Recommendations for Changes to Game Engine

Problem 1: GameMap's method not linked to its instance

In GameMap, the method locationOf() returns the location of actor despite the absence of the actor on the map that this method is called on. For example, it is possible for earthMap.locationOf(player) to return moonMap when player is not on earthMap. We find this confusing because, by only looking at the interface of the class, we would expect the method to raise an exception or return null when we try to get the location of the actor that is absence on the map which the function is being called on.

Other methods that have similar problem are removeActor(), moveActor() and isAnActorAt. The instance of GameMap is not important when using these methods. This will confuse the readers and programmers because we may put focus on the instance object although the method does not act on that particular instance.

Design change proposed: Declare static actorLocations and static methods

Since the methods mentioned above do not act on one particular instance, we can say that it belongs to the class GameMap, so they can be declared as static methods. Therefore, the methods can be invoked on GameMap instead of its instance to avoid confusion.

Besides, the variable actorLocations will have to be modified to static as static methods can only access static variables. The reason why the methods can work correctly no matter they are invoked on which instance on GameMap is because all GameMap share the same actorLocations. Thus, declaring actorLocations of GameMap as static variable can make it clearer to the readers that there is only one instance of ActorLocations shared by all maps.

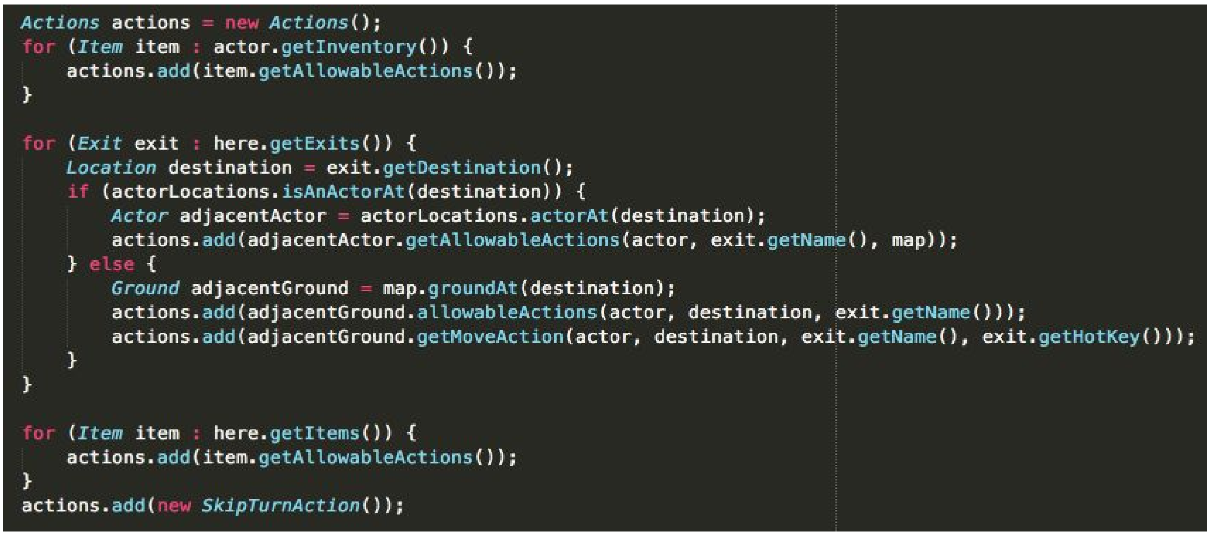
The downside of making the methods static is they cannot be overridden due to the prohibition by Java. This means that when the clients want to create subclasses of GameMap, the implementations of those static methods cannot be modified.

Problem 2: Defining Actions can be taken by an Actor in World

World's processActorTurn() checks the actor's inventory, adjacent grounds and adjacent actors for the actions allowed. All the actions are added into an Actions object which is then passed into actor's playturn() to let the actor decides which action to take. However, we think that there is no need for processActorTurn() to collect the list of allowable actions for the actor. In our game, the only actor that uses the parameter actions is Player. For other actors, they have overridden playTurn() that recreate an Action object. Similarly, in other game, not all the actions collected by processActorTurn() is allowed to be taken by the actor. Regardless, processActorTurn() will still create an Actions object for every actor's play turn. This causes additional runtime, which will be a problem when the game consists of many actors and items.

