REQ 1

The design goal for this requirement is to create a flexible and extensible system for spawning different types of actors from craters in the game, while adhering closely to relevant design principles and best practices such as the Open/Closed Principle and Dependency Inversion Principle.

In implementing the spawning mechanism, the decision was made to use an interface Spawner that defines the methods createActor and canSpawn. This approach allows us to define different types of spawners (like AlienBugSpawner and SuspiciousAstronautSpawner) that implement this interface, each with their own logic for creating actors and determining if they can spawn. This design decision was driven by the need for flexibility (different types of actors can be spawned), extensibility (new types of spawners can be easily added), and maintainability (changes to one type of spawner do not affect others).

The Crater class represents a crater in the game. It uses a Spawner interface to manage the spawning of creatures. The Crater class doesn't create the creatures itself, but instead delegates this responsibility to the Spawner object it contains. The AlienBugSpawner and SuspiciousAstronautSpawner classes implement the Spawner interface to spawn their respective actors. This design allows different craters to spawn different actors, simply by using different Spawner implementations.

The use of the Spawner interface and its implementing classes is an example of the Factory Method design pattern. This pattern provides an interface for creating objects, but allows subclasses to alter the type of objects that will be created. It's a good fit for the Crater class, as the type of actor that a crater spawns can vary, and this variation is encapsulated by the Spawner implementations.

Single Responsibility Principle (SRP): Each class has a single responsibility. AlienBugSpawner and SuspiciousAstronautSpawner are responsible for spawning AlienBug and SuspiciousAstronaut actors respectively. This separation of concerns makes the code easier to understand, test, and maintain, as changes to the spawning logic of one actor type won't affect the other.

Open-Closed Principle (OCP): The Spawner interface is open for extension but closed for modification. New types of spawners can be added by implementing the Spawner interface, without needing to modify existing code. This promotes code flexibility and reduces the risk of introducing bugs.

Liskov Substitution Principle (LSP): Any class that implements the Spawner interface can be used interchangeably. This allows us to use different types of spawners in the Crater class without altering the program, providing flexibility in which actors are spawned.

Interface Segregation Principle (ISP): The Spawner interface is client-specific. It only includes the methods needed to spawn actors, ensuring that classes implementing the interface don't depend on methods they don't use, which simplifies the implementation and reduces the impact of changes.

Dependency Inversion Principle (DIP): The Crater class depends on the Spawner abstraction, not on any concrete Spawner implementations. This reduces the coupling between classes, making the code more flexible and easier to change.

However, there are a few drawbacks to this design. For new developers or those unfamiliar with the factory pattern, it might take more time to understand the code, as they need to find the factory method to understand how objects are being created. Additionally, for each type of Actor that can be spawned, a new Spawner implementation needs to be created. This can lead to a large number of classes, which can make the codebase harder to manage and understand.

An alternative design to the factory pattern could be to directly instantiate the SuspiciousAstronaut and AlienBug actors within the Crater class itself. Here's a brief explanation of how this might work:

Instead of using a Spawner to create SuspiciousAstronaut and AlienBug actors, the Crater class could include a method that directly creates these actors based on a probability. This would eliminate the need for the Spawner interface and its implementing classes. However, this alternative design would violate several key software design principles:

Single Responsibility Principle (SRP): The Crater class would now be responsible for both representing a crater and spawning actors. This mixes concerns and makes the class harder to understand, test, and maintain.

Open-Closed Principle (OCP): If we wanted to add a new type of actor, we would have to modify the Crater class, which goes against the principle of being open for extension but closed for modification.

Dependency Inversion Principle (DIP): The Crater class would depend directly on the concrete SuspiciousAstronaut and AlienBug classes, rather than on the abstract Spawner interface. This increases the coupling between classes and makes the code less flexible and harder to change.

Connascence of Type (CoT): The Crater class directly instantiates SuspiciousAstronaut and AlienBug actors. This introduces a high level of CoT as the Crater class needs to know the specific types of these actors to create them. This design is less flexible and more tightly coupled, as any changes to these actor classes or the addition of new actor types would require modifications to the Crater class.

By contrast, the current design using the factory pattern adheres to these principles, making the code more flexible, maintainable, and easy to understand. It allows for easy addition of new types of actors without modifying existing classes, and it separates concerns by having different classes responsible for representing craters and spawning actors.

