to update stunStatus), otherwise returns SkipTurnAction. getAction() will return the StunAction object itself using the keyword "this" to enable us to call execute() on itself later in World's processActorTurn() method. The StunAction will override it's execute() method to implement the stun to the Player. To achieve this, inside the execute() method of StunAction, we will first check the stunStatus of Player. There are three possible stunStatus of the Player, which are 0, 1, and 2.

Inside StunAction's execute() method:

- If stunStatus is 0, we will first check if there is a Ground object such as Wall blocking the way. If it is not blocked, we will make use of the java.util.Random.nextBoolean() method so that there is 50% chance of the Ninja missing the subject. Otherwise, we will stun the Player. Firstly, we will move the Ninja back one space from the Player using the MoveActorAction and set stunStatus to 2 to indicate that the Player needs to be stunned for the following next two rounds. Then, we will remove the player from the map and add stunnedPlayer to the world using World's addPlayer(). By doing this, in World's stillRunning() method, it will check if the stunnedPlayer is still on the map instead of Player.

  - If stunStatus is 1, the Player should recover after this round, so we need to decrement stunStatus back to 0, then when the stunStatus is back to 0, we remove the stunnedPlayer from the map and add the actual Player back to the world, replacing stunnedPlayer. This is also where we decrease the Player's hit points by the damage stunnedPlayer has taken in place of the Player when it is stunned. We will also heal the stunnedPlayer to its maximum hitPoints so that when the player is stunned and freed again, it can return the value of hit point that the player supposed to be deducted accurately.
  - If stunStatus is 2, the Player is stunned for 1 round during Ninja's previous turn, so there is one round left so now it has to be decremented by 1.

- We decided to use a StunnedPlayer object to substitute Player during the time when Player is stunned. This is like making the actual Player that can carry out different actions to temporarily disappear from the game when it is stunned and is replaced by StunnedPlayer.
- StunnedPlayer's playTurn() will always return SkipTurnAction. We are going to give StunnedPlayer a method that returns the damage caused by enemies during its lifetime (the two rounds that Player is being stunned) so that the actual Player's hit points can be decreased accordingly after the stun effect finishes. StunnedPlayer will be given the same priority as Player during construction to ensure the flow of the game remains the same.

- This can also ensure that if there are more than one Ninjas, other Ninja cannot stun the Player that is already stunned because StunnedPlayer cannot be stunned, only Player can.

Q

Class added: Q

Roles and responsibility:

- Represents a Q that is able to wander around the map at random or talks to Player if the Player is at adjacent ground. If Player gives it rocket plans, it will give Player rocket body in return then disappear. Q is cannot be attacked by other actors.
- Allows Player on adjacent ground to perform GivePlansAction.
- Can perform TalkAction, MoveActorAction and SkipActorAction.
- Has a field named actionFactory which will be initialised with WanderBehaviour.
- Has a field of type Actor to store the reference to the target Q will talk to, which is the player.
- Has a field of type Item that stores the reference to the rocket body so that it can be passed into its allowableActions, GivePlansAction, to be processed.
- Has another field of type Item that stores the reference to the rocket plans so that it can be passed into TalkAction in playTurn() and also its allowableActions, GivePlansAction, to be processed.

Class added: WanderBehaviour

Roles and responsibility:

- Represents a behaviour of Q to be added to its actionFactories that, if player is not standing next to Q, randomly selects one of the exits, then move Q to the location of the selected exit.

- Has a method, getAction() that returns the action to be performed by Q due to this behaviour, which is either null or MoveActorAction that moves Q to a random valid adjacent ground.
- Has a field of type Actor to store the reference to player such that Q not wander around the map if it is on an adjacent ground.

Class added: TalkAction

Roles and responsibility:

- Represents an action that will always be performed by Q when the Player is on adjacent ground to prompt Player to give Q rocket plans. If the Player's inventory contains rocket plans, Q will say "Hand them over, I don't have all day!" but if Player does not have rocket plans, Q will say "I can give you something that will help, but I'm going to need the plans.".
- Has a field of type Actor to store the target object. This instance variable will be initialised with the Player object during the construction of TalkAction.
- Has another field of type Item to store the reference to rocket plans which we will search for in the target's inventory during execution.
- In its execution, this action will search for rocket plans in the inventory of the Player, then return appropriate String.

