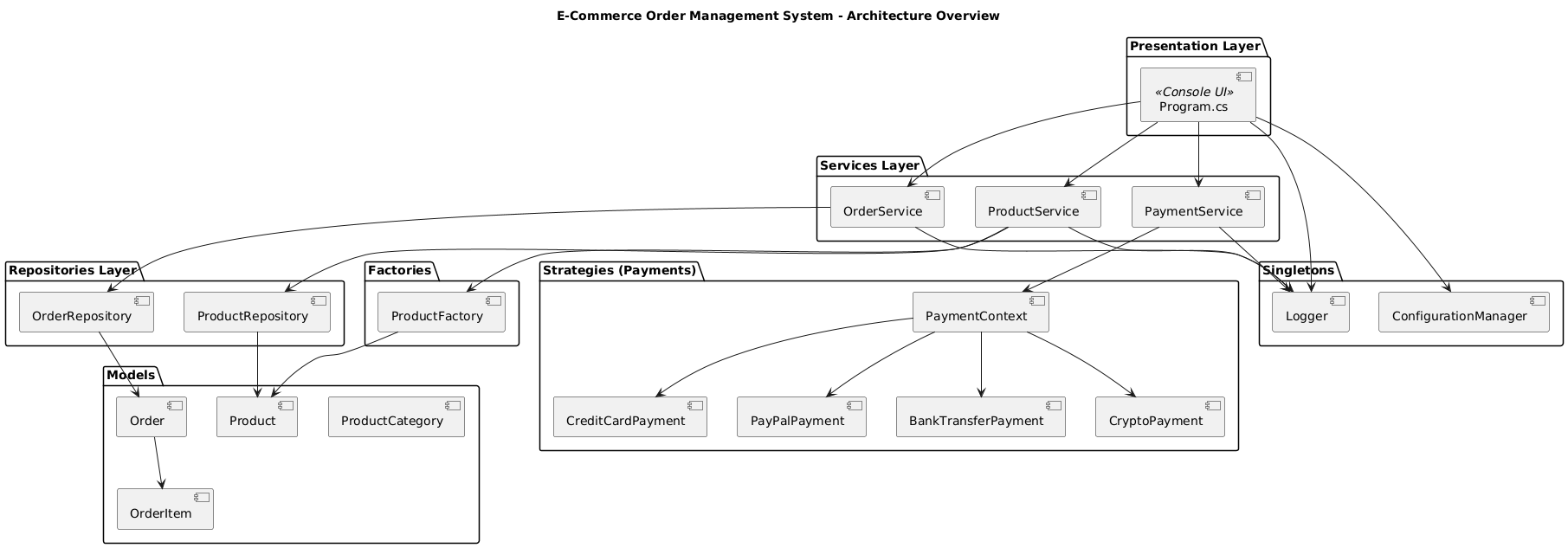
# **Architecture Overview**



## **2. Design Pattern Implementations**

### **1. Factory Pattern – Product Creation**

**Goal:** Dynamically create different product types (Electronics, Clothing, Books) based on user input.

* **Where:** ProductFactory.cs
* **How:** Uses ProductCategory enum to return concrete product types.

**Example:**  
var product = ProductFactory.CreateProduct(ProductCategory.Electronics, "Laptop", 1000);

### **2. Singleton Pattern – Configuration & Logging**

**Goal:** Centralized and consistent configuration and logging system.

* **Where:** ConfigurationManager.cs, ConsoleLogger.cs
* **How:** Thread-safe single instance of logger/config manager using static properties.

**Example:**  
Logger.Instance.LogInfo("Order created");

### **3. Strategy Pattern – Payment Methods**

**Goal:** Handle different payment methods at runtime (Credit Card, PayPal, Bank, Crypto).

* **Where:** Strategies/
* **How:** Each class implements IPaymentStrategy; selected via PaymentContext.

**Example:**  
var paymentContext = new PaymentContext(new PayPalPayment());

paymentContext.ProcessPayment(100);

### **4. Repository Pattern – Data Access**

**Goal:** Decouple business logic from data storage.

* **Where:** ProductRepository.cs, OrderRepository.cs
* **How:** Implements IRepository<T> and custom methods like GetByCategory().
* **Example:**  
  var products = await productRepository.GetByCategory(ProductCategory.Clothing);

## **3. SOLID Principles in Practice**

### **Single Responsibility**

* Each class has one responsibility (e.g., Logger logs, OrderService manages orders).

### **Open/Closed**

* Product creation is open for extension (new categories) but closed for modification (ProductFactory switch case).

### **Liskov Substitution**

* All Product subclasses can be used wherever Product is expected.

### **Interface Segregation**

* Separate interfaces for ILogger, IConfigurationManager, IPaymentStrategy.

### **Dependency Inversion**

* High-level services depend on abstractions (interfaces), not concrete implementations.

## **4. Challenges & Solutions**

### **Challenge 1: Handling multiple payment types**

* **Problem:** Each payment method required unique validation.
* **Solution:** Used Strategy Pattern to isolate validation logic in separate classes.

### **Challenge 2: Avoiding duplicate logic across product types**

* **Problem:** Common product fields were repeated.
* **Solution:** Created a base Product class and extended it for specifics.

### **Challenge 3: Managing object creation cleanly**

* **Problem:** Product creation logic was cluttered in Program.cs.
* **Solution:** Moved creation logic to a centralized ProductFactory.

### **Challenge 4: Thread-safe singletons**

* **Problem:** Logger and config manager had race conditions during testing.
* **Solution:** Used lazy instantiation with **lock** to ensure thread safety.

## **5. Future Enhancements :**

* Add **order cancellation** and refund flow
* Integrate with **real payment gateways**
* Convert to **Web API** using ASP.NET Core for scalability
* Replace in-memory repositories with **database** or **file-based** persistence