Changes in Design

***This documentation only shows the major changes that we made and the reasons behind. For the complete updated documentations , see "Documentations & Design Rationale".*

First, we introduced a new class called GameWorld that inherits World in order to implement the new requirements. Our game is now modified to use GameWorld instead of World.

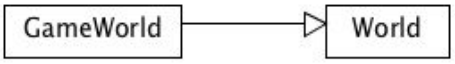

Figure 1: GameWorld inherits World

The next change is adding a instance variable of type Item to BuildRocketAction to store the reference to the rocket on earth. Before this, the rocket of type Item is built during the execution of BuildRocketAction, then we lose the reference after execution ends. Now, we have to add a MoveActorAction to rocket's allowable actions which moves the actor to the location of moon's rocket. The only place we have the reference to earth rocket is in the execution of BuildRocketAction, but we do not have the reference to moon rocket location there. Therefore, we modify our design such that the construction of earth rocket, followed by MoveActorAction to its allowable action, are all done in Application class. The reference to earth rocket will be used to initialise the newly added BuildRocketAction's field during constructor.

However, BuildRocketAction is constructed in RocketPad class, this means that we cannot directly pass the reference to earth rocket to BuildRocketAction as it has to get through RocketPad first. Therefore, RocketPad is also added with a field of type Item to store the reference to earth rocket. The new field will be initialized with the reference to earth rocket in Application class, then in allowableActions(), the reference will be passed into BuildRocketAction through its constructor. Besides, RocketPad is added with a field of type Actor to store the reference to player. Then, allowActorEnter() is overriden so that it is only passable for player. This is because we want to avoid another actor to be at the rocket location at the destination map. When this happens and player takes the rocket, the game will crash because we are trying to move player to a location that is already occupied with another actor. This modification will be able to solve this problem.

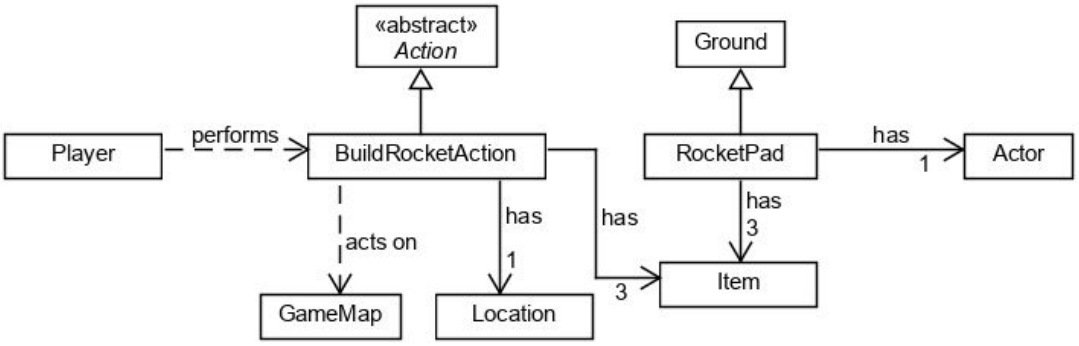

Figure 2: Updated class diagram for requirement - Building a rocket

The next changes is in Enemy's playTurn() method. Previously, we add AttackAction acting on all the adjacent actors to the enemy's list of possible actions. However, now we do not want the enemy to be able to attack all adjacent actor. For example, Q and OxygenDispenser are actors that should not be able to attacked by enemy. Thus, in Enemy's

playTurn(), AttackAction acting on adjacent actors are being replaced by adjacent actors' allowable actions which can be gotten through Actor's getAllowableActions(). The list of actions returned by getAllowableAction() in Q and OxygenDispenser will not contain AttackAction, so they will not be attacked by enemy.

Besides, in StunAction class, we removed the 'static' keyword from stunStatus. When we added another Ninja at the moon map, we realised that we do not have to declare stunStatus as static. The stunStatus, is for the Ninja to record how many rounds player still have to be stunned. It is managed by Ninja itself who performs the StunStation, we want the Ninja who stunned the player to be responsible to update stunStatus each turn. If the stunStatus is static, all Ninja/StunAction will share the same stunStatus. Consequently, in a single turn, if a Ninja stunned player and set the stunStatus to 2, the next Ninja will immediately update the stunStatus to 1. This means that player will be stunned for fewer turns. Making it non-static will make sure that every Ninja may be able to stun player correctly for 2 rounds when there is more than one Ninja in the game.

