The design for REQ 2 involves creating two new classes, `AlienBug` and `SuspiciousAstronaut`, both of which extend the `HostileActor` class. This design adheres to the DRY (Don't Repeat Yourself) principle as it avoids duplicating code by reusing the `HostileActor` class's methods and properties.

The `AlienBug` class has a `PickupBehaviour` which allows it to pick up scraps from the ground. This behaviour is encapsulated within the `AlienBug` 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 `SuspiciousAstronaut` class has an `AttackBehaviour` with a unique feature: it can instantly kill the Player. This behaviour is also encapsulated within the `SuspiciousAstronaut` class, adhering to the SRP.

The design also adheres to the Open-Closed Principle (OCP) of SOLID principles. The `HostileActor` class is open for extension (as demonstrated by `AlienBug` and `SuspiciousAstronaut` classes) but closed for modification. If a new character is added in the future, we can simply extend the `HostileActor` class without modifying its code.

The design also minimises connascence. Connascence of name is minimised 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:
1. Code is reusable and maintainable due to adherence to DRY and SOLID principles.
2. The design is easily extendable for new characters or behaviours.
3. Encapsulation of behaviours within classes improves readability and organisation of code.

Cons:
1. The design might be overkill for simple additions, leading to unnecessary complexity.
2. If not properly managed, the number of classes can grow quickly, making the codebase harder to navigate.

To extend this design, new behaviours can be created and added to existing or new classes. For example, a `FleeBehaviour` could be created for a new character that flees from the `Intern` when the `Intern` is nearby. This would involve creating a new `FleeBehaviour` class and adding an instance of it to the desired character class.