Assignment 2

Design Rationale Requirement 1.

Design Goal: The design goal for this assignment is to implement a scalable monster spawning system in Crater, allowing for flexible creation and management of different types of monsters with varying spawn probabilities.

Design Decision: In implementing the monster spawning system, the decision was made to utilize the Factory Method design pattern. We introduced the MonsterFactory interface to define a contract for creating different types of monsters and retrieving their spawn probabilities, actor, and double types. Concrete implementations of MonsterFactory (e.g., HuntsmanSpiderFactory) were then developed to encapsulate specific monster creation logic and spawn probabilities.

The rationale behind this decision is to promote modularity and extensibility. By separating the monster creation process into distinct factories, we adhere to the Single Responsibility Principle (S), ensuring that each factory is responsible for a specific type of monster. This approach facilitates easy extension by allowing the addition of new monster types without modifying existing code, aligning with the Open/Closed Principle (O).

Alternative Design: One alternative approach could involve a singular unmodifiable unique Spawner class responsible for creating a singular specific monster instance, with more such spawner classes being made to represent each new monster needing to be spawned.

Analysis of Alternative Design: The alternative design is not ideal because it violates various Design and SOLID principles:

> Single Responsibility Principle (S): The Spawner class would have multiple responsibilities (e.g., managing spawn probabilities and creating monsters), leading to less cohesive and maintainable code.
>
> Open/Closed Principle (O): Adding new monster types would require creating new Spawner classes, violating the principle of being closed to modification without code modification.
>
> Interface Segregation Principle (I): The alternative design does not segregate behaviors effectively, potentially leading to unnecessary dependencies and complexity.

Final Design: In our design, we closely adhere to relevant design principles or best practices:

Single Responsibility Principle (S):

- Application: Each MonsterFactory subclass is responsible for creating a specific type of monster (Actor) and managing its spawn probability.
- Why: Adhering to the SRP promotes code clarity and maintainability by ensuring that each class has a single, well-defined purpose.
- Pros/Cons: This approach simplifies the codebase, making it easier to understand and modify, but will require a class for each different monster type.

Open/Closed Principle (O):

- Application: The MonsterFactory interface allows for the creation of new MonsterFactory implementations without modifying existing code.
- Why: Upholding the OCP facilitates code extensibility and minimizes the risk of introducing bugs when extending the system.
- Pros/Cons: The system becomes more adaptable to change, but will require a class for each different monster type.

Liskov Substitution Principle (L):

- Application: All MonsterFactory subclasses can be used interchangeably where MonsterFactory is expected, without affecting system functionality.
- Why: Upholding the LSP ensures that subclasses can replace their base class without introducing errors or unexpected behaviors.
- Pros/Cons: This principle promotes polymorphism and supports a more flexible and robust system architecture, but requires careful consideration of subclass behaviors and contracts.

Interface Segregation Principle (I):

- Application: The MonsterFactory interface segregates monster creation behaviors (createMonster()) from spawn probability retrieval (getSpawnProbability()), promoting modular design.
- Why: By separating interfaces based on specific behaviors, we avoid unnecessary dependencies and promote code reusability.
- Pros/Cons: This approach reduces class complexity and promotes maintainability, but may result in a larger number of interfaces and implementations.

Dependency Inversion Principle (D):

- Application: The high-level Spawner class depends on abstractions (MonsterFactory interface) rather than concrete implementations (specific MonsterFactory subclasses).
- Why: Adhering to the DIP promotes loose coupling between components, reducing dependencies on specific implementations and facilitating easier maintenance and future enhancements.

- Pros/Cons: This design approach allows for interchangeable MonsterFactory implementations, enabling flexibility and facilitating testing and integration of new features.

Conclusion: Overall, our chosen design provides a robust framework for implementing a scalable and flexible monster spawning system. By adhering to relevant design principles such as SRP, OCP, and ISP, we have developed a solution that promotes code maintainability, extensibility, and modularity. This design lays a solid foundation for accommodating future enhancements and optimizations.