1.For SuspiciousTrader class:
SRP - Single Responsibility Principle：
The "SuspiciousTrader" class has a very clear responsibility, it just defines a Suspicious Trader. It includes the constructor (setting the name, char, etc.) and other behaviors that determine the player's behavior and the interactions that can be made with it. Therefore, this class follows the Single Responsibility Principle very well.

Open/Closed Principle - OCP:
The "SuspiciousTrader" class partially follows the Open/Closed Principle. For example, it allows extending the behavior by inheriting from the "Actor" base class, which determines what actions other actors can perform on the SuspiciousTrader without having to modify the code of the "Actor" class. However, judging by the "allowableActions" method in the code, this class does not fully use this principle.

Liskov Substitution Principle - LSP：
SuspiciousTrader is a subclass of Actor. Since it inherits the Actor class, SuspiciousTrader can be used as an Actor type. The playTurn() and allowableActions() methods in the code are overrides of the methods in the Actor class, which ensures that the subclass SuspiciousTrader can seamlessly replace the parent class. The rewritten playTurn() returns DoNothingAction: Even if the player uses SuspiciousTrader as an Actor, it conforms to the expected behavior of the parent class. allowableActions() conforms to all of the parent class: any other Actor object can interact with SuspiciousTrader normally and safely. Finally, SuspiciousTrader follows the LSP principle, and its behavior does not violate the contract of Actor, and can be replaced by the parent class Actor.

DIP - Dependency Inversion Principle：
The playTurn() and allowableActions() methods use the abstract classes Action and ActionList instead of concrete implementation classes. This makes the class highly extensible and interdependent. If trading logic is enabled, TradeAction will be added to ActionList as a concrete implementation instead of implementing the trading logic directly inside SuspiciousTrader. This is a good practice of dependency inversion.

2.For DivineBeastHead class:
SRP - Single Responsibility Principle：
The responsibility of DivineBeastHead is to implement a special weapon, define attack behaviors for it, and manage DivinePower. There are no extra responsibilities in the code, only for weapon functions.

OCP - Open/Closed Principle：
By inheriting WeaponItem: DivineBeastHead implements weapon extension through inheritance without modifying the original code of WeaponItem. Attack logic is extensible: If you need to enhance the logic of the attack() method in the future, you can rewrite or extend this method without modifying WeaponItem.

LSP - Liskov Substitution Principle：
DivineBeastHead inherits from WeaponItem, so it can be used as a WeaponItem class. The attack() method of the parent class is overridden to ensure that its behavior is consistent with

the parent class and does not violate the expected usage of the weapon class. Follows LSP principles, as it can be used as a WeaponItem type without breaking program logic.

DIP - Dependency Inversion Principle：
DivineBeastHead depends on the abstract class WeaponItem, not a concrete implementation. This improves interdependencies between modules. currentPower is a DivinePower type, not a concrete Wind implementation. This allows the DivinePower implementation to be replaced in the future without having to modify DivineBeastHead. Follows the DIP principle because it relies on abstractions (DivinePower and WeaponItem) rather than concrete implementations.

3.For FurnaceEngine class:
SRP - Single Responsibility Principle：
The FurnaceEngine class has very clear responsibilities: it represents a particular weapon and is responsible for executing the attack behavior through StompingFoot. Its task is only focused on the weapon's attack, and it does not handle behaviors related to the map or other game logic. Therefore, when the attack logic changes, only the code of StompingFoot needs to be modified, without large-scale modifications to FurnaceEngine. This reflects a good separation of responsibilities.

OCP - Open/Closed Principle：
The FurnaceEngine class extends WeaponItem by inheriting from WeaponItem without modifying the code of WeaponItem itself. It uses a combination approach and uses StompingFoot as its attack mechanism. This means that if you want to change the attack logic in the future, you only need to modify the StompingFoot class or replace its instance without modifying the FurnaceEngine code. If you need to add new weapon logic (such as adding a fire attack weapon), you can create a new class to extend WeaponItem without modifying the existing class.

LSP - Liskov Substitution Principle：
FurnaceEngine inherits from WeaponItem and can be used as a drop-in replacement for WeaponItem anywhere it is used. FurnaceEngine inherits all of the parent class's behavior and overrides the attack() method, but its behavior remains consistent and does not violate the parent class's expectations. For example, if a piece of code requires an object of type WeaponItem, passing it to FurnaceEngine will work just fine and the program logic will not be affected.