<u>Design change proposed: Delegate the task of collecting allowable actions to Actor class</u>

It is common that different game characters can perform different tasks, so instead of deciding what actions an actor can take in World, the actor can be responsible for their own play turn. This means that the parameter actions of type Actions can be removed in Actor's playTurn().

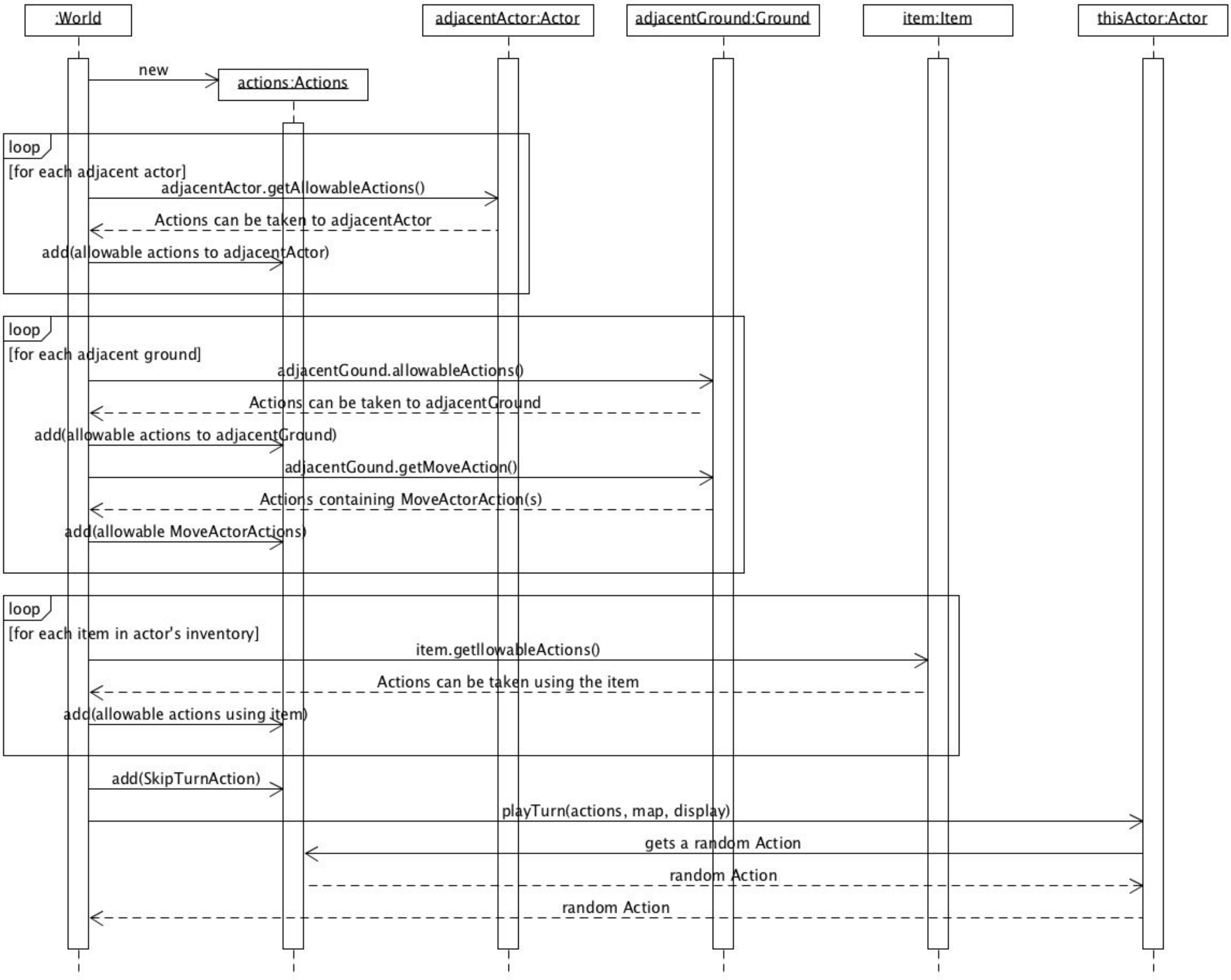


```
Actions actions = new Actions();
for (Item item : actor.getInventory()) {
    actions.add(item.getAllowableActions());
}

for (Exit exit : here.getExits()) {
    Location destination = exit.getDestination();
    if (actorLocations.isAnActorAt(destination)) {
        Actor adjacentActor = actorLocations.actorAt(destination);
        actions.add(adjacentActor.getAllowableActions(actor, exit.getName(), map));
    } else {
        Ground adjacentGround = map.groundAt(destination);
        actions.add(adjacentGround.allowableActions(actor, destination, exit.getName()));
        actions.add(adjacentGround.getMoveAction(actor, destination, exit.getName(), exit.getHotKey()));
    }
}

for (Item item : here.getItems()) {
    actions.add(item.getAllowableActions());
}
actions.add(new SkipTurnAction());
```

