Introduction

In our opinion as a pair, we thought the engine was extremely well made using a wide range of the design principles we were taught. Because of the design principles, the code was easy to extend with a minimalistic design. In this review we will talk about the design principles the engine used, the design principles we used from the engine and why this helped with our project.

Documentation

One of the design principles the engine used that immediately stood out to us was the java documentation. Java documentation was important to us because it helped us to identify what methods and classes were used for without having to read all the code behind it. The java documentation would outline the parameters it used, what it would return and what the overall function was for. Despite mainly using java documentation for identifying the use of methods within a class, it was still helpful to understand what the overall purpose of a class was using the documentation. In our case the documentation created a layer of abstraction, as instead of having to look at all the code we only had to read the documentation in most cases to understand what was happening.

Encapsulation

The engine also used encapsulation to hide the unnecessary code from us. This was useful as it made a clear interface for us when extending from it. We could easily identify how the class was meant to be used and what functions could be used. These functions were the means to control the data not only within the class, but between classes. As the attributes would only be able to be passed through accessors and changed through mutators. This also controlled the way we were able to access and mutate the classes attributes, minimising the chance of breaking the code in the engine or using it in an unintended way. Because of encapsulation, it also made it easy to navigate the prompt when typing code related to the class, as there were less options to choose from. Although this is a small feature, it helped us quickly identify the methods we wanted.

Inheritance

Inheritance was widely used not only in the engine, but in our extensions from the engine. The main advantage of inheritance was minimizing code redundancies. When we looked at the code behind the engine, it was easy to read the code because inheritance reduced the redundancy of code amongst classes, making it faster to read through less lines of code whilst also making it easy to identify where attributes and behaviours belonged. We also used inheritance to minimize our code repetition, as we didn't need to rewrite all of the code behind engine, we could inherit from it and add

onto it from our own class, this saved us time and also made it easier to fix bugs as they would be in one spot, rather than multiple spots due to code repetition. Lastly, we were able to apply the open closed design principle, where we could extend from the existing code without modifying any of it.

Polymorphism

Due to the abundance of inheritance, we were also able to polymorph existing code when a part of the functionality in a child class was different to that of the parent class. This was mainly useful for extending upon the engines code. We used polymorphism for getting allowable actions from various ground types. It wasn't until later we saw the benefit of doing this, as instead of getting the allowable actions in a conditional statement outside the class, we were able to encapsulate all the allowable actions within the class itself and simply call a function outside to retrieve this information, ultimately reducing repetition of code outside the class whilst also making retrieving the allowable actions a reusable function which adhered to the DRY principle (don't repeat yourself).

Abstract classes and methods

Abstract classes were used within the engine to make sure we could not instantiate certain classes. In this case it helped us understand which classes were templates and which classes were complete. For example, the Actor class was an abstract class so we knew that it would serve as a template for us to extend our code upon to implement a class that can be instantiated.

Abstract classes would also allow us to up cast multiple types of classes into one data type, making use of Liskov's substitution principle. This proved extremely useful with the use of abstract methods as we only needed one iteration when having to loop through items in the inventory as there were WeaponItems, FoodItems and PortableItems. Abstract methods also allowed us to perform the same operations across different but related classes, such as when we checked when MamboMarie should appear by calling shouldSpawn(), an abstract method in the abstract class we created called RespawnableActor.

Dependency injections

We noticed that some of the classes used dependency injection through setter methods to minimize dependencies. This made it easier to use the class in a different context, where we had derived classes with different/added dependencies. As the dependencies in the class were reduced, it made it easier to extend upon. We further added dependency injection in our own classes to reuse the engines code, despite our derived classes having different dependencies to the engine. Because of dependency injection

we didn’t have to change the parameters of methods, and this increased the codes extensibility and reusability.

Delegation

Lastly, the classes in engine made good use of delegation, for example in GameMap it would store an attribute of actorlocations, and if you would like to change the position of an actor or add/remove an actor from the map, it would use the attribute actorlocations to do so. This minimized method chaining for us, making the code easier to understand due to the method calls shortened length. It also gave the method a single purpose which was easy to maintain as you would not have to change many things upon calling a method.