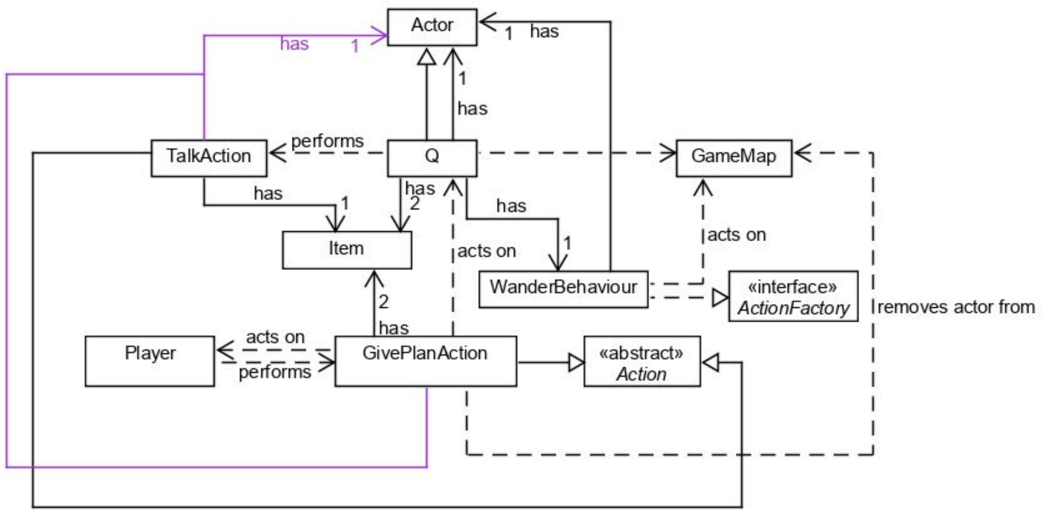
Class added: GivePlansAction

Roles and responsibility:

- Represents an action that can be performed by Player if the Player's inventory contains the rocket plans and Q is on an adjacent ground. This is an action that give Q rocket plans and get rocket body from Q.

- Has a field of type Actor to store the target, which will be initialised with a Q object.
- Has a field of type Item to store the reference to rocket plans which we will search for then remove in the actor's inventory during execution.
- Has another field of type Item to store the reference to rocket body to be added into the actor's inventory.
- In its execution, this action will search for rocket plans in the inventory of the Player, then if it exists, modify the Player's inventory to replace the rocket plans with rocket body, then remove Q from the game map.

Interaction between classes:



- Q is created as a subclass of Actor to inherit similar actor attributes such as displayChar() and getAllowedActions(). This is also to enable instance of Q to be processed polymorphically together with other actors such as Player and Enemy in World's processActorTurn() method.

- Q acts on GameMap because Q’s playTurn() method have a parameter of GameMap since in order to know what action can Q perform at a particular turn, we need to know what the map looks like.
- Q has 2 instance variable of type Item, one is to store the reference to rocket plans, another to store the reference to rocket body. The reference to rocket plans is required in order to know what it should say during TalkAction. When Q replace rocket plans with rocket body in player’s inventory, both references are also needed.
- Besides, Q is able to wander around the map, unless player is standing on an adjacent ground, it performs TalkAction, so it has WanderBehaviour. This behaviour class also implements ActionFactory so that it implement the getAction() method. WanderBehaviour has a field of type Actor to store the reference to target player such that Q will not wander if the target is next to Q. WanderBehaviour also acts on GameMap because the action returned by its getAction() depends on the game map.
- TalkAction performed by Q is a subclass of the abstract class Action. The reason of establishing the inheritance relationship is to enable TalkAction to inherit the similar attributes and also can be processed polymorphically together with other actions in other classes. We are going to give TalkAction a field of type Actor because Player is the target of this action as Player’s inventory needs to be checked. TalkAction will also have a field of type Item to store the reference to rocket plans. We will use this reference to search for rocket plans in Player’s inventory, then Q can talk appropriately.
- Player will be performing GivePlansAction, which also inherits Action to inherit similar attributes and to be processed polymorphically with other actions.

