# Build Payment Gateway System

## *Problem statement:*

- *Functional requirement:*
  - Should support ***multiple providers*** (e.g. Paytm, PhonePe, PayPal etc).
  - We can easily ***add new gateways*** in future.
  - There should be a standard ***payment flow***, with required ***validations***.
  - Have ***error handling*** & ***retries mechanism***.
- *Non-Functional requirement:*
  - Entire design should be ***scalable***.
  - Model should be ***plug*** and ***play*** in nature.
  - Should follow ***SOLID principles***.

---

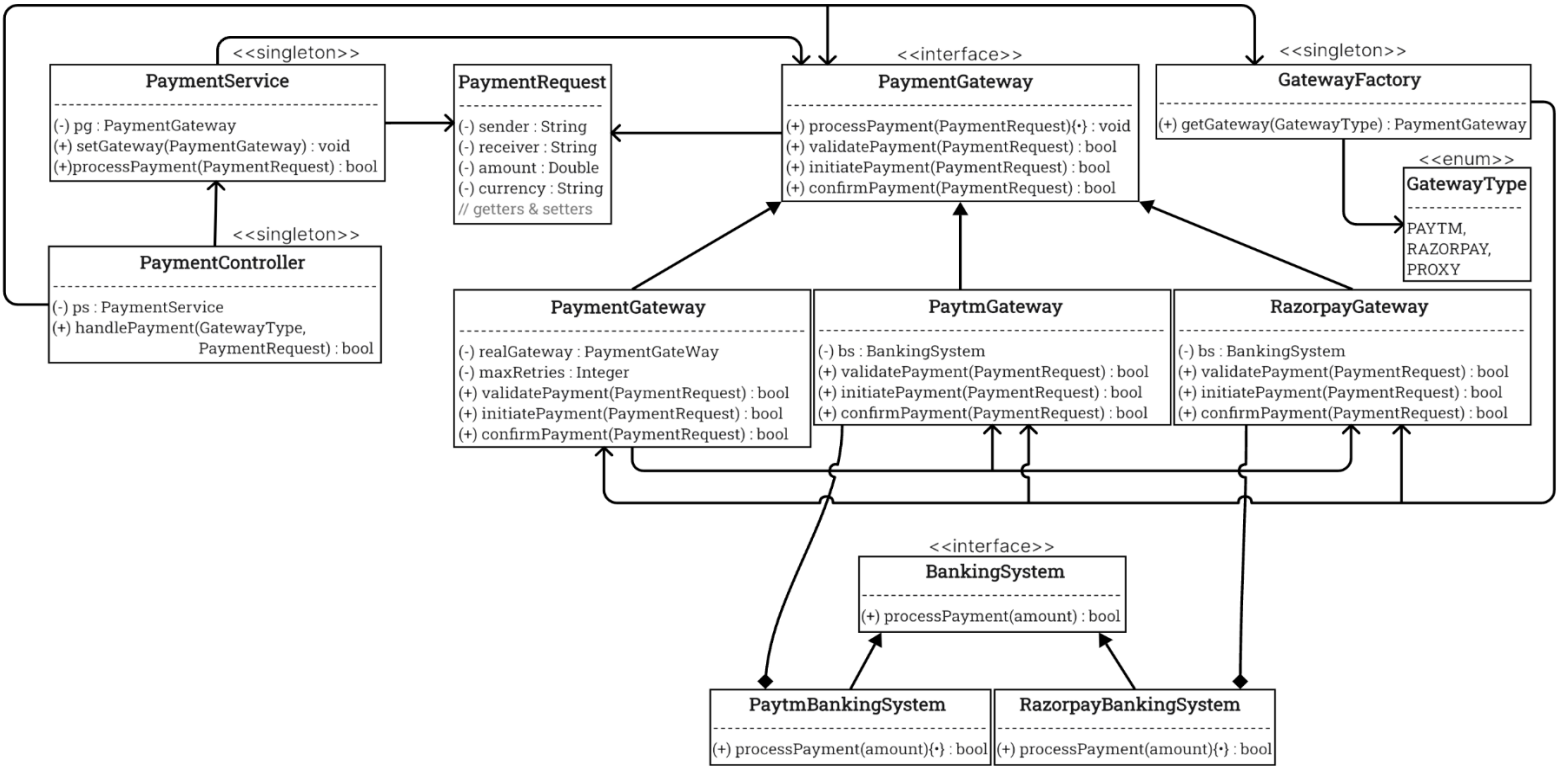
## *Applied Design Patterns:*

1. **Strategy Pattern**
   - Where Used: PaymentGateway interface and its implementations.
   - Why: Each provider defines its own algorithm for executing payments while exposing the same interface.
   - Benefit:
     - Makes it easy to swap providers at runtime.
     - Supports Open/Closed Principle → new gateways can be added without modifying core logic.
2. **Proxy Pattern (Protection Proxy / Retry Proxy)**
   - Where Used: ProxyPaymentGateway wraps real gateways.
   - Why: Adds retry mechanism, error handling, logging, and resilience without altering gateway logic.
   - Benefit:
     - Improves reliability.
     - Separates cross-cutting concerns from business logic.
3. **Template Method Pattern**
   - Where Used: In the PaymentGateway abstract class, the processPayment() method defines the fixed sequence of steps:
     1. validatePayment()
     2. initiatePayment()
     3. confirmPayment()
   - Why: To enforce a common flow across all payment providers while allowing subclasses (PaytmGateway, RazorpayGateway) to implement the details of each step.
   - Benefit:
     - Guarantees a standardized payment lifecycle.
     - Ensures consistency while still allowing flexibility in implementation.
4. **Factory Method / Factory Pattern**
   - Where Used: GatewayFactory creates appropriate PaymentGateway based on GatewayType.
   - Why: To abstract away the creation logic from the client (PaymentService).
   - Benefit:
     - Centralized object creation.
     - Makes the system plug-and-play for new providers.
5. **Singleton Pattern**
   - Where Used: PaymentService and GatewayFactory.
   - Why: Ensure that only one instance of these core services exists.
   - Benefit:
     - Avoids inconsistent state.
     - Optimizes resource usage.

