Engine Review

The engine code provides us with the basic building blocks to create a simple game. For the most part, we did not find that we had much problems working with the engine code and in fact, thought that it provided us with lots of examples of how well designed code should look like.

In class, we were taught encapsulation which is to group similar data and functionalities together and put them together into classes and packages. This helps prevent other objects or classes from directly accessing the internal state of the object. The engine code does this very well by breaking down a game into its most basic form - actors, items, actions, etc. - where each class is responsible for their own attributes and methods. For example, GameMap doesn't keep track of how many or what items are currently on it because GameMap doesn't need to know that bit of information. GameMap is only in charge of initializing the ground, keeping track of actors, storing Locations and the like.

Another principle that we were taught is to minimize dependencies across encapsulation boundaries. We can do this by declaring methods and instance variables as private or protected as this ensures that it can only be used by the class itself or subclasses. The engine code utilizes this technique.

In terms of privacy leaks, the engine code handles it well. It makes sure that most attributes are declared as private and there are a limited number of getters and setters. Where there are getters or setters, it is coded in such a way that it doesn't leak any private information. A prime example would be the method getAllowableActions() in the Item class. It returns an Unmodifiable list of the item's allowable actions. Nobody will be able to modify the original list of actions. Another way that the engine code prevents privacy leaks, is by declaring methods such as hurt() and heal() in Actor. Instead of providing a setter for the actor's hitPoints, these methods allow other classes to decrease and increase the hitPoints when and where appropriate.

There are some examples of the code in the engine packages that adheres to the principle that states to declare things in the tightest possible scope. For example, declaring methods as private such as createMapFromStrings(GroundFactory groundFactory, List<String> lines) means that it can only be used within the class itself. This particular method served no purpose for other classes to use and so should not be accessible to other classes since they have no need for it.

There are also a lot of abstract classes in the engine package. For example, Action is an abstract class. Other classes like PickUpItemAction, DoNothingAction and DropItemAction inherits from the Action and methods like getNextAction() is not present in PickUpItemAction, DoNothingAction and DropItemAction while methods like hotkey() is present in DoNothingAction. This gives flexibility to the other classes whether they want to implement this methods or not. It makes the code more maintainable and flexible. It also follows a very important concept called Don't Repeat Youself(DRY).

Fail Fast. The engine packages follows the principle of Fail Fast which makes it easy for us to correct ourselves. For example, ActorLocation class's method, add(Actor actor, Location location) and move(Actor actor, Location newLocation). Sometimes the system will fail because we add player at an invalid location. It makes it super helpful for debugging.

The engine code follows the principle: Avoid variables with hidden meanings. There are barely any of them which makes the code easier to understand and less overwhelming.
Another good thing about the engine code is that it is filled with Java Docs. It shortens the time for us to read the code class by class, just to understand it. It quickens the process a lot and was super helpful.

Based on my perspective, most if not all of a method is located in the right class. This is very important as it can reduce unnecessary connascence. Another way the engine code reduces connascence, in this case, connascence of execution is by throwing exception. For example, in the World class, if player == null, throw exception.

The interfaces(GroundFactory,printable,Weapon,Capable) are also relatively small. It is good as doesn't force us to implement all of the methods and the interface is more accessible. The simpler an accessible interface is, the fewer opportunities for connascence there are.
The fact that the interface are so small, shows that they follow the Interface Segregation Principle where clients should not be forced to depend upon interfaces that they do not use.

The engine code makes a lot of use of multiple constructors. Multiple constructors are good because they can reduce the use of interface. For example in actions, there are two constructors in moveactoraction, and three in GameMap.

Something that improves the engine code's maintainability is that it overrides a lot. It makes it easier to add on code and also helps reduce repeated codes, DRY. This follows the Open-Closed Principle because it is easier to add extensions to it.

Overall, the engine package was a great way for us to learn about the design principles and see how they look like in practice.