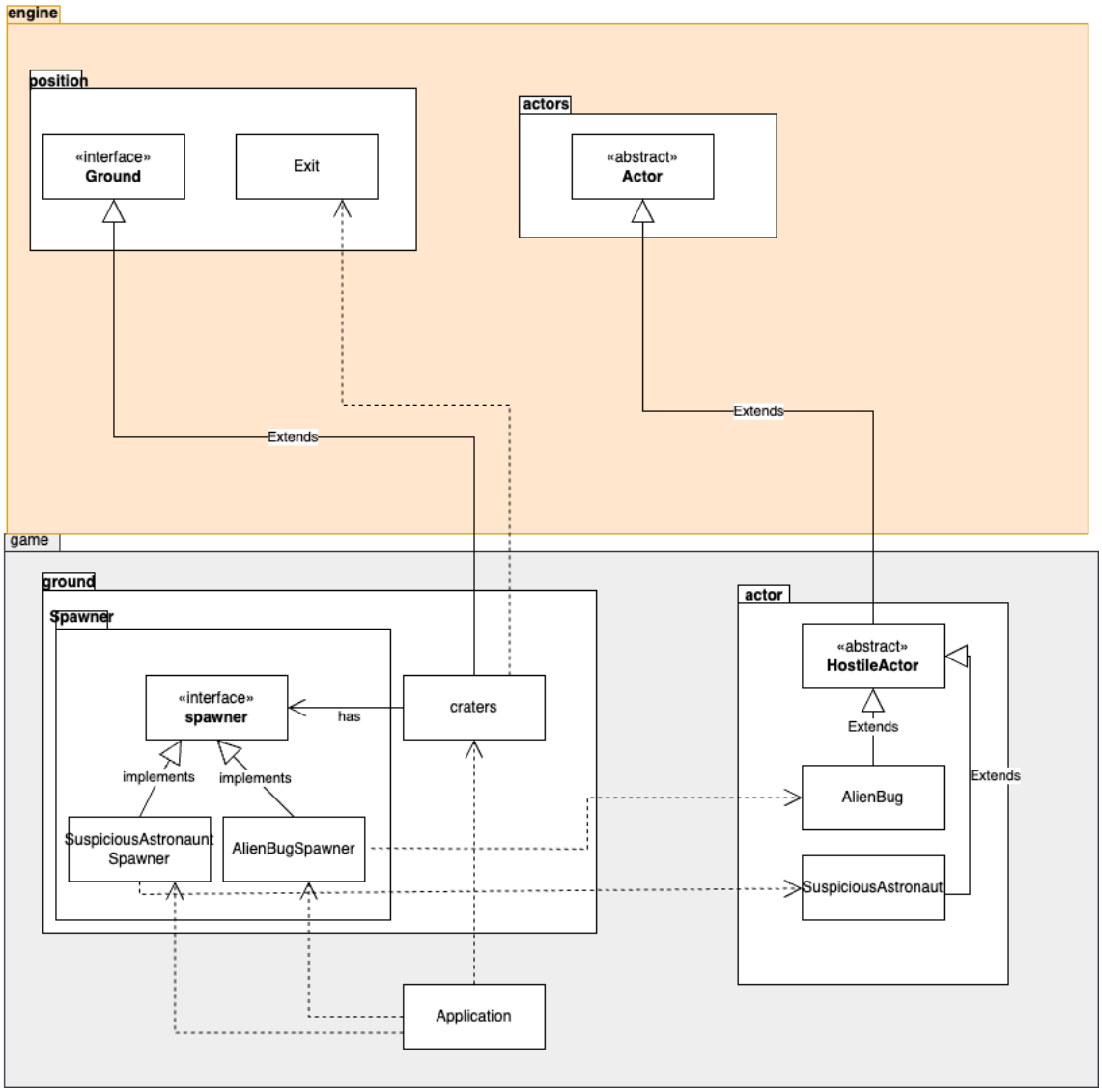
Assignment 2 Design Rationale

Requirement 1: The moon's (hostile) fauna II: The moon strikes back.

UML Diagram:



Design Goal:

The design goal for this assignment is to implement two more hostile characters namely Alien Bug (a) and Suspicious Astronaut (ඞ) in game which is kind of like the already implemented Huntsman Spider using the moon's craters (u) through the spawner interface while adhering closely to relevant design principles and best practices.