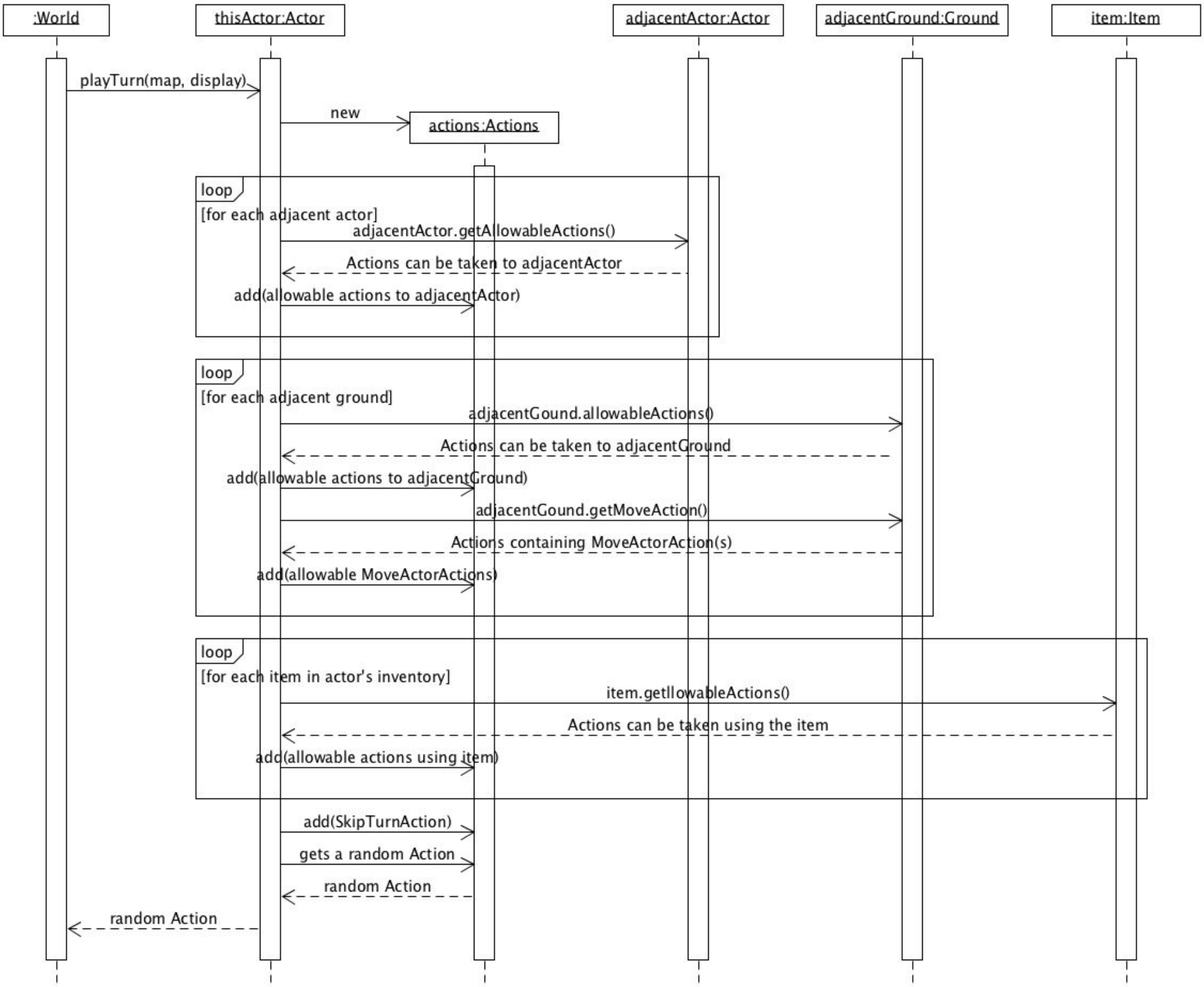
We suggest to move the code above from World's processActorTurn() to Actor's playTurn() so that the subclasses can also inherits it. If playTurn() is overridden so that the subclass actor can act differently, no unnecessary code has to be run.

