Design Rationale

REQ1

Single responsibility Principle (SRP):

- Each class is responsible for a single aspect of the system. For example, WanderBehaviour is only responsible for generating a wandering action for an actor.
- HostileCreature manages behaviours and actions of hostile creatures, while AlienBug, HuntsmanSpider, and SuspiciousAstronaut represent specific types of hostile creatures.
- Spawner is responsible for managing spawning logic, and AlienBugSpawner is responsible for spawning AlienBug actors.

Pros:
Enhances code maintainability and readability by keeping classes focused on a single task. Reduces the risk of bugs and makes it easier to understand and modify the code in the future.

Open/Closed Principle (OCP):

- Classes are designed to be open for extension but closed for modification. For instance, new hostile creatures can be added to the game without modifying existing classes by creating subclasses of HostileCreature.

Pros: Promotes code reusability and extensibility. Minimises the impact of changes, as existing code remains untouched.

REQ2

Single responsibility Principle (SRP):

Each class is responsible for a single aspect of the system:

- FollowBehaviour manages the behaviour of actors following other actors.
- PickUpBehaviour handles the behaviour of actors picking up items.
- WanderBehaviour controls the behaviour of actors wandering around the game map.
- GenerateRandom provides a reusable utility for generating random integers.
- Ability enum represents different abilities that actors can possess.

Pros: Enhances code maintainability and readability by ensuring that each class has a clear and focused purpose.

Open/Closed Principle (OCP):

The classes are designed to be open for extension but closed for modification. New behaviours can be added by creating new classes that implement the Behaviour interface without modifying existing behaviour classes.
Pros: Promotes code reusability and extensibility. Minimises the risk of introducing bugs in existing code when adding new behaviours.

Polymorphism

1. **Interfaces**:
- The Behaviour interface defines a contract that specifies a method getAction(Actor actor, GameMap map). Multiple classes (FollowBehaviour, PickUpBehaviour, WanderBehaviour) implement this interface and provide their own implementations of the getAction method. This allows for polymorphic behaviour, as objects of different classes that implement Behaviour can be used interchangeably where a Behaviour is expected.

2. **Inheritance**:
- The HostileCreature class is abstract and serves as a base class for different types of hostile creatures (AlienBug, HuntsmanSpider, SuspiciousAstronaut). Each subclass inherits common behaviour from the HostileCreature class and can also override methods to provide their own implementations. For example, AlienBug overrides the unconscious method.
- Substitution of subclasses (AlienBug, HuntsmanSpider, SuspiciousAstronaut) for their base type (HostileCreature) is possible without affecting the behaviour of the program, as per the Liskov Substitution Principle (LSP), which is a key aspect of polymorphism.

<u>REQ3</u>

Single Responsibility Principle (SRP):

Each class is responsible for a single aspect of the system:
- JarPickles, PotOfGold, and Puddle represent different types of items with consumable behaviour.
- ConsumeAction handles the action of consuming a consumable item.
- ConsumableItem interface defines the contract for consumable items.

Pros: Enhances code maintainability and readability by ensuring that each class has a clear and focused purpose.

Open/Closed Principle (OCP):

The design is open for extension but closed for modification. New consumable items can be added by creating new classes that implement the ConsumableItem interface without modifying existing code.

Pros: Promotes code reusability and extensibility. Minimises the risk of introducing bugs in existing code when adding new consumable items.

Cons of the design:

Potential duplication: If multiple consumable items have similar behaviour, there might be some duplication of code, which could be mitigated by refactoring common functionality into shared components or helper classes.

Limited flexibility in action assignment: The current design ties the consume action directly to the item class, which might limit flexibility in assigning different actions to consumable items dynamically during runtime.

REQ4

Encapsulation:

- Each class encapsulates its data and behaviour within itself. For example, the ComputerItems class encapsulates the properties price and chance, controlling access to them through the class methods.

Inheritance:

- The Sword and ToiletPaper classes inherit from the ComputerItems class, demonstrating the principle of inheritance. This allows for the reuse of common attributes and behaviour defined in the parent class.

Polymorphism:

- Polymorphism is evident in the implementation of the allowableActions method in the Computer class. Each ComputerItems subclass provides its own implementation of allowableActions, allowing for different actions to be performed depending on the specific type of item present.

Single Responsibility Principle (SRP):

- Each class has a single responsibility. For example, the ConsumeAction class is responsible for executing the consumption action, while the ComputerItems class is responsible for defining properties and behaviours specific to computer items.

Open/Closed Principle (OCP):

- The design allows for extension without modification. New computer items can be added by creating subclasses of ComputerItems without needing to modify existing code, adhering to the OCP.