***How These Patterns Solve the Problem:***

1. Multiple Providers & Extensibility → Achieved with **Strategy** + **Factory**.
2. Standard Payment Flow → Enforced via **Template Method Pattern**.
3. Error Handling & Retries → Managed using **Proxy Pattern**.
4. Scalability & Plug-and-Play → Enabled by **Factory** and interface-driven design.
5. Centralized Orchestration → Achieved via **Singletons** (PaymentService, GatewayFactory).
6. SOLID Principles:
    - **SRP:** Each class has one responsibility.
    - **OCP:** New gateways added without modifying existing code.
    - **LSP:** All gateways interchangeable via the common interface.
    - **ISP:** Lean interfaces (PaymentGateway, BankingSystem).
    - **DIP:** High-level modules depend on abstractions, not concrete implementations.

**UML:**



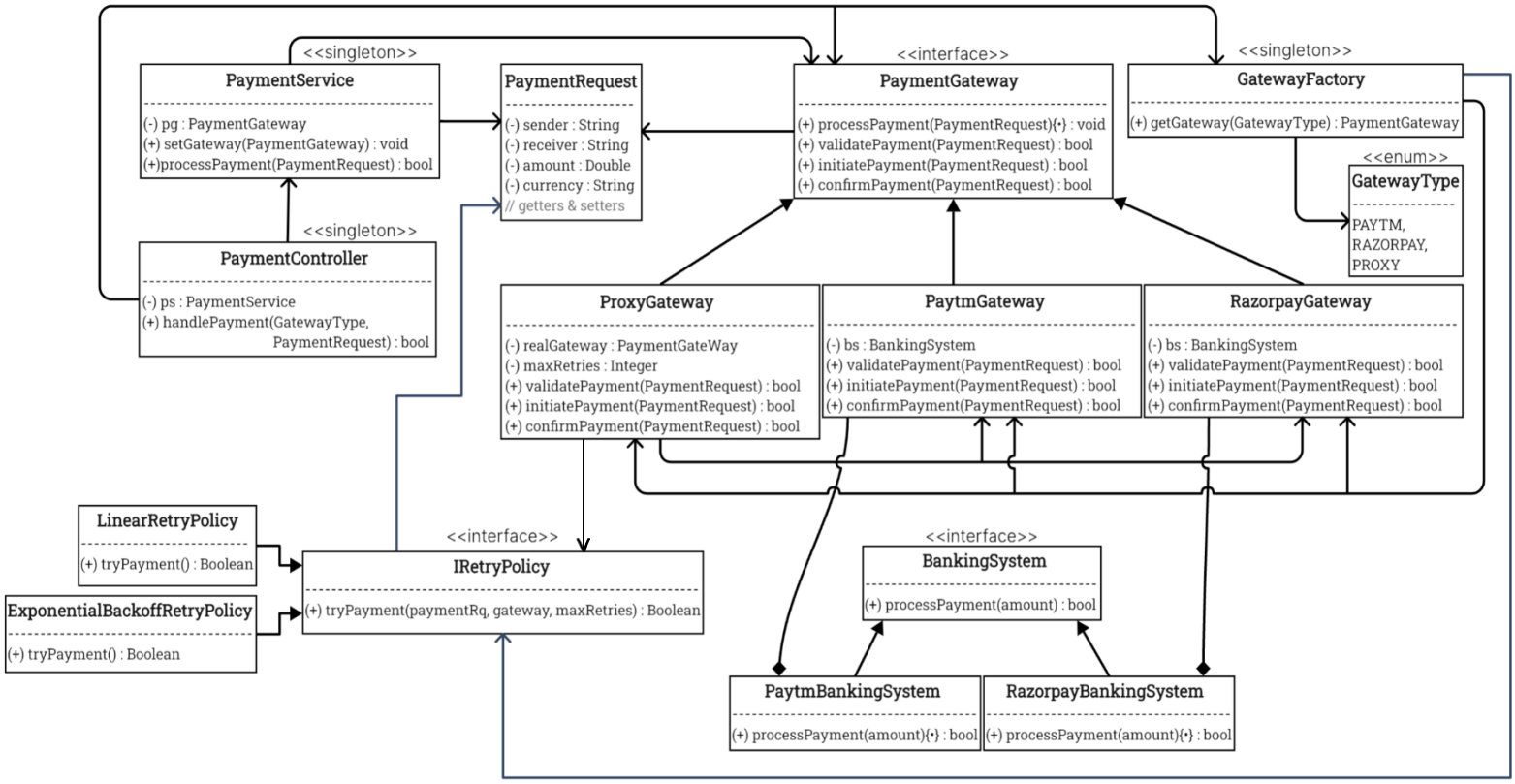

We can extend it by adding recurring strategies. Means for different gateways there are different ***recurring strategies*** like,

1. ***Razorpay*** has more success rate, so we should use ***linear retry*** there.
2. ***Paytm*** has less success rate, so we should use ***exponential backoff retry*** there.

In this case we have to use ***strategy design pattern*** which makes it more **scalable** and **maintainable**.

Gateway Proxy going to contain an extra variable ***retryPolicy*** basically strategy and we can choose retry policy at the time **gateway object creating**.

UML of payment gateway with retry strategy implementation:




**Code link:** https://github.com/sibasundarj8/-System-Design-/tree/main/Projects/Payment%20Gateway%20System