Here's an assignment designed to help you practice the SOLID principles by creating an **E-commerce Order Management System**. This assignment will focus on applying each of the SOLID principles to ensure your code is clean, extensible, and adheres to best practices in object-oriented design.

Assignment: E-commerce Order Management System

Objective:

To design an E-commerce Order Management System using the SOLID principles, focusing on clean, modular, and maintainable code.

Requirements:

1. **Classes**: Implement the following core classes:

    - `Product`: Represents an item available for purchase.
    - `Order`: Represents an order containing multiple products.
    - `Customer`: Represents a customer placing an order.
    - `OrderProcessor`: Handles the processing of orders.
    - `PaymentProcessor`: Handles payment processing.
    - `InvoiceGenerator`: Generates an invoice for the order.

2. **Requirements and SOLID Principles**:

    Implement the following requirements, focusing on how each requirement addresses a SOLID principle.

---

1. Single Responsibility Principle (SRP)

Each class should have a single responsibility and only one reason to change.

- **Product**:

    - **Attributes**: `name`, `price`, `stock`
    - **Methods**: `get_details()`, `reduce_stock(quantity)`

- **Customer**:

    - **Attributes**: `name`, `email`, `address`
    - **Methods**: `get_contact_info()`

- **InvoiceGenerator**:

    - **Methods**: `generate_invoice(order)`: Takes an `Order` instance and generates an invoice for it.

---

2. Open/Closed Principle (OCP)

The system should be open for extension but closed for modification.

- **OrderProcessor**:

    - **Methods**: `process_order(order, payment_type)`: Processes an order with the specified payment type.
    - Use an interface for payment processing that allows additional payment types without modifying `OrderProcessor`.

    Implement at least two payment types, e.g., `CreditCardProcessor` and `PayPalProcessor`, both of which implement a `PaymentProcessor` interface with a `process_payment(amount)` method.

---

3. Liskov Substitution Principle (LSP)

Derived classes should be substitutable for their base classes.

- Ensure that each `PaymentProcessor` subclass (e.g., `CreditCardProcessor` and `PayPalProcessor`) can be used interchangeably within the `OrderProcessor` without breaking the functionality. This means `OrderProcessor` should only rely on the `PaymentProcessor` interface for processing payments, without needing to know the specific payment type.

---

4. Interface Segregation Principle (ISP)

Classes should not be forced to implement interfaces they don’t use.

- Define interfaces with specific responsibilities:

    - `IPaymentProcessor`: Contains methods for `process_payment(amount)`.
    - `IStockManager`: Contains methods for `check_stock()` and `update_stock()`.

    Ensure that `Product` only implements the `IStockManager` interface and doesn't contain payment methods.

---

5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules but on abstractions.

- Use dependency injection to inject dependencies like `PaymentProcessor` and `InvoiceGenerator` into the `OrderProcessor` class.
- The `OrderProcessor` should depend on abstractions (interfaces) rather than concrete implementations.

Additional Requirements

Implement each functionality as specified below, using the SOLID principles:

1. **Add Products to Order**: The `Order` class should allow adding multiple `Product` instances.
2. **Process Payment**: The `OrderProcessor` should process payments using the `PaymentProcessor` interface.
3. **Generate Invoice**: After successful payment, `OrderProcessor` should use `InvoiceGenerator` to create an invoice.
4. **Stock Management**: When an order is placed, the system should check product stock and update it accordingly.

Bonus Tasks (Optional):

1. Add a new payment method, such as `BankTransferProcessor`, by implementing the `PaymentProcessor` interface without modifying `OrderProcessor`.
2. Implement logging for order and payment processing.

Example Interaction

```
Unset
1. Customer selects products to add to the order.
2. OrderProcessor checks stock and processes the order.
3. PaymentProcessor charges the customer based on selected payment type.
4. InvoiceGenerator generates an invoice if payment is successful.
5. System updates product stock.
```

Deliverables

1. Code with comments explaining each class, method, and how it meets SOLID principles.
2. A README file explaining how to run the code and a brief summary of SOLID principles applied.
3. A sample interaction script showing the system in action.