**Factory Pattern in Logistics Application**

**Detailed Problem Statement:**

**Background and Context**

In the modern logistics industry, shipping companies manage a diverse array of shipment types to cater to varying customer needs, delivery timelines, and geographical constraints. A typical shipping company, such as "GlobalShip Logistics," handles thousands of shipments daily, ranging from domestic ground deliveries to urgent air express services and large-scale international sea freight. Each shipment type involves unique operational requirements, cost calculations, regulatory compliance, and tracking mechanisms. For instance:

* **Ground Shipments**: These are cost-effective for local or regional deliveries, primarily involving road transport. Costs are typically based on weight, distance, and fuel surcharges, with standard delivery times of 2-5 days.
* **Air Shipments**: Designed for time-sensitive packages, these use air freight and include priority handling fees, faster transit times (often 1-2 days), and additional charges for expedited customs if international.
* **Sea Shipments**: Ideal for bulk or oversized cargo over long distances, these involve ocean freight with considerations for container sizes, port fees, customs declarations, and longer delivery windows (7-30 days).

The company's software system must efficiently create and manage these shipment objects dynamically based on customer selections via a user interface (e.g., a web portal or mobile app). However, directly instantiating specific shipment classes in the client code (e.g., using new GroundShipment()) leads to tight coupling, making the system rigid and difficult to maintain. If a new shipment type (e.g., "rail" or "drone") is introduced due to evolving business needs, the client code would require extensive modifications, violating the Open-Closed Principle (open for extension, closed for modification).

To address this, the system requires a centralized mechanism to handle object creation, promoting loose coupling, scalability, and ease of extension. The Factory Pattern is ideal for this scenario, as it provides an interface for creating objects while delegating the instantiation logic to subclasses or a factory class, hiding the complexity from the client.

**Requirements**

1. **System Architecture**:
   * Define a common interface IShipment that all shipment types must implement. This interface should include a method CalculateCost(double weight, double distance) to compute the shipment cost based on provided parameters. Optionally, include methods for other behaviors like GetEstimatedDeliveryTime() or GenerateTrackingInfo(), but focus primarily on cost calculation for this case study.
2. **Concrete Shipment Classes**:
   * **GroundShipment**: Cost formula: weight \* 0.5 + distance \* 0.1 (base rate per kg and per km). Assume standard road transport with no additional fees.
   * **AirShipment**: Cost formula: weight \* 2.0 + distance \* 0.5 + 50 (higher rate plus a fixed priority fee of $50).
   * **SeaShipment**: Cost formula: weight \* 0.3 + distance \* 0.2 + 100 (lower variable rate plus a fixed customs/port fee of $100).
   * Each class should encapsulate its specific logic, ensuring that future changes (e.g., updating fees) are isolated.
3. **Factory Implementation**:
   * Create a ShipmentFactory class with a method GetShipment(string shipmentType) that accepts a string input (e.g., "ground", "air", "sea") and returns the corresponding IShipment instance.
   * Use case-insensitive comparison for the input string to handle variations like "Ground" or "AIR".
   * If the input is null or an invalid type, throw appropriate exceptions (e.g., ArgumentNullException or ArgumentException) with meaningful messages.
   * The factory should be the single point of creation, allowing easy addition of new types by extending the factory without altering client code.
4. **Client Usage and Demonstration**:
   * In a main program (e.g., console application), instantiate the factory and use it to create shipments based on sample inputs.
   * Demonstrate cost calculations for each type with example values (e.g., weight = 10 kg, distance = 100 km).
   * Show error handling by attempting an invalid type (e.g., "space") and catching the exception.
   * Ensure the client code interacts only with the factory and the IShipment interface, not the concrete classes directly.
5. **Non-Functional Requirements**:
   * **Flexibility**: The design must allow adding a new shipment type (e.g., "RailShipment" with cost weight \* 0.4 + distance \* 0.15) by only modifying the factory class, without changing existing client code.
   * **Error Handling**: Robust validation for inputs to prevent runtime errors.
   * **Code Quality**: Follow C# best practices, including proper naming conventions, comments, and namespace organization.

**Expected Outcomes**

* The application should output calculated costs for valid shipments and handle errors gracefully.
* This pattern ensures the logistics system remains maintainable as the company expands its services, such as integrating with third-party carriers or adding eco-friendly shipment options.