**World class**

Problem encountered:-

1)Extra functionalities cannot be added to World class method by inheriting this class as declaring the inherited function instead of the parent function will leads to some errors as the attributes of parent class is set to private instead of protected.These attributes cannot be accessed by the method overrode in the private class.

2)run() method will process all the actors on both earth and moon Map which leads to player at Moon able to be stunned by Ninja at earth.

3)Removing player to end game results in null exception instead of displaying the “Game Over” message as some of the enemies requires player’s object to determine the player’s position.

Design changed to address the problem:-

1)The attributes of World class has been changed to protected to allow the subclass to access them which in turn allows modification of methods and extra functionalities to be added.This allows the subclass to be declared in the Application class instead of declaring the parent class.

2) run() method should keep track of the map that the player is currently on and process only those actors on the same map insteading of process all actors in both maps.

3)Ending game should not just check only the existence of player.It should remove all the actors on the maps if the player appears to be null to stop processing other actors on the map.

Advantages of the proposed changed:-

1)No conflict of behaviour of enemies on different maps on player.

2)Null pointer exception can be avoided when player is removed from the game.

Disadvantages of the proposed changed:-

1)The enemies on the other map is freezed and time is pause on the other map.

2)Time complexity of ending the game will be greater as removing all the actors on both maps requires a for loop with complexity of O(N),where removing just player alone has a complexity of O(1).

**GameMap Class**

Problem encountered:-

1)moveActor method parameter requires the location reference to move the actor.A Location object needs to be declared every single time when invoking the moveActor action.

Design changed to address the problem:-  
1)moveActor method should straight away accept the x and y coordinate instead of location reference which makes the implementation easier and clearer.

**Player Class**Problem encountered:-

1) playTurn(...) method only return showMenu(...) method, which does not allow us to add any different behaviour to the player each turn.  
2) showMenu method has a few magic number such as **“char c = hotKey.charAt(0)”** . Magic number is considered a bad design because they increase the difficulty on maintaining the code. If the magic numbers produce errors in multiple places, it will make the debugging process hard.

3) Adding an item straight to player’s inventory will result in player being able to “pick up” the item in player’s inventory.

Design changed to address the problem:-  
1) This can be solved with creating a new Player class by extending player class. Then we can override the playTurn(...) method to give this method a different behaviour. This way it allowed us to code any behaviours we want for the player, such as to deplete oxygen point each turn. This was shown in our ExtendedPlayer class.  
2)This can be solved by creating a variable Integer that is constant for all instances for the magic numbers. Example, we can refactor the 0 in **char c = hotKey.charAt(0)** to,   
   
 **private static final int INDEX\_TO\_REMOVE = 0;**

**Char c = hotKey.charAt (INDEX\_TO\_REMOVE);**

So if we want to make changes to the index, we can just change the variables value.

3)Refactor PickUpItemAction class to not depend on DropItemActionClass.  
  
**Actor Class**Problem encountered:-  
1) playTurn(...) method return random actions for the actors. This method does portrays AI moves of the characters, but we do not want the same behaviour for all characters in the game. This method restricted us to stop any character from performing a specific actions.

2)The allowable actions added to Actor action does not remains at the same index for a given direction if there is any actor or wall blocking it.To return a certain direction of movement for a particular actor,a for loop needs to be implemented to check for the index of the action in a certain direction.

Design changed to address the problem:-  
1) This class can be refactored into an Abstract class with playTurn(...) method as an abstract method. This is called polymorphism, and allow modifications of playTurn method to give arise different behaviors of that method. This will also indirectly minimize duplication between classes  
2)Allowable actions of certain direction should stay at the same index and filled the non allowable direction will dummy data such as null.

Advantages of the proposed changed:-

1)Maintaining the same index for a certain direction will maintain the time complexity of O(1) for the execution of the code for returning an actor’s action.Making use of for loop to search for the index to move to a certain direction will have O(N) time complexity for the worst case.In the case of maintaining the same index,just actions.get(i),where i is the index of the actions of movement in a particular direction is enough to return the action,no for loop is required to determine the direction.

Disadvantages of the proposed changed:-

1)Having dummy data such as null will have a greater chance of asserting NullPointerException if the programmer does not understand and handle the code well.

**DropItemAction and PickUpItemAction class**  
Problem encountered:-1) Both the execute(...) method in these 2 classes instantiate a new class of each other. This problem is called tight coupling and it also breaks the encapsulation. It is hard to modify when we want to change the behaviour such as dropping an item without possibility of being that item being picked up again, because then we are required to override both the execute(...) method of these 2 classes to achieve desired behaviours.  
  
Design changed to address the problem:-

1) This can be refactored using principle of Single - responsibility principle by making sure these 2 classes only have 1 job that do not overlap or depend on each other. “item.getAllowableActions().add(new PickUpItemAction(item));” , we can add new item behaviour in other classes. This will decrease coupling and dependencies.  
  
**Display class**  
Problem encountered:-

1) readChar() method only reads the first number of hotKey, which limits the hotKeys we can assigned to each action. We were forced to use Capital Alphabets for hotKeys after (0 to 9 had all been used

Design changed to address the problem:-  
1) readChar() method should read all the user inputs. Even though this might mean we have to implement a check input method (can be easily implemented by using throw exceptions), but this will benefits user and increase design consistency

Advantages of the proposed changed:-

1)Accepting a large range of input allows instead of just in the range of 0-9 allows more actions to be added to an actor in a single round.

Disadvantages of the proposed changed:-

1)Check input method needs to be implemented,which in turn increase the complexity of the code.

**Item Class**

Problem encountered:-  
1) There are 2 static methods, newFurniture and newInventoryItem, we do not see the need for these 2 methods to be static. We feel that static methods is not encouraged in object oriented programming because the method could be used without instantiating an object of Item class . Methods being static also breaks the encapsulation of this class.

2) Duplication of getAllowableActions in various classes such as Item and Actor class.  
  
Design changed to address the problem:-  
1) The idea of the 2 static methods is to create Items with different behaviours ( furniture or can be picked up), so we could create an ItemFactory Class that behaves like GroundFactory class to be responsible of creating item that can or cannot be picked up  
2) This method can be an abstract method of Actions class, this also means changing Actions class to an abstract class.

Advantage:  
1) Increase encapsulation of Item class and also follow Single Responsibility Principle. Item class should not be responsible of both creating new item and giving behaviours to items.

2) Decrease duplications of codes and dependencies