REQ 1 Design Rationale

Design Goal

The design goal for this assignment is to allow new types of hostile to be spawned at specific locations provided in the crater while adhering closely to relevant design principles and best practices.

Design Decision

The approach towards requirement 1 was split into 2 halves – The spawner package and new hostile types.

The spawner package includes HuntsmanSpiderSpawner, AlienBugSpawner, SuspiciousAstronautSpawner with a random attribute to determine spawn percentages that implements Spawner Interface by overriding its methods for spawning the hostiles in the crater. The spawner is responsible for providing methods to all the spawners to spawn hostiles in one of its adjacent locations. Exit class is used to locate all the exits of the crater and 1 was chosen to randomly spawn the hostiles.

The second part involves new hostile types, AlienBug and SuspiciousAstronaut which are extended to the existing Hostile class. The hostile class is then extended to Actor Class. This class defines the attributes and methods needed as an enemy in the game.

The implementation mentioned above allows code reusability and easier extension with the existence of interfaces. By separating all the class with their own requirements and responsibilities, the code can be easier to be augmented or to go through maintenance, offering better readability.

Alternative Design

One alternative approach may be achieved without the implementation of interface. The AlienBugSpawner,HuntsmanSpiderSpawner and SuspiciousAstronautSpawner are all created as individual class without any extension or implementation. This design allows all these spawner classes to be more flexibility as they are not restrictions from any extension or implementation, allowing specific methods assigned to each of the classes. Besides that, this code design promotes simplicity as it reduces the abstraction layer or implementations. However, this design is lacking of polymorphism, which makes it hard to treat all the spawner classes uniformly. Generalizing operation becomes a hassle as different spawner have their own individual methods.

Analysis of Alternative Design

The alternative design is not ideal because it violates various Design and SOLID principles:

Open/Closed Principle (OCP)

Without the usage of interface or abstract classes, the design may be less extensible because the design requires the modification of the spawner behaviors in a direct way.

Liskov Substitution Principle (LSP)

Since the subclasses have specific methods, it is going to be hard to let the subclasses substitute parent classes as modification is needed to make it possible.

Final Design

In our design, we closely adhere to the SOLID Principles alongside with the DRY Principles.

Don't Repeat Yourself (DRY)

The DRY Principle allows code reusability and modularity by making use of inheritance which allows the code to not be redundant. For example, the Hostile Class, including its subclasses, condenses all their common attributes and methods, reducing redundancy. The spawner has the same principle applied being an interface, encapsulating methods used by different spawners. With the use of this design, the code can promote code reusability and maintenance.

Single Responsibility Principle (SRP)

Each class is responsible for a specific functionality. The spawner interface serves the purpose of providing methods of spawning each individual hostile. For instance, AlienBugSpawner, SuspiciousAstronautSpawner, HuntsmanSpiderSpawner decides type of hostile spawning at the specific crater. AlienBug and SuspiciousAstronaut initialize the attributes and the actions and behaviours it contains during the game. This serves the goal of us achieving a interface with the sole task, promoting the principle.

Open/Closed Principle (OCP)