REQ2

The FollowBehaviour and PickUpBehaviour classes are introduced, to represent the behaviours of the AlienBug enemy. The WanderBehaviour and AttackBehaviour classes are used, which were

introduced in the previous assignment, to represent the behaviours in AlienBug and SuspiciousAstronaut. The RandomUtil class also introduced in the last assignment, is used to pick a random item in the PickUpBehaviour.

The WanderBehaviour and AttackBehaviour classes have similar functionality for the behaviour of attacking and wandering in the enemies, so it is best to reuse these classes to avoid repetition (DRY). The RandomUtil class also has similar functionality in selecting a random item, which is also reused to avoid introducing repeated code.

Instead of having a huge Behaviour class to handle the behaviours of enemies, the WanderBehaviour, AttackBehaviour, PickUpBehaviour and FollowBehaviour are separated so they each handle their own responsibilities (Single Responsibility Principle).

PickUpBehaviour and FollowBehaviour are abstracted by implementing the Behaviour interface, so the Enemy class does not have to be modified to implement these new behaviours (Open-Close Principle).

Furthermore, by abstracting PickupBehaviour and FollowBehaviour with the Behaviour interface, this will allow the Enemy classes, such as AlienBug and SuspiciousAstronaut, to depend on abstractions instead of concrete Behaviour classes (Dependency Inversion Principle). This makes it easier to extend the behaviours since the Enemy classes does not have to depend on another concrete behaviour class.

The advantage of the design is the ease of extensibility for behaviours due to the abstraction layer allowing them to be added without having to modify classes that have a dependency on them. Another advantage would be reducing the number of dependencies (ReD) through the Enemy abstract classes to the behaviours by using a Behaviour interface, which will lower coupling and result in less modifications. It also makes debugging easier since the Behaviour interface contains empty methods so bugs can only lie in the PickupBehaviour and FollowBehaviour concrete classes that implement the Behaviour interface.

The disadvantage is implemented behaviours may have similar methods and attributes so there may be repeated code which will violate the DRY principle. The alternative approach is to use an abstract Behaviour class, but there may be behaviours that are vastly different from each other and do not share similarities.

REQ3

The JarOfPickles and PotOfGold classes are introduced to represent the respective scrap items that implements the Consumable interface so the scrap can be consumed by the Player with ConsumeAction. The Puddle class has been extended to also implement the Consumable interface so the water in the puddle can consumed by the Player with ConsumeAction.

Consume Action inherits from the abstract Action class. The actions share common methods, so it is best to abstract their identities to prevent repetition (DRY). Both types of scrap which are

JarOfPickles and PotOfGold inherit from the abstract Item class in the game engine, which also prevents repetition.

In addition, instead of having a god Scrap class where both JarOfPickles and PotOfGold are handled in the same Scrap class, they are split into separate individual classes. This will allow JarOfPickles and PotOfGold to be responsible for a single part of the functionality, such as their actions, rather than having a Scrap class handle everything (Single Responsibility Principle).

JarOfPickles, PotOfGold and Puddle implements the Consumable interface. This will reduce multiple associations from consumeAction to concrete consumable classes. When adding a new consumable, we do not need to modify the ConsumeAction class to extend it (Open-Close Principle).

The scraps also preserve the meaning of the methods from Item, since it does not change any of its methods, so it can be replaced with the Item without interfering with the behaviour of the program (Liskov Substitution Principle). The same goes for Puddle preserving the methods of Ground. This will ensure that polymorphism is robust since the scraps and Puddle will still be able to work similarly to an Item and Ground class.

Furthermore, JarOfPickles, PotOfGold and Puddle are abstracted by implementing the Consumable interface. This will allow ConsumeAction to depend on abstractions rather than concrete Consumable classes (Dependency Inversion Principle). This will make it simple to add an additional Consumable since ConsumeAction does not need to depend on another concrete Consumable class, and the Consumable can inherit from different parent classes.

