Engine Code Recommendations

Group Name: Pikachu
Members: Jun Qing Lim, Nicholas Loong Kah Fai

Overview

This documentation describes the recommended changes over the engine design. It includes the perceived problem, the design changes and the advantages and disadvantages of the proposed solution. The recommended changes covers the design principles that the classes must adhere to while fulfilling its functionality.

1. **Actor**

- **getWeapon() method in Actor class should have greater flexiblity over having the options of selecting weapons instead of only returning the first Weapon in the inventory**

  The getWeapon() method in the Actor class iterates every item in the inventory and returns the first Weapon that it finds. When we were trying to implement the extra feature of player picking up multiple weapons and able to select and deselect the weapon to use, we could not find any way to utilize the existing methods in the Actor class. This is because the first weapon selects the first weapon in the inventory, and not allowing the second weapon picked up to be used. We say that this design is not future-proof.

  An added feature of selecting and deselecting weapon of the actor's choice would have to modify the engine code of Actor, which is bad because a class should only have one reason to change due to single-responsiblity principle. Developers would have to override the getWeapon() method in the Actor's subclasses (like GamePlayer) in order to add the feature of selecting weapon of choice. Doing so however will not allow other subclasses of Actor (like Enemy) to utilize the similar functionality.

  A solution to this problem could be adding actions like SelectWeaponAction and DeselectWeaponAction that adds and removes an enum skill of WEAPONSKILL to the WeaponItem and adding a conditional if statement that verifies if the item has this WEAPONSKILL. Adding such condition and implementation in the engine allow classes that inherits from the Actor to hold the similar functionality without the need of overriding it many times to implement the similar feature. It is said that this prevents code redundancy, because Actor's subclasses can execute same functionality with this design.

- **Actor that implements ActorBehaviours interface (or have a list of ActionFactory that represents the list of behaviours hold by the Actor)**

  In our implementation, we made both GamePlayer and GameActor to implement the ActorBehaviours interface (which consists of the ability of executing the behaviour and adding a behaviour into the list of ActionFactory). Both GamePlayer and GameActor have the a list of ActionFactory and overrides the methods in ActorBehaviours in the same way. This is considered as a code duplication because in our game, we implemented such that every actor in the game has some behaviours to be executed.

  One of the approaches to this code redundancy problem is that allowing the Actor class to implement ActorBehaviours interface (or have a list of ActionFactory and adding behaviour method in the Actor class) to allow greater flexibility because we can easily add new behaviours into an actor without further making many refactoring across the game.

2. PickUpItemAction

- **Set the PickUpItemAction constructor access modifier to public**

  PickUpItemAction's constructor is not set to public but set to its default access thus not allowing developers to extend it from outside its package. Such access restricts it to only be instantiated in its same package and it restricts further modification to the Action if more changes are required to be made to the overall design of the game. The issue that we have encountered here is that we are trying to creeate a PickUpWeaponAction that extends from PickUpItemAction. This PickUpWeaponAction adds a Weapon into the GamePlayer's list of weaponItems. However, because we cannot extend from the PickUpItemAction, so it does not adhere the open-closed design principle.

  A solution to this design problem is setting the constructor's access for PickUpItemAction to be public to allow extensions of PickUpItemAction so that more changes can be made to it without further complicating the base PickUpItemAction class. It also gives more flexibility for future modifications (let's say we want to pick up a certain item that adds into another arraylist, setting the constructor as public gives us the ability of not creating another separate entity, but extending from this and thus creates a better generalization of classes). This adheres to open-closed principle.

3. Range & Skills

- **Documenting the classes with detailed description**

  Range and Skills class are not well documented and it is tough for future developers to understand what the code does. During the implementation of throwing stun, we had trouble using the methods in the Range class as clear documentation is not provided, causing us to have to go through the code to understand the logic behind the class. For instance, the purpose of the for loop in the Range's constructor is not documented, making us to unable to have a clear idea on the purpose of iterating the count and adding into the list of Integers. This design is not flexible enough for future-proofs.

  Documenting the entire class in detailed, gives developers the ability to utilize the class without further wasting time on reading the code and building the mental logic behind it.

4. **AttackAction**

- **Adding the actor's skills into the sleeping item dropped when actor is knocked out**

  The current implementation of the AttackAction drops a default Item that would signify an Actor's unconscious body whenever the Actor is knocked out. During the process of implementing YugoMaxx, we realized that the existing AttackAction class does not have a method of differentiating the sleeping Item dropped (other than the different sleeping name). This is because the responsbility of dropping the unconscious Item is intrinsically linked to the AttackAction, we would have to modify the AttackAction to be able to add in a new feature to differentiate the items so as to allow the player to bring back the unconscious body that belongs to YugoMaxx.

  A solution to this is to add every skill that the actor has into the sleeping item. This proposed solution has a couple of advantages such as allowing the game to identify which actor belongs to the unconscious body. It also creates greater flexibility because we can add on other features without making a lot of modifications. An example would be creating a Necromancer of type Enemy that ressurrecting the enemies that were unconscious.

- **A separate high level entity Manager that checks if actors in the game are conscious and drop all items in the actor's inventory if unconscious (highly-recommended for future-proof, an alternative solution to the first approach above to avoid overcomplicating the AttackAction)**

  Instead of combining all responsiblities - attacking a target (aka hurting), checking if the target is conscious and dropping the items in the target's inventory after defeated. Adhering to design principles means each class should only responsible for one thing. A highly recommended approach would be creating a separate higher level entity Manager that manages the consciousness of all actors in the game, dropping all items in the actor's inventory when it is knocked out. This means that for every single play turn, the Manager entity is going to do "checking" on the consciousness of the actors across the game. Consequently, the AttackAction would only responsible for hurting the target. Doing so allows modifications to be done in the future without refactoring lots of classes across the engine.