Moreover, making Actor to be responsible for checking the actions allowed can also make the code to be easier to be understood. When the readers look at the Actor class, they can see how the list of actions that can be taken by the actor is collected and do not have to look for other classes where playTurn() is called in order to understand how the parameter actions is created.

Sequence of operations in getting Action to be performed by Actor in current engine code:


:World
adjacentActor:Actor
adjacentGround:Ground
item:Item
thisActor:Actor
new
actions:Actions
loop
[for each adjacent actor]
adjacentActor.getAllowableActions()
Actions can be taken to adjacentActor
add(allowable actions to adjacentActor)
loop
[for each adjacent ground]
adjacentGound.allowableActions()
Actions can be taken to adjacentGround
add(allowable actions to adjacentGround)
adjacentGound.getMoveAction()
Actions containing MoveActorAction(s)
add(allowable MoveActorActions)
loop
[for each item in actor's inventory]
item.getllowableActions()
Actions can be taken using the item
add(allowable actions using item)
add(SkipTurnAction)
playTurn(actions, map, display)
gets a random Action
random Action
random Action

Sequence of operations in proposed way of getting Action to be performed by Actor:


:World
thisActor:Actor
adjacentActor:Actor
adjacentGround:Ground
item:Item
playTurn(map, display)
new
actions:Actions
loop
[for each adjacent actor]
adjacentActor.getAllowableActions()
Actions can be taken to adjacentActor
add(allowable actions to adjacentActor)
loop
[for each adjacent ground]
adjacentGound.allowableActions()
Actions can be taken to adjacentGround
add(allowable actions to adjacentGround)
adjacentGound.getMoveAction()
Actions containing MoveActorAction(s)
add(allowable MoveActorActions)
loop
[for each item in actor's inventory]
item.getllowableActions()
Actions can be taken using the item
add(allowable actions using item)
add(SkipTurnAction)
gets a random Action
random Action
random Action

Problem 3: Item's allowable actions

The constructor of Item creates Item that can be picked up, there are also static methods that create Item that cannot be picked up and dropped, and also Item that can be dropped. If the programmers choose the constructor or method wrongly when creating an Item, the game will have a very noticeable bug. One example is when the player is added with Item that is created using the constructor, the player will be able to pick up the Item again, during the next turn, there will be two DropItemAction allowed that drops the same Item. This is quite problematic to the programmers because we always have to ensure that the Item is constructed correctly, causes the program to be less maintainable. For instance, an Item is created and added to the player's inventory, but if later we decide to modify the game by adding the Item to the game map, we have to change the code where we construct the item too. This is an unobvious dependency and the programmers are required to keep this in mind to prevent bugs.

Design change proposed: Add PickUpAction and DropItemAction in Actor's addItemToInventory() and GameMap's addItem()

The suggested change is, modify the Item class so that it only has one constructor that creates an Item with empty allowableActions. Then, in Actor's addItemToInventory(), add DropItemAction to the Item's allowableActions after adding it to the inventory whereas in GameMap's addItem(), add PickUpAction to the Item's allowableActions. Therefore, all items added to inventory will automatically can be dropped and all items added to map can be picked up. To implement furniture item that cannot be picked up or dropped, a function called addFurnitureToInventory() and addFurniture() can be coded in Actor class and GameMap class respectively. These methods will not add any allowable actions to the Item.

This can reduce the risk of programmers that use the engine code in making mistakes. The disadvantage is, now there is two methods each to add item to an actor's inventory and game map, which means that we still have to make decision on which method to use. However, the need of making correct decision here is more explicit and obvious to the programmers as compared to the current design. The reason is that, an Item can be created, then added to GameMap or Actor's inventory at different classes. In the current design, when the Item is modified to be added to GameMap instead of Actor’s inventory or vice versa, we have to find out where the Item is created and change it. In the same situation where the modified game engine is used, we only have to change the code at one place. Thus, by making this modification, the game program can also be easier to maintain.