REQ3 – Design Rationale – Assignment 1 – Dean Mascitti- 33110360

Design Goals

The aim of this requirement was to implement craters and their ability to spawn huntsman spiders based off a 5% drop rate each tick. I addition the creature is expected to wonder around the map until n the vicinity of the intern then it will attack the intern with its long legs.

The goals of my design were to ensure that my program is extensible in that in implementing the craters ensuring that they could handle dropping other creatures in the future easily. Use of abstraction and restricting repeating code are also goals of this design.

Design & What OOP Principles Were Applied

A SpawnEnemyGround abstract class was made in which the crater inherited from, this class is expected to be inherited from any ground types In which can spawn an enemy including craters.

This SpawnEnemyGround class implemented the Spawnable interface mentioned in REQ2 that allows any class to spawn or drop something, grouping all of these classes together.

In terms of the huntsman spider implementation an abstract class was made called HostileCreature as it is expected that all hostile creatures would have the same functionality.

In terms of implementing huntsman spider attack intern functionality a new behaviour was added in which checks if the hostile creature is in the vicinity of an actor that isn't hostile to the intern and then prompts the creature to attack that actor.

The design OOP principles used in this requirement include: dependency inversion, single responsibility, open-closed, DRY, abstraction and extensibility.

How They Were Applied & Why in This Way

- The reason the SpawnEnemyGround abstract class was made is because you would expect that any ground type that spawns an enemy will have similar functionality, therefore by adding this functionality in this abstract class, subclasses like crater can simply inherit this and use the functionality without writing the code in its own class, therefore DRY and open-closed principles are applied as when more ground types that can spawn enemies are added no original code will need to be refactored, and the program will therefore be extensible.
- The reason the HostileCreature class was added is because it is assumed that all hostile creatures will attack players when in the vicinity, wander otherwise and have the ability to attack. Therefore, all of this functionality can be made in the abstract class and when other hostile creatures are added in the future little code will need to be refactored and subclasses will not need to repeat each other; achieving extensibility, abstraction, DRY and open-closed principles. In addition, this abstract class allows for dependency inversion in the SpawnEnemyGround class due to the fact that we can specify its attribute as a HostileCreature rather than the specific subclass, therefore reducing dependencies.
- The reason why the AttackBehaviour class was made it may be possible that other actors may have the same behaviour, therefore could be used in future applications elsewhere in

addition to for HostileCreature, promoting extensibility and single responsibility, as other classes won't be responsible for this behaviour.

Design Pros and Cons

Pros

- Extensible for future enemy spawners, hostile creatures and attacking behaved actors.
- Detailed use of abstraction
- Reduced dependencies
- Polymorphism capabilities
- Lack of repeated code

Cons

- Potentially could be abstracted broader (instead of hostile creature could be hostile actor, however this would come with other issues).
- Use of double abstractions

Alternative Designs

An alternative design I initially thought of was to not have the SpawnEnemyGround class omitted, and therefore have the crater class customizable based on drop rate and what creature it spawns, however that ignores the fact there may be ground types other than craters in which can spawn creatures, therefore I chose not to implement that design.