- In GivePlansAction, We are going to give this action a field of type Actor which will be the actor to give rocket plans to. This action will look for rocket plans in inventory of the actor performing this action then acts accordingly in the execute() method, so it also depends on Player because Player's inventory is required.
- GivePlansAction has a dependency relationship with GameMap because this action may remove Q from the map. It also has a dependency relationship on Q as Q is the target to give the rocketPlans in return for a rocketBody. Besides, it has 2 fields of type Item to store the reference to rocket body and reference to rocket plans. These are required to swap the rocket plans to rocket body in Player's inventory during the execution of this method.

How the required functionality is delivered:

- In order to enable the Player to give rocket plans to Q, getAllowableActions() of Q will have to be overridden so that Actions returned will contain GivePlansAction instead of AttackAction as we do not want Q to get hit by other actors. The way of executing GivePlansAction is to first get the Player's inventory using getInventory() inherited from subclass Actor, then iterates through it to search for the item, rocket plans. If rocket plans exist, rocket plans will be removed from the inventory and an item, rocket body, will be added. Before the execution of this action ends, we will make Q disappear using removeActor() from GameMap. Q's execute() method will be overridden accordingly.
- TalkAction class is added and will be a valid action to Q if Player is on an adjacent ground. The action is first get the Player's inventory, then search for the rocket plans in it. Appropriate string will be returned accordingly, which is "Hand them over, I

don’t have all day!” if Player does not have rocket plans, “I can give you something that will help, but I’m going to need the plans.” otherwise.

- Q is required to wander around the map at random, so WanderBehaviour will be added to its actionFactory in the constructor. In WanderBehaviour’s getAction(), we will calculate the distance between Q and Player, if the Player is not on the adjacent ground to Q (distance > 1), determine the location of Q, gets the valid exits at that location, if there are any, randomly selects one of the exits, then return a MoveActorAction that moves Q to the location of the selected exit. Otherwise when Player is on the adjacent ground (distance == 1) or there is no exit available, return null.
- The reason why we return null in getAction() when Player is on adjacent ground is because Q will perform TalkAction instead of wandering around in this situation. In order to achieve this, we need to override the playTurn() method in Q. In playTurn(), first, use getAction() get the action due to WanderBehaviour. If the Action returned is not null, return that action, otherwise, there may be two cases: Player is on adjacent ground or Q cannot find an exit. If Player is on adjacent ground, return TalkAction; if Q cannot find an exit, return SkipTurnAction.

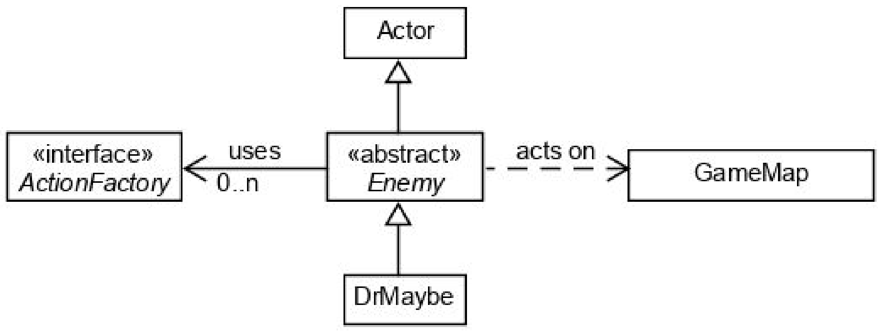
Miniboss: Doctor Maybe

Class added: DrMaybe

Roles and responsibility:

- Represent a Doctor Maybe that does not move at all. It is weak in combating as it has hit points of 25 and can only attack other actor for a damage of 3.
- Has a field named inventory of type List<Item> which will be added with a rocket engine that will be dropped when defeated.
- Allows actors on adjacent ground to perform AttackAction on it.
- Can perform AttackAction and SkipTurnAction.

