Recommendations for Extensions

1. ItemLocations extension

Within the Location class, there is an arrayList that stores objects of class Item. However, there isn't a class responsible for iterating over Items, more specifically, iterating over the classes that extend Item. A class with this functionality would be useful since it can be used to find the closest Item to an Actor. Currently, our method to search for the closest Item to an Actor suffers from connascence of meaning/convention since it matches Items based on the Item's name. We can address this issue by creating an ItemLocations class with an ItemIterator subclass similar to the ones created for ActorLocations. The ItemLocations class would define methods to initialise, store and update the locations of these Items, which can be iterated through using the ItemIterator. The advantages of designing the class this way is that it addresses the connascence of meaning whilst conforming to the single responsibility principle. However, the ItemLocations class would need to be updated in the GameMap class like ActorLocations, which does create slightly more responsibility for the GameMap class. However, the advantage of removing connascence outweighs this cost.

2. Have the driver run each tick of run rather than world.run() ticking forever.

The reason why a new class called GameDriver was created was because there was no way to intercept each tick of World.run() that prevented a lot of new functionality to be added between turns and made it impossible to end the game early when a certain number of turns was reached.

The proposal is to have Application run each tick of run. This means that between turns, checks and sets can be done which allows more functionality to be added to the engine itself such as ending the game early after a certain number of turns. This would better match the Open/Closed Principle (OCP) where new features can be added without too much hassle.

The advantage of this is that it makes it much easier to add more complex functionality to the engine itself and how it decides to deal with each turn. Relying on world.run() to run forever and that code being unmodifiable (due to the assignment specifications) or at least being difficult to change without breaking the game is not the best design. It also means that it is possible to add a global turn counter that makes implementing functions like do something every 10 turns much easier to add. It also separates the functionality of the World class and make it just deal with each turn and another class can initiate the world and deal with the game between turns which prevents the World class from turning into a 'God' class supported SRP.

The disadvantage is that the whole game running in that single function is very easy to understand and follow the code. Separating the world.run() function between application and the world class makes it more difficult to follow and understand the code. It also does not follow the Single Responsibility Principle (SRP) as much as before as the world class ran the entire game while this proposal will split the work between world and application however this was also not beneficial for SRP as it created the 'god' class as discussed above.

Positive Opinions of the System

3. Action is only responsible for implementing the action

The Action class defines only the key methods needed to implement an action. This is done primarily through the execute and menuDescription methods, which ensures that the Action class conforms to the Single Responsibility Principle. This prevents a ‘god’ class from forming, where an Action may handle more complicated logic that is better suited in Behaviour for example. If we gave it other responsibilities such as choosing which Action to execute, then the control flow would be harder to follow which makes the program a lot less maintainable. This issue is addressed well by restricting the scope of responsibility given to the Action class.

4. The design of Actors’ methods

The Actor class is an abstract class that defines several methods that use abstract classes and interfaces. The idea behind this design principle is to ensure that classes are open for extension but closed for modification (Open-Closed Principle). For example, the Actor class can add objects belonging to the Item class to their inventory without modifying any of the code in Actor. This enables several people to work on the Actor class since they can extend the functionality of the Actor class without modifying any of the existing code within the class itself. If we designed the addItemToInventory method to interact differently with each Item, then the code within Actor would need to change constantly with each new requirement/feature defined.

5. Actions class

The Actions class represents an iterable collection of Action objects that Actors can perform. However, it contains a lot more functionality than a basic collection and serves as an abstraction layer between an Actor and the possible Actions they can perform. This is a good example of the Dependency Inversion Principle as the higher level classes (such as Actor, GameMap and GameDriver) that operate on a set of Actions do not depend on the lower level classes (Action in this case). These higher level classes can retrieve, add, sort, compare and remove Action classes within Actions without being concerned with the implementation details. Without the Actions class, the logic to perform such operations and interact with a collection of Action objects would be handled within one of these higher level classes. This violates the Single Responsibility Principle as well as these classes would take on this additional responsibility, which could potentially overload them and cause compatibility issues in the future. Furthermore, the greater degree of dependence between the classes also violates the Open-Closed Principle, as changes within Action would need to be addressed in the higher level classes.

6. Capable interface for Capabilities

The Capable Interface in tandem with the Capabilities class enables designers to easily extend classes that have capabilities without the need to modify existing code in order to integrate such changes with the rest of the application (Open-Closed Principle). The benefits of doing this have already been highlighted earlier, where designing classes with the Open-Closed Principle in mind enables greater potential for extension with easier maintainability. However, the conciseness and simplicity of the Capable Interface demonstrates another principle - the Interface Segregation Principle. This design principle states that classes should only be required to define methods that they need, and promotes breaking up interfaces so that classes may end up implementing several interfaces. This allows greater specificity within what classes implement, without forcing classes to implement a lot of unused/irrelevant methods outlined in a generic, all-purpose Interface.

7. Weapons are implemented very well.

It was very easy to create new weapons like the laser gun by just extending the WeaponItem class where the WeaponItem extended Item and implements Weapon. The WeaponItem class contained all the functionality that was required of items (such as letting the player have it in their inventory) while the Weapon interface dealt with how weapons can attack. It is a good use of the Single Responsibility Principle (SRP) as well as the Interface Segregation Principle (ISP) where only items that can be used as weapons will implement everything relating to weapons from Weapon interface. It also follows the Dependency Inversion Principle (DIP) where changes to the LaserGun class will not affect something like the AttackAction which depends on the Weapon Interface.

8. Location is an abstraction layer between GameMap and Ground.

Location and Ground are kept as separate classes which supports SRP. Location contains a lot of the functionality that is required of every single tile on the game map such as holding items, asking if an actor is at the location and dealing with x,y coordinates. Other classes rely on this functionality existing for all tiles. Having this be separate to Ground means that functionality that is unique to each type of ground can be applied to each location separately without affecting the functionality required of each location. It reduces repeating code that is required of all locations (DRY). This also applies the Dependency Inversion Principle (DIP) where the details of the ground class will not affect how classes interact with the functionality contained in the location class.

9. Printable Interface is concise and makes classes extendable

Considering that so many classes need some sort of displaychar, making a printable interface is a good way to minimise repeating code (DRY) and forces many classes like Items and Ground to have a displayChar which is required for the application to work. The Printable Interface also makes adding new functionality related to printing to the display much easier to add (OCP) and will be applied to all classes that are displayed in some way. Lastly, its conciseness demonstrates the interface segregation principle as classes can choose to implement its methods if needed without being forced to implement methods not relevant to printing display characters.