________Recommendations and Evaluation of Engine

Introduction

Working with the game engine there were several instances where the codebase provided felt limiting or even difficult. With ample time to work through the assignments there was a greater understanding developed and below is a description of the sole issue faced along with the aspects that worked well for the purpose of developing our project.

Evaluation of Engine

Suggested Fix

A common issue faced by our group was that the engine package contained too many classes within one package making it considerably difficult to get accustomed to the code base in the initial stages. For example, there were several classes which were related to ‘Actions’ (i.e Action, Actions, DoNothingAction, ...). One way to solve this problem would be to create other packages that would contain all related classes and interfaces together. Based on this suggestion, an example would be to have a package named edu.monash.fit2099.engine.Actions which would contain all classes related to actions which in effect would increase clarity significantly. A drawback of this solution is that we need to be more careful with the usage of the protected modifier as only classes in the same package would get direct access to the variables/methods that have the protected modifier. However, the implications are not severe as the drawbacks can be easily dealt with by using the public modifier where the protected keyword causes any issues.

Single Responsibility Principle

As the classes utilized design principles adequately there wasn’t any need to provide any additional functionalities. For example, one good thing about the classes in the engine package was that the classes followed the Single Responsibility Principle and thus were generally cohesive. They were all small and the methods that were in the classes were all working to accomplish a single goal. This is especially evident from the Location class as everything related to location was contained within it as the name would suggest. As a result, when implementing the new functionalities, it was straightforward to add all features as we already knew which class to refer to in order to accomplish certain tasks. This characteristic applied to the majority of the classes provided in general.

Open/Closed Principle

However this was not the only instance of design principles being present within the classes as the engine package made use of many other principles such as the Open/Closed Principle. This was accomplished by making use of interfaces and abstract classes. By following this principle, it was very easy for us to add the new functionalities without having to make any changes to the classes in the engine package, i.e. we were able to extend the existing system without changing the implementation details of any of the classes.

Liskov Substitution Principle

Another design principle present within the codebase that made implementation easier was the Liskov Substitution Principle. This characteristic can be explicitly seen within the ‘Action’, ‘Actor’ and all their related classes. Using the action class as an example, by having the class ‘DropItemAction’ inherit the Action class, polymorphism is utilized and thus we were able to simply use an instance of the child class when the code was expecting an instance of the base (Action) class within certain methods. As a result the difficulty of implementing the Player class reduced where we had to add a lot of actions, but instead of using an instance for each sub class, we were able to simply replace that by making the code expect an instance of the parent (Action class in this case).

Interface Segregation Principle

In addition, the engine package followed the Interface Segregation Principle as the interfaces were small and the classes didn’t have to provide implementation for methods which they wouldn’t use hence making the system more extensible. One example where this was achieved was the ‘Weapon’ interface which only had two methods: damage and verb which then all weapon related classes implemented. This in turn works well as we ensure all items that are weapons would need to provide the implementation of the method names specified by the interface. It also makes sense as any weapon should cause damage no matter how small and should have a description to indicate to the user when the weapon is being used.

Dependency Inversion Principle

Furthermore, the classes in the engine package also followed the Dependency Inversion Principle. This was achieved by making the higher-level modules depend on the abstractions (abstract classes and interfaces) as well as the lower-level modules. By following this principle, it was possible to extend the system without modifying any of the higher level modules located in the engine package such as the ‘Item’ class which was generally not affected by the addition of things like fruit and corpses. As a result of making use of this principle, the classes in the engine classes were able to implicitly utilize the Open/Closed Principle which is the reason why we were able to extend the system without the need to change anything in the engine classes.

DRY Principle

Moreover, there was no code duplication in the classes involved in the engine package which further reinforced the DRY principle. One way this was achieved was by having abstract classes and one example can be found in the Actor class. Since the Actor class is an abstract class, we can have methods which don't have a method body and we can also provide implementation for some of the methods, especially those which will always be the same for all classes which will inherit from the Actor class. In the class, we can see that the following methods had a method body: heal, isConcious, hurt and many more. By providing an implementation for these methods, we ensure that all the child classes will have a basic implementation. If for some reason, one of the children needs to modify one of these methods, it can do so easily, but in most cases, it doesn't need to override these methods. This makes sure our code is easier to maintain since if in the future we need to change the way an Actor gets hurt, we only need to change the implementation provided in the abstract class which would translate to its children subsequently.

Lastly, privacy leaks were avoided in the engine package. One example of where it is being prevented could be found in the Location class, the method 'getExits' was making sure to return an unmodifiable list to the caller. In doing so, it makes sure that the private instance variable can't be modified outside of the Location class. This was helpful to us as we explicitly knew which variables we could not work with from the get go and would therefore reduce any potential misunderstandings.