Interaction between classes:



- Since Doctor Maybe will need to combat with the Player, the class also has similar attributes as Enemy class so we decided to make DrMaybe as a subclass of Enemy.
- DrMaybe can also perform AttackAction but AttackAction class is not shown in our class diagram because this is already included in the class diagram of engine package given.

How the required functionality is delivered:

- DrMaybe's damage can be set by overriding its getIntrinsicWeapon() to return a new IntrinsicWeapon with damage initialized to half its super class, Enemy's damage. Its hit points can be set during construction by calling the super constructor, then divide its hit points by 2 so it has half of Grunt's hit points. This is because Grunt's has the default hit points, which is same as that in the Enemy class.
- Doctor Maybe will only stay at a fixed location in a locked room. The locked room can be built using walls and doors in the Application class.
- Since Dr Maybe can only perform AttackAction and SkipTurnActions, we do not have to override the playTurn() from Enemy since Enemy's playTurn() is already implemented that way so DrMaybe can straight away use the playTurn() from Enemy. In that method, as we are not planning to give Doctor Maybe any behaviour, its actionFactories will be empty so it will return AttackAction if there is an actor beside it, SkipTurnAction otherwise.
- An Item, rocket engine, will be added to the inventory of Doctor Maybe during construction in the Application class so that it can drop the engine when defeated. We do not have to worry how to make Doctor Maybe to drop the engine because this is handled in the AttackAction class provided. All items in the inventory of an actor defeated will be automatically dropped.

Building a rocket

Class added: RocketPad

Roles and responsibility:

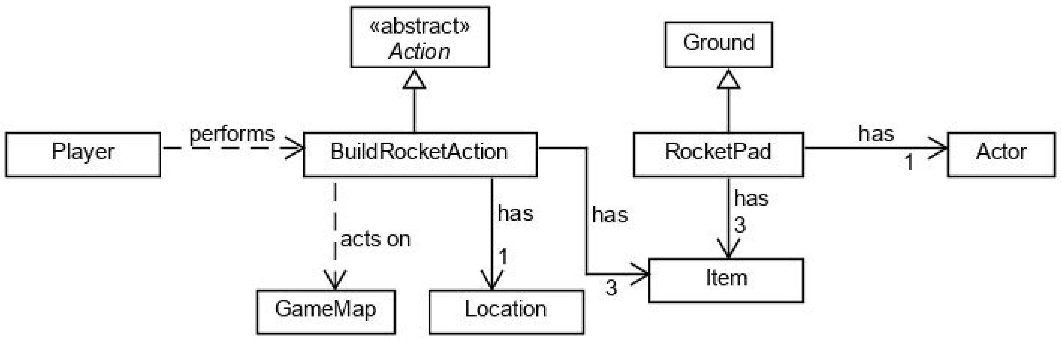
- Represents a rocket pad that allows Player to build a rocket when both rocket engine and rocket body have been dropped on it. This rocket pad will only be passable for player to prevent other actor from occupying the rocket location.
- Allows Player on adjacent ground to perform BuildRocketAction if both rocket body and rocket engine on the location of RocketPad.
- Has two fields of type Item which stores the reference to a rocket body and a rocket engine so that we can use these reference to compare with the items at rocket pad to check if the rocket pad contains rocket body and rocket engine.
- Has another field to type Item which stores the reference to the rocket to be built.
- Has a field of type Actor to store the reference to player.

Class added: BuildRocketAction

Roles and responsibility:

- Represent an action to be performed by Player to build a rocket.
- In its execution, it will return a string indicating that a rocket is built.
- Has a field of type Location that stores the reference to the location of rocket pad
- Has two fields of type Item which stores the reference to a rocket body and a rocket engine so that we can use these reference to remove rocket body and rocket engine from the rocketPad before the rocket is built.
- Has another field to type Item which stores the reference to the rocket to be built.

Interaction between classes:



- The game requires a rocket pad that enables the Player to place the rocket items on. The rocket pad mimics a ground object so RocketPad class extends Ground so it can be processed polymorphically in other classes.
- RocketPad has 3 field of type Item. Two of them are to store the reference to rocket body and rocket engine so that the references can be used to compare with the items at rocket pad to check if these two items are present at the rocket pad. Another Item is rocket to be built. This rocket will be placed on the location of rocket pad to represent that rocket has been built. It also has a field of type Actor to store the reference to player.
- BuildRocketAction is an action that is performed by the Player. It inherits the abstract class Action as it shares similar attributes with Action class and most importantly, we can process BuildRocketAction polymorphically with other actions.

