Requirement 2.

Implementations:

- **Enemy (Abstract Class):** An abstract base class which represents Enemy actors (extends Actor abstract class). This abstract class encapsulates common attributes such as hit points and a collection of behaviours applicable to all enemies. The subclasses of Enemy will have the AggressiveBehaviour by default as its priority.

- **AlienBug (Concrete Class):** A subclass of Enemy representing a type of enemy Alien Bug. This class implements specific behaviours and properties unique to the Alien Bug. The Alien Bug has 3 behaviours, when ordered according to its priority or precedence: StealingBehaviour, FollowingBehaviour (only added once a valid target enters its surroundings i.e. one of its exits), WanderBehaviour. They are also given an Ability which allows them to enter the spaceship and steal from within the spaceship.

- **Imposter (Concrete Class):** A subclass of Enemy representing a type of enemy Imposter. This class implement specific behaviours and properties unique to the Imposter. The Imposter has 2 behaviours, when ordered according to its priority or precedence: AggressiveBehaviour, WanderBehaviour. It's designed to instantly kill the player when the player enters its surroundings (one of its exits) which is handled by AggressiveBehaviour.

- **FollowBehaviour (Concrete Class):** This class implement a behaviour (implements the Behaviour interface) of an Enemy which allows them to follow another actor depending on a certain status the actor has. It works by calculating the Manhattan Distance between the actor and its target then moving to the best exit to minimize the distance between them through returning a new MoveAction. The constructor takes in an Actor target which represents the actor to follow.

- **StealingBehaviour (Concrete Class):** This class implements a behaviour (implements the Behaviour interface) of an Enemy. This behaviour allows an Enemy to pick up items from the ground they're currently standing on through returning a PickUpAction. If there's a stack of items on the ground, it'll choose the item to pick up randomly.

Modifications:

- **HuntsmanSpider:** A subclass of Enemy representing a type of enemy Huntsman Spider. Modified the HuntsmanSpider class to extend Enemy instead of Actor as there are common attributes and mechanics which are shared across all enemies. Also removed the collection of behaviours from HuntsmanSpider and relocated it to the Enemy abstract class as all Enemies will have a collection of Behaviours. This will reduce redundancy as most enemies are aggressive and will have a collection of behaviours.

Approach:

This requirement tasks us to implement 2 new types of enemy actors which are considered to be the player's enemies and to implement their behaviours accordingly. To effectively introduce the 2 new types of enemy actors and their behaviors in the game, I implemented a design where all enemies will extend an abstract class named Enemy. This class extends the Actor class and encapsulates shared attributes throughout all enemies, reducing redundancy and simplifying the process of introducing new enemy type. Now, each new enemy simply extends the Enemy class. This design choice enhances the scalability and extensibility of our code.

One of the new enemies is the Alien Bug, characterized by its unique behaviors such as picking up/stealing scraps, following other actors, and wandering around the map. To implement these behaviours, I introduced 2 new classes which are FollowBehaviour and StealingBehaviour. FollowBehaviour enables the Alien Bug to track other actors based on having a certain status when it enters its surroundings. The behaviour calculates its movements by calculating the shortest path between the actor and the target using the Manhattan Distance. This behaviour is triggered only when a potential target is within the actor's surroundings. On the other hand, StealingBehaviour allows the Alien Bug to pick up randomly chosen items within the location its currently on. A decision was also made to replace its AggressiveBehaviour with StealingBehaviour as it aligns with the Alien Bug's non-hostile nature.

Additionally, specific game mechanics, such as the ability to enter spaceships (Ability.CAN_OPEN_DOOR), are added to the Alien Bug's capabilities. The unique identifier for each Alien Bug (Feature-XXX) helps with tracking individual AlienBugs within the game.

Both FollowBehaviour and StealingBehaviour are implemented as implementations of a pre-existing Behaviour interface, which ensures consistency and modularity in behaviour implementation across different enemy types. This approach also helps with extensions in the future or modifications to enemy behaviours. By decoupling behaviours from the enemy actors and managing them through each own classes, the code becomes more flexible and easier to extend.

The Imposter is the other new enemy type. It's implemented by also extending the abstract Enemy class as previously done with other enemies. Unlike the AlienBug, the imposter is hostile. It's behaviour of the Imposter are handled through having AggressiveBehaviour and WanderBehaviour. The AggressiveBehaviour allows it to instantly kill another actor which has certain status. Meanwhile, WanderBehaviour allows the Imposter to wander around the map.

OOP Principles:

1. Single Responsibility Principle (S)

Implementation: Each class is focused on a single responsibility. For example, AlienBug handles attributes and actions specific to the Alien Bug, while StealingBehaviour is focused to only manage the logic for taking items on the ground. This separation of responsibilities makes the code easier to understand and maintain. Changes in one part of the code, like modifying the stealing behaviour won't affect parts which are unrelated to it.

2. Open/Closed Principle (O)

The code is designed to allow the introduction of new enemy types and behaviours without modifying existing code. Usage of abstract classes which in this case is the Enemy class provide a base template which new enemies can extend. Additionally, behaviours are encapsulated in classes which implement the Behaviour interface. This improves the code's extensibility and reduces the chance of getting new bugs when extending functionality. New enemy types can be added by simply adding new classes, without needing to modify existing code.

3. Liskov Substitution Principle (L)

Subclasses of Enemy, such as AlienBug and Imposter, are designed to be fully substitutable for their parent/super class. The same goes for behaviours as any class implementing Behaviour can be substituted for others without causing any issues towards the system.

4. Interface Segregation Principle (I)

Interfaces like Behaviour ensure that implementing classes only need to commit to performing tasks they actually require to do. For instance, AlienBug implements FollowBehaviour because it needs this functionality, without being affected by other unrelated behaviours. This makes it so that the classes are not forced to depend on interfaces they do not use.

5. Dependency Inversion Principle (D)

Implementation: High-level modules such as mechanics/behaviours that manage enemy actions rely on abstract classes or interfaces rather than concrete implementations. For example, Enemy depends on abstract Behaviour types, not specific behaviour classes. This helps minimize direct dependencies on concrete classes, making the code more maintainable and flexible.

6. Don't Repeat Yourself (DRY)

Implementation: The abstract Enemy class encapsulates common logic and attributes shared by all enemies, reducing code redundancy across AlienBug, Imposter, and HuntsmanSpider. This prevents code duplication and redundancy, which in turn minimizes the potential for bugs and reduces the effort needed for updates and maintenance.