Design Decision:

In implementing both the AlienBug and SuspiciousAstronaut the decision was made to for each one to have its own spawner class which implements the spawner interface.

1. **Modularity and Encapsulation**: By creating dedicated spawner classes for each hostile character (Alien Bug and Suspicious Astronaut), It is ensured that the spawning logic for each type of creature is encapsulated within its own module. This promotes modularity by isolating the implementation details of each creature type, making the codebase easier to understand, maintain, and extend.
2. **Scalability**: Adopting a separate spawner class approach enhances the scalability of the system. As the game evolves, the introduction of additional hostile characters or modification of existing ones can be done seamlessly. Having distinct spawner classes for each creature type facilitates the smooth integration of new creatures into the game environment without disrupting the spawning mechanics of existing ones. This scalability ensures that the design remains flexible and adaptable to future changes or expansions.
3. **Maintenance:** The separation of spawning logic into dedicated spawner classes simplifies maintenance and debugging efforts. If issues arise with the spawning behavior of a particular creature type, developers can focus on debugging the corresponding spawner class without affecting other parts of the system. This granularity enhances code maintainability and facilitates troubleshooting processes.

Alternative Design:
One alternative approach could involve implementing a single, generic spawner class that dynamically determines the type of creature to spawn based on parameters passed to it. For example, the spawner class could accept a parameter indicating the type of creature (e.g., "Alien Bug" or "Suspicious Astronaut") and use conditional logic to instantiate the corresponding actor.

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

1. **Single Responsibility Principle (SRP):**
   - The alternative design violates the SRP by separating the responsibility of spawning different types of creatures within a single class. This class is responsible for both determining the type of creature to spawn and instantiating it. This violates the principle of separation of concerns, making the class less cohesive and harder to maintain.
2. **Open/Closed Principle (OCP):**

- The alternative design violates the OCP by requiring modification of the **Spawner** class whenever a new type of creature is added to the game. This class must be modified to include additional conditional logic for each new creature type, leading to increased complexity and decreased flexibility.

3. **Interface Segregation Principle (ISP)**:
   - The alternative design does not explicitly violate the ISP, but it does introduce unnecessary complexity and potential dependencies by combining multiple responsibilities within a single class. This can lead to a less cohesive and more fragile design compared to the chosen approach with separate spawner classes, which adheres to the ISP by keeping each class focused on a single responsibility.

Final Design:

In our design, we closely adhere to:

1. **Single Responsibility Principle (SRP):**
   - **Application:** Each creature type (Alien Bug and Suspicious Astronaut) has its own dedicated spawner class, adhering to the SRP by encapsulating the responsibility of spawning within a single module.
   - **Why:** Adhering to the SRP promotes modularity, maintainability, and code clarity. Each spawner class has a clear and focused responsibility, making it easier to understand, test, and modify.
   - **Pros/Cons:** This design choice enhances code organization and facilitates future modifications or extensions specific to each creature type. However, it may lead to some duplication of code across spawner classes.
2. **Interface Segregation Principle (ISP):**
   - **Application:** Each spawner class implements the spawner interface, providing a uniform interface for spawning different types of creatures. This adheres to the ISP by ensuring that clients are not forced to depend on interfaces they do not use.
   - **Why:** Adhering to the ISP promotes low coupling and flexibility, allowing clients to interact with spawner classes through a standardized interface without being affected by implementation details.
   - **Pros/Cons:** This design choice enhances flexibility and maintainability by decoupling, but it may introduce some overhead in managing multiple interfaces.
3. **Open/Closed Principle (OCP):**
   - **Application:** The design allows for the addition of new creature types without modifying existing code. Each spawner class can be extended or replaced to accommodate new requirements, adhering to the OCP.
   - **Why:** Adhering to the OCP promotes flexibility and extensibility, enabling the system to evolve over time without risking destabilization.

- **Pros/Cons:** This design choice enhances maintainability and scalability by minimizing the impact of changes to existing code. However, it requires careful design to ensure that new creature types can be seamlessly integrated without disrupting existing functionality.

Conclusion:
Overall, our chosen design provides a robust framework for implementing hostile characters such as Alien Bug (a) and Suspicious Astronaut (ඞ) in the game while adhering closely to relevant design principles and best practices by carefully considering factors such as modularity, extensibility, and interface design, we have developed a solution that promotes code clarity, maintainability, and flexibility, paving the way for future enhancements, extensions, and optimizations.