4. For RemembranceOfDancingLion class：
SRP - Single Responsibility Principle：
The RemembranceOfDancingLion class has a very clear responsibility: it represents a specific carryable item, the Remembrance Of Dancing Lion, and is responsible for initializing the item's properties, including its name, display character, and abilities.

OCP - Open/Closed Principle：
RemembranceOfDancingLion implements new functionality by extending the Item class without modifying the code of the Item class itself. Extending capabilities through the addCapability method complies with the OCP principle: if you need to add additional

functionality, you can extend the class or add a new class instead of modifying the existing class.

LSP - Liskov Substitution Principle：
RemembranceOfDancingLion is a subclass of Item, and its behavior and properties are consistent with the expectations of the Item class. If a piece of code uses an object of type Item, it will not cause an error to replace it with RemembranceOfDancingLion.

DIP - Dependency Inversion Principle：
In this class, there is no direct dependency on other concrete classes, but the capability extension is achieved through the Item abstract class and the ItemCapability enumeration. This shows that the design of this class complies with the DIP principle. It complies with the DIP principle because it depends on the abstract class Item and the enumeration ItemCapability, and does not directly depend on the concrete implementation.

5. For RemembranceOfTheFurnaceGolem class:
SRP - Single Responsibility Principle：
RemembranceOfTheFurnaceGolem's only responsibility is to represent a specific item (the memory of the Furnace Golem), and is only responsible for initializing its properties, such as its name, display character, portability, and abilities. This makes the class very clear in its responsibilities, and does not take on other non-item-related functionality.

OCP - Open/Closed Principle：
This class extends the Item class. Without modifying the Item, a new RemembranceOfTheFurnaceGolem class is added and given a new capability (ItemCapability.REMEMBRANCE_OF_FURNACE_GOLEM). If you need to add new items in the future, you only need to add a new subclass without modifying the existing code.

LSP - Liskov Substitution Principle：
RemembranceOfTheFurnaceGolem is a subclass of Item and its behavior is compatible with its parent class. If there is a piece of code that depends on Item, replacing it with RemembranceOfTheFurnaceGolem will not cause errors. This ensures that the class complies with the LSP principle. It complies with LSP because it can be safely replaced with the parent class Item without breaking the correctness of the program.

DIP - Dependency Inversion Principle：
The RemembranceOfTheFurnaceGolem class inherits the abstract class Item and does not directly depend on other specific implementations. It uses the ItemCapability enumeration to define capabilities. This abstract design reduces the dependence on specific implementations.

6. For ItemCapability class:
SRP - Single Responsibility Principle：
The ItemCapability enumeration class's sole responsibility is to list all possible item capabilities. It defines two capabilities, REMEMBRANCE_OF_DANCING_LION and REMEMBRANCE_OF_FURNACE_GOLEM, and can be extended in the future. The class's responsibility is very clear, which is to provide a uniform definition of capabilities for items.

OCP - Open/Closed Principle：
ItemCapability defines the capabilities of items in an enumeration. When you need to add a new capability, you only need to add a new value to the enumeration without modifying the existing code. This complies with the requirements of the OCP principle: to expand functionality without modifying the original code.

7. For TradeAction class:
SRP - Single Responsibility Principle:
The TradeAction class is only responsible for handling the trade logic, that is, removing the specified item from the player's inventory and giving a reward. All trade-related actions are completed in this class, and there is no other unrelated logic. This makes the class's responsibility single and clear.

OCP - Open/Closed Principle:
Different item capabilities are defined through the ItemCapability enumeration to avoid frequent modifications to the TradeAction code. If you want to add a new trading item, such as adding a "REMEMBRANCE_OF_NEW_BEAST", you only need to extend ItemCapability without modifying the core logic of TradeAction.

LSP - Liskov Substitution Principle:
TradeAction inherits from Action and overrides the execute() and menuDescription() methods, which ensures that TradeAction can be seamlessly substituted wherever an Action type is required.

DIP - Dependency Inversion Principle:
TradeAction relies on the abstract ItemCapability and Actor instead of the specific implementation. For example, TradeAction determines the item type through ItemCapability instead of directly relying on the specific item class.