E-Commerce Order Management - Legacy Code Refactoring Project

Overview

This is a **substantial legacy codebase** (~800+ lines) that needs comprehensive refactoring. Your task is to transform this monolithic, poorly-structured code into a clean, maintainable, well-tested system following SOLID principles.

System Features (Current Implementation)

The legacy system includes:

1. **Order Management**
   - Complex order processing with multiple discount types
   - Order status tracking and updates
   - Order cancellation and refunds
2. **Inventory Management**
   - Product stock tracking
   - Low stock alerts
   - Restock operations
   - Inventory logging
3. **Customer Management**
   - Customer profiles with membership tiers
   - Loyalty points system
   - Order history tracking
   - Customer lifetime value calculation
   - Automatic membership upgrades
4. **Pricing & Discounts**
   - Membership-based discounts (Gold, Silver, Bronze)
   - Promotional codes with expiration
   - Bulk purchase discounts
   - Loyalty points redemption
   - Category-specific promotions

5. **Shipping & Logistics**
    - Multiple shipping methods (standard, express, overnight)
    - Weight-based shipping calculations
    - Shipment tracking
    - Free shipping thresholds
6. **Supplier Management**
    - Supplier tracking
    - Automatic reorder notifications
    - Supplier reliability scoring
7. **Reporting & Analytics**
    - Sales reports
    - Revenue by category
    - Top customers analysis
    - Product performance tracking
8. **Marketing**
    - Customer segmentation
    - Targeted email campaigns
    - Inactive customer identification

Current Problems (What Makes This Legacy Code)

Major SOLID Violations

1. **Single Responsibility Principle (SRP) - SEVERELY VIOLATED**
    - `process_order()` does EVERYTHING: validation, pricing, discounts, payment, inventory, notifications, loyalty points, supplier alerts
    - Functions mix business logic, data access, and presentation
    - `generate_sales_report()` handles data aggregation, calculation, and customer analysis
2. **Open/Closed Principle (OCP) - VIOLATED**
    - All discount logic hardcoded in `process_order()`
    - Shipping calculation embedded with no extension points
    - Tax calculation hardcoded for specific states
    - Adding new discount types requires modifying core function
3. **Liskov Substitution Principle (LSP) - NOT APPLICABLE**
    - No abstractions or inheritance to evaluate
4. **Interface Segregation Principle (ISP) - NOT APPLICABLE**
    - No interfaces defined at all

5. **Dependency Inversion Principle (DIP) - SEVERELY VIOLATED**
    - Everything depends on global state
    - No abstractions, no interfaces
    - Direct access to concrete data structures
    - Impossible to inject dependencies or swap implementations

Additional Code Smells

6. **Global State**
    - Multiple global dictionaries (products, customers, orders, suppliers, etc.)
    - Global counters (next_order_id, next_shipment_id)
    - Not thread-safe
    - Impossible to test in isolation
7. **God Functions**
    - `process_order()` : ~150 lines doing everything
    - `generate_sales_report()` : Multiple responsibilities
    - Functions are untestable due to complexity and global dependencies
8. **Hardcoded Business Rules**
    - Membership discounts hardcoded
    - Tax rates hardcoded by state
    - Shipping calculations embedded
    - Loyalty point rules hardcoded
    - No flexibility to change rules
9. **No Type Hints**
    - No static type checking possible
    - Function signatures unclear
    - Runtime errors likely
10. **No Error Handling**
    - Reliance on print statements for errors
    - No exceptions, no error codes
    - Inconsistent return values (None, False, objects)
11. **Tight Coupling**
    - Order processing tightly coupled to inventory, customer, pricing, payment
    - Cannot change one aspect without risk to others
    - Cannot test components independently
12. **No Tests**
    - Zero unit tests
    - Code structure makes testing nearly impossible

- No mocking strategy possible with global state

Your Tasks (50 Hours - One Week Project)

Phase 1: Analysis & Planning (5 hours)

- Read and understand the entire codebase
- Document all SOLID violations
- Identify all responsibilities and create a separation plan
- Design the target architecture
- Plan refactoring strategy (what order to tackle)

Phase 2: Extract Domain Models (5 hours)

- Move Product, Customer, Order, etc. to proper domain layer
- Add validation logic to domain models
- Add type hints to all domain classes
- Create value objects where appropriate (Address, Money, etc.)