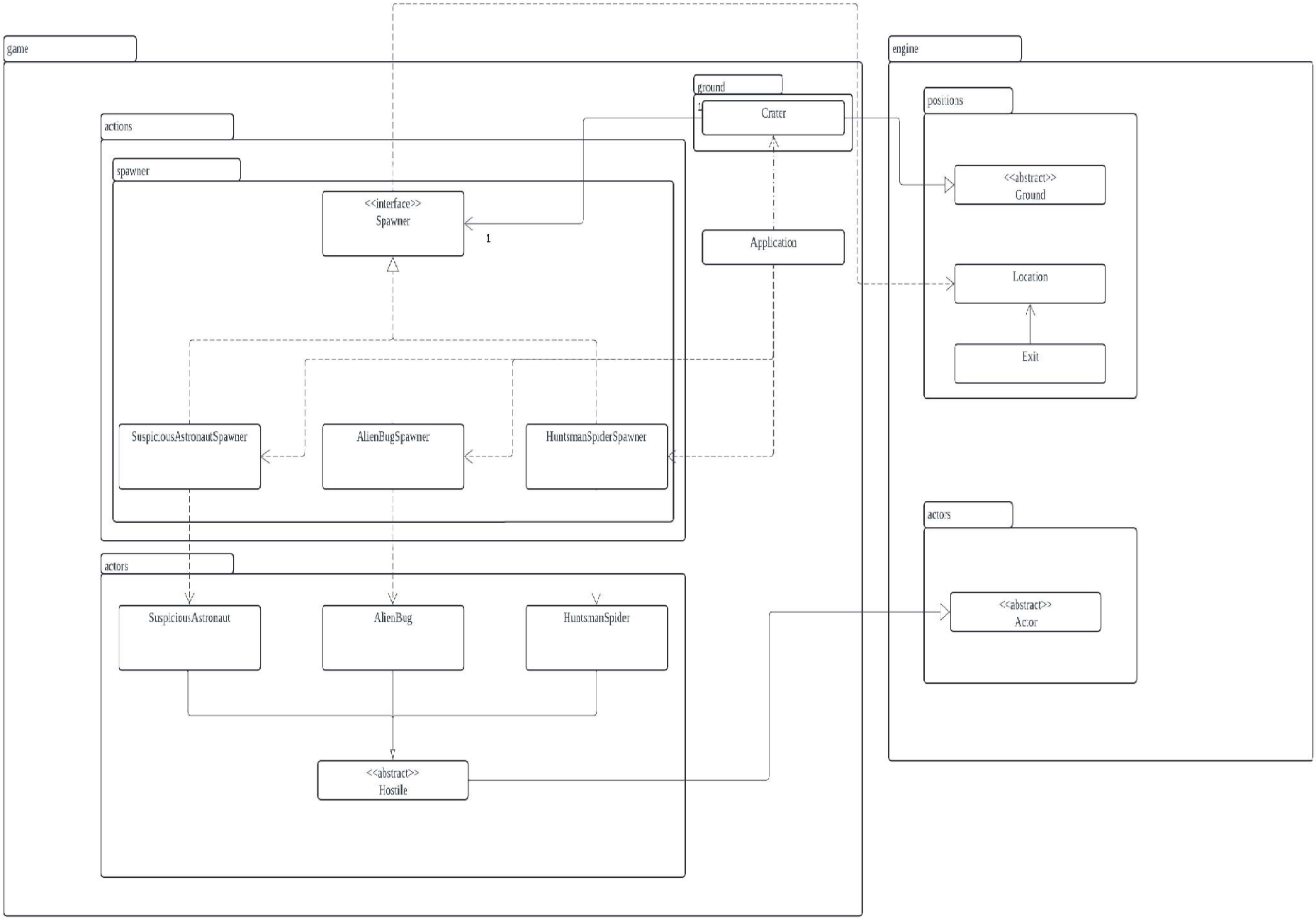
By making use of extension and interfaces, the system can be making new classes easily by extending the classes rather than modifying existing classes. Bugs can be reduced, and code stability is more likely. For example, AlienBug, SuspiciousAstronaut and HuntsmanSpider class extends to Hostile class that extends to Actor class. This design let us achieve code extensibility with no modifications of existing classes.

Liskov Substitution Principle (LSP)

Liskov Substitution Principle is applicable as the extensibility of classes created maybe served as a substitution for the parent class without affecting the correctness of the code. For example, AlienBug and SuspiciousAstronaut are instances of Hostile class therefore allowing it to be a substitute for when different part of the code requires the Hostile instance. With the code design of the Hostile class, class substitution is possible due to its high extensibility.

Pro & Cons

The advantages of interfaces allow modular and flexible implementation which supports further augmentation in the future. By following the SRP, the code is focused on 1 responsibility which allows the code to be simply maintained and customized based on requirement needs without affecting other classes, promoting independence. However, the code will get complex without proper maintenance when the number of classes extended or implemented gets higher and higher.


game
engine
ground
1
Crater
positions
actions
spawner
<<interface>>
Spawner
1
Application
<<abstract>>
Ground
Location
Exit
SuspiciousAstronautSpawner
AlienBugSpawner
HuntsmanSpiderSpawner
actors
SuspiciousAstronaut
AlienBug
HuntsmanSpider
actors
<<abstract>>
Actor
<<abstract>>
Hostile