- The Item objects involved are rocket body, rocket engine and rocket. RocketPad will check whether its location has both the rocket parts, then only determine whether rocket can be built, so BuildRocketAction has a field of type Location to store the location of rocket pad and it also has three fields of type Item. Two of them are used to store the reference to a rocket body and a rocket engine so that we can remove rocket body and rocket engine from the rocketPad before the building the rocket. Another Item field is to store the reference to the rocket to be built on rocket pad so that it can be placed on its location to represent a rocket has been built

How the required functionality is delivered:

- In the Application class, a room with locked door will be built when setting up the terrains to allocate space to store rocket plans. Then, rocket plans will be created and added to the map using GameMap's addItem() method right after setting up the map.
- The rocket body will be given to the Player when the Player carries out the GivePlansAction after obtaining rocket plans in a locked room (Refer: Section Q). The dropping of rocket engine is also handled in DrMaybe Class (Refer: Section MiniBoss: Doctor Maybe).
- The rocket parts should be able to be dropped by the Player on the rocket pad in order to build a rocket, so we will ensure that DropItemAction is in their allowableActions.
- When all the rocket parts have been placed on the rocket pad, the rocket pad should be able to allow the Player to carry out an action, which is building a rocket. In order to perform this action, Player has to stand on a ground adjacent to the rocket pad. We decided to create a BuildRocketAction class which will be an allowable actions of rocket pad if the location contains all three rocket parts. Thus, allowableActions() in

RocketPad will return Actions that will contain BuildRocketAction if all the rocket parts have been gathered. In BuildRocketAction's execute(), the rocket will then be added to the game map by GameMap's addItem() at the location of RocketPad.

How our implementations apply the design principles

When we are implementing the system, we apply the principle of "Minimize dependencies that cross encapsulation boundaries" by declaring all the instance variables of the classes as private, then add setters and getters for the variables to act as the interface of the class if required. This means other classes will have to interact with the class through the interface and will not be able to depend on the instance variable directly. So, when we make changes to this class, the system will not break as long as the interface is not altered.

Many of our implementations involve creating new classes that inherit existing classes. For example, BuildRocketAction, StunAction, TalkAction, GivePlansAction and InsultBehaviour inherit Action class; Enemy, StunnedPlayer inherits Player; Q inherits Actor; Door and RocketPad inherit Ground. We also modify the classes in game package by creating a new abstract class and make existing class to inherit the abstract class so that new classes that have similar attributes can also inherit that abstract class. For example, Grunt, Goon, Ninja and DrMaybe inherit that abstract class. By using inheritance, we can reduce duplicated code because for instance if we just created a new class by copying Ground class implementations and making the desired changes, we would end up with even more duplicated code, this violates the "Don't repeat yourself" principle.

The abstract class acts as core class that contains the shared attributes of subclasses so it represents a commonality of subclasses instead of a real-world concept. By making the class abstract, we can also prevent instantiating object of this class. For example, the Enemy class we created is an abstract class because we are not going to have any Enemy objects but actors such as Grunt, Goon, Ninja and DrMaybe are like enemy to the Player as they can combat with the Player.

Besides using abstract classes, we use interface for classes that do not have a inheritance relationship but have the same interface method. For example, we make StunAction, FollowBehaviour and InsultBehaviour implement ActionFactory. The interface method here is getAction(). This is because all the classes stated need to have this method but this method acts differently for each of these classes. By using interface, we can ensure that the classes implement this method.

Inheritance and interfaces enable us to write polymorphic code that can reduce code duplication so here we apply the principle of “Don’t repeat yourself”. Besides, the code will be simpler and future-proof.

Furthermore, we apply the principle of “Classes should be responsible for their own properties” as it is stated clearly in our documentations what each class is responsible of and each class is in charge of themselves as they only perform their own specific roles.