Next, another change made in StunAction class is in distance() method. This method is used to determine how many steps apart is Ninja from the target player. The distance is calculated using the coordinate of locations without taking different maps into account. Now, we have more than one map in the game and this method will not work as expected. For instance, when player is on (1, 3) on earth whereas a Ninja is on (2, 4) on moon, this method will return distance as |1 - 2| + |3 - 4|, which is 2. However, earth is supposed to be very far from moon and the Ninja should not be able to stun player under this situation. Thus, in the distance() method, before returning the distance calculated, we first check whether the two

locations are on the different map. If the maps are different, a large integer (Integer.MAX_VALUE) will be returned to indicate the huge distance between different maps.

Besides, we gave StunAction a field of type World but we modified its type to GameWorld. GameWorld is a class that we created to implement different ways of ending the game and the safety system functionality. World's addPlayer() was used to change the world's Player to StunnedPlayer and vice versa during the execution of StunAction. After introducing the safety system functionality, if the same addPlayer() is used, the safety system will be faulty because it will not be able to act on player when it is stunned. GameWorld's addPlayer() is overridden so that it not only change the world's player but it also changes the safety system's player. By modifying the type of field to GameWorld, the overridden GameWorld's addPlayer() will be used instead of the parent class's version and this will ensure safety system works correctly.

YugoMaxx has to wander around the map unless player is next to it. We realise that this is similar Q, which also wander around the map unless player is on an adjacent ground. Therefore, we decided to create a class called WanderBehaviour that implements ActionFactory. By doing so, this behaviour only has to be coded once, then can be added to both Q and YugoMaxx, thus reducing code duplication, making our code easier to be maintained. Since now we want to add WanderBehaviour to Q, we have to give Q a field called actionFactory of type ActionFactory to store it, then its playTurn() has to be modified to first use getAction() to get the actions that matches WanderBehaviour. If null is not returned by getAction(), playTurn() will return the action. If WanderBehaviour returns null, it

is either player is on an adjacent ground or there is no place for Q to wander around. Therefore, TalkAction or SkipTurnAction will be returned appropriately.

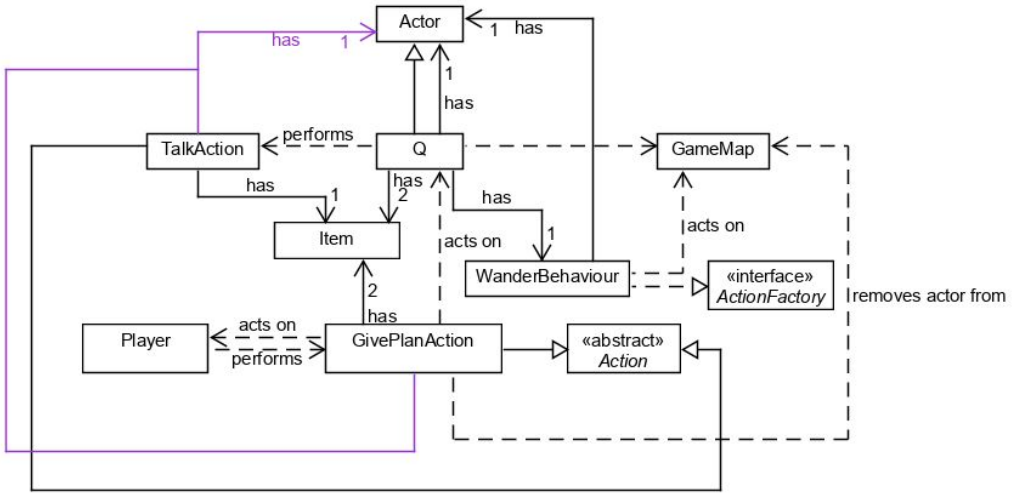

Figure 3: Updated class diagram for requirement - Q