The advantage of this design is the ease of extensibility for Consumables, since it will work for any class that implements it, without modifying the ConsumeAction class. Even though the parent classes for the scraps and Puddle are different, they can implement the same interface and be used in ConsumeAction. If consumable was an abstract class, the scraps and Puddle classes cannot extend from it since they already extend from Item and Ground abstract class respectively. Another advantage is reducing the places bugs can be since the Consumable interface has empty methods so the bug can only be in the JarOfPickles, PotOfGold or Puddle class.

However, the disadvantage is that the consume method from Consumable may be very similar for most consumables. This may lead to repetition and violating the DRY principle. The alternative approach is to use an abstract Consumable class so concrete consumable classes can inherit it, but this will be a problem for Consumable that already inherit from a parent class, which are the scraps and Puddle introduced in this requirement. The game will have to handle Consumables and Item or Ground separately, and Consumables may require duplicate code to have similar functionality, which would also break the DRY principle.

REQ4

The design goal for this requirement is to create a flexible and extensible system for purchasing items from a computer terminal. The system should be able to handle a variety of different items, each with their own unique properties and behaviours.

In implementing the purchasing system, the decision was made to use an interface, Purchasable, that defines items that can be bought, such as the DragonSlayerSword, EnergyDrink, and ToiletPaperRoll. Each of these classes implement a purchase method from the interface that handles the purchase logic of each item allowing for unique behaviours. The ComputerTerminal class represents a terminal where actors can purchase items. It maintains a list of Purchasable items to keep track of the items that can be bought from the computer terminal. The PurchaseAction class defines the action of purchasing an item. The design allows for flexibility and extendibility, as new types of purchasable items can be added by simply implementing the Purchasable interface and being passed into ComputerTerminal which creates a PurchaseAction for each item.

Single Responsibility Principle (SRP): Each class in the code has a single responsibility. For instance, the ComputerTerminal class is responsible for managing purchasable items and providing actions related to them. The PurchaseAction class is responsible for executing the purchase action. This is good because it makes the code easier to maintain and understand. Changes in one part of the system are less likely to affect other parts.

Open-Closed Principle (OCP): The code is open for extension but closed for modification. For instance, the Purchasable interface allows new types of purchasable items to be added without modifying the existing code. Moreover, new purchasable items can also be added to the computer terminal without modifying the existing code by just passing in a list with the new items into the computer terminal's constructor. This is good because it allows the system to be extended without risking the introduction of new bugs in existing code.

Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types. For instance, EnergyDrink, ToiletPaperRoll, and DragonSlayerSword are all Purchasable and can be used interchangeably. This is good because it allows the system to be flexible and to use different types of objects interchangeably.

Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. For instance, the Purchasable interface is small and focused, and classes that implement it are not forced to provide any other methods. This is good because it makes the system more flexible and easier to change.

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. For instance, the ComputerTerminal class depends on the Purchasable interface, not on concrete classes. This is good because it reduces the coupling between classes, making the system easier to change and maintain.

However, for each type of Item that can be purchased, a new Purchasable implementation needs to be created. This can lead to a large number of classes, which can make the codebase harder to manage and understand.

An alternative design to the current implementation could be to directly handle the purchasing logic within the Terminal class itself. Here's a brief explanation of how this might work:

Instead of using the Purchasable interface to handle the purchasing of items, the Terminal class could include a method that directly handles the purchasing logic. This would eliminate the need for the Purchasable interface and its implementing classes allowing for less classes and a simpler codebase. However, this alternative design would violate several key software design principles and introduce a design smell:

Single Responsibility Principle (SRP): The Terminal class would now be responsible for both representing a terminal and handling the purchasing logic. This mixes concerns and makes the class harder to understand, test, and maintain.

Dependency Inversion Principle (DIP): The Terminal class would depend directly on the concrete subclasses of Item, rather than on the abstract Purchasable interface. This increases the coupling between classes and makes the code less flexible and harder to change.

Design Smell - Shotgun Surgery: The Terminal class becomes a target for frequent changes as it now handles the purchasing logic. If the purchasing logic needs to be changed, it would require changes in the Terminal class, which could potentially affect other parts of the system that depend on it.

By contrast, the current design using the Purchasable interface adheres to these principles, making the code more flexible, maintainable, and easy to understand. It allows for easy changes to the purchasing logic without modifying existing classes, and it separates concerns by having different classes responsible for representing items and handling purchasing logic.