Phase 3: Create Service Layer (15 hours)

Extract separate services with single responsibilities:

- **ProductService**: Product management, inventory checks
- **InventoryService**: Stock management, restock operations, logging
- **CustomerService**: Customer management, loyalty points
- **PricingService**: All pricing logic
  - Create strategy pattern for different discount types
  - Membership discounts
  - Promotional discounts
  - Bulk discounts
  - Loyalty discounts
- **OrderService**: Order creation and management
- **PaymentService**: Payment processing and validation
- **ShippingService**: Shipping calculations, tracking
- **NotificationService**: Email/SMS notifications
- **ReportingService**: Sales reports and analytics
- **SupplierService**: Supplier management and communications

Phase 4: Create Repository Layer (8 hours)

- Define repository interfaces (Protocols)
- Implement in-memory repositories for testing
- Separate data access from business logic
- Remove all global state

Phase 5: Apply Dependency Injection (5 hours)

- Refactor services to accept dependencies via constructors
- Create a proper application orchestrator
- Wire dependencies explicitly
- No more global state access

Phase 6: Add Comprehensive Tests (10 hours)

Write unittest-based tests for:

- **Domain models**: Validation, business rules
- **Services**: Each service with mocked dependencies
  - Test all discount calculation paths
  - Test inventory deduction and restoration
  - Test membership upgrade logic
  - Test order cancellation flow
- **Integration tests**: End-to-end order processing
- **Edge cases**: Out of stock, invalid payment, expired promos
- Aim for >80% code coverage

Phase 7: Add Type Hints & Mypy (2 hours)

- Add type hints to all functions and methods
- Configure mypy for strict checking
- Ensure `mypy .` passes with no errors
- Use proper types (not just basic types)

Expected Deliverables

1. **Refactored Codebase**
   - Clear package structure (domain/, services/, repositories/, application/)

- No global state
- All SOLID principles applied
- Clean separation of concerns

2. **Comprehensive Test Suite**
   - Unit tests for all services
   - Integration tests for key workflows
   - > 80% code coverage
   - All tests passing
3. **Complete Type Hints**
   - All public APIs typed
   - mypy --strict passes
   - Clear function signatures
4. **Documentation**
   - Architecture diagram
   - Service responsibilities documented
   - How to extend the system (adding new discount types, payment methods)
   - Updated README with new architecture
5. **Working System**
   - All original functionality preserved
   - main.py still works (refactored)
   - Demonstrable improvements in maintainability

Suggested Architecture

```
domain/
  models/           # Product, Customer, Order, OrderItem
  value_objects/    # Money, Address, Email, etc.
  enums/           # OrderStatus, MembershipTier, ShippingMethod

services/
  product_service.py
  inventory_service.py
  customer_service.py
  pricing/
    pricing_service.py
    strategies/     # Different discount strategies
      membership_discount.py
      promotional_discount.py
      bulk_discount.py
      loyalty_discount.py
  order_service.py
  payment_service.py
  shipping_service.py
  notification_service.py
  reporting_service.py
  supplier_service.py

repositories/
  interfaces/       # Repository protocols
  in_memory/       # In-memory implementations for testing

application/
  order_processor.py    # Orchestrates services

tests/
  test_domain/
  test_services/
  test_integration/
```

Getting Started

1. **Run the current system**:

```
python main.py
```

Understand what it does.

2. **Try to run mypy** (it will fail):

```
mypy order_system.py
```

3. **Start refactoring incrementally**:
   - Extract one service at a time
   - Write tests as you go
   - Keep the system working after each change

Assessment Criteria

- **SOLID Principles** (30%): Clear demonstration of all 5 principles
- **Architecture** (20%): Clean separation, proper layering
- **Test Coverage** (25%): Comprehensive tests, >80% coverage
- **Type Safety** (10%): Complete type hints, mypy passes
- **Code Quality** (15%): Readability, maintainability, documentation

Tips for Success

1. **Don't try to refactor everything at once**
2. **Extract services one at a time**
3. **Write tests immediately after extracting each service**
4. **Keep main.py working throughout** (refactor it last)
5. **Use dependency injection from the start**
6. **Create interfaces (Protocols) before implementations**
7. **Focus on one SOLID principle at a time**
8. **Commit frequently** (if using git)

Time Estimates

- Understanding codebase: 5 hours
- Domain layer: 5 hours
- Service extraction: 15 hours
- Repository layer: 8 hours

- Dependency injection: 5 hours
- Testing: 10 hours
- Type hints: 2 hours

Total: ~50 hours

Good luck! This is a realistic refactoring project that mirrors real-world legacy code challenges.