This design adheres to the DRY (Don't Repeat Yourself) principle as it avoids duplicating code by reusing the Crater class's methods and properties. The Crater class has a tick method which allows it to spawn creatures based on a certain probability. This behaviour is encapsulated within the Crater class, adhering to the Single Responsibility Principle (SRP) of SOLID principles. Each class is responsible for a single part of the software's functionality, and this responsibility is entirely encapsulated by the class. The design also adheres to the Open-Closed Principle (OCP) of SOLID principles. The Crater class is open for extension (as demonstrated by the ability to spawn different types of creatures) but closed for modification. If a new creature is added in the future, we can simply extend the Crater class without modifying its code. The design also minimizes connascence. Connascence of name is minimized by using clear, descriptive names for classes and methods. Connascence of position is avoided by using objects and methods instead of relying on the order of parameters or data.

Pros:
Code is reusable and maintainable due to adherence to DRY and SOLID principles.
The design is easily extendable for new creatures.
Encapsulation of behaviours within classes improves readability and organization of code.

Cons:
The design might be overkill for simple additions, leading to unnecessary complexity.
If not properly managed, the number of classes can grow quickly, making the codebase harder to navigate.
To extend this design, new creatures can be created and added to the Crater class. For example, a SpaceZombie could be created that spawns from a crater with a certain probability. This would involve creating a new SpaceZombie class and adding an instance of it to the Crater class.