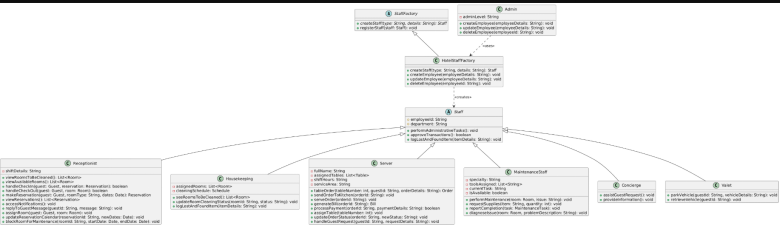
**Design Patterns**

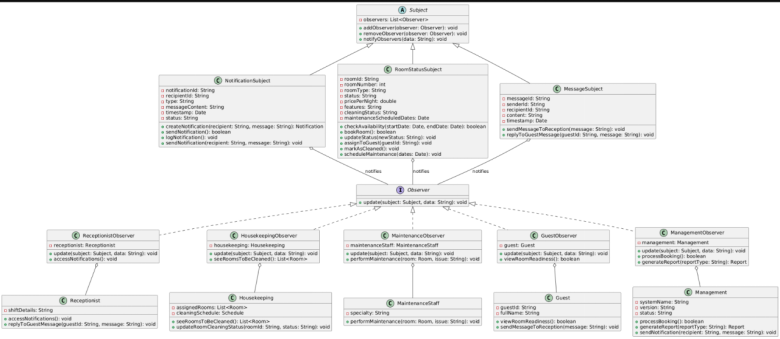
**Factory (Arsildo Veliu,Megi Almadhi)**

The Factory Pattern in this hotel management system provides a centralized and flexible way to create different types of staff objects (Receptionist, Housekeeping, Server, MaintenanceStaff, Concierge, and Valet) without exposing the complex instantiation logic to client code. The HotelStaffFactory acts as a single point of creation where one can request a specific type of staff member by passing a type string and details, and it handles all the construction complexities behind the scenes. This design promotes loose coupling between the staff creation process and the rest of the system, making it easier to add new staff types in the future, modify existing staff creation logic, or change how staff objects are initialized without affecting the Admin class or other parts of the system that need to create staff members. The pattern also enables consistent staff registration and management through the factory's registerStaff() method, ensuring all created staff members are properly integrated into the hotel management system.



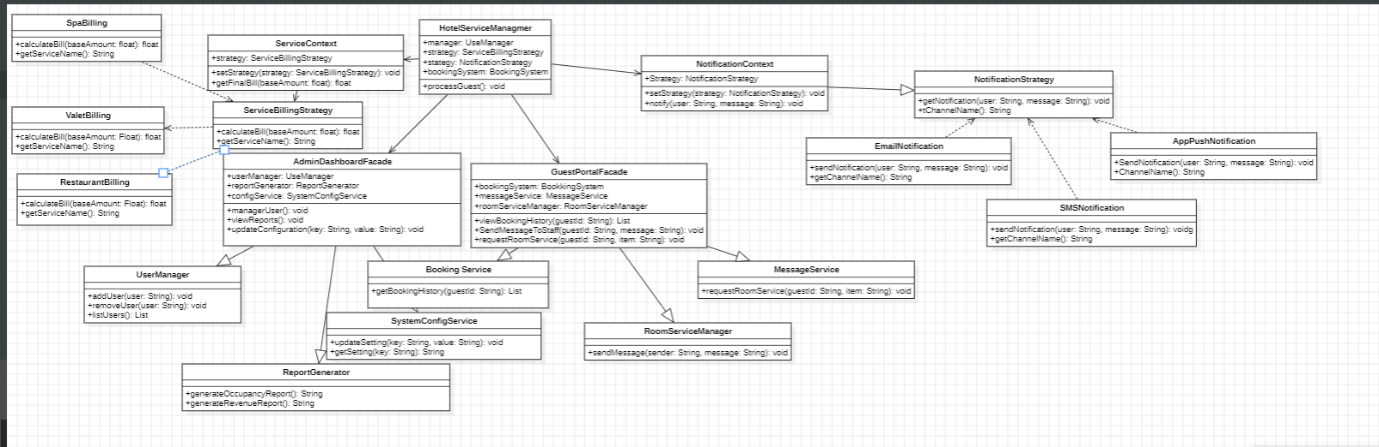
**Observer (Arsildo Veliu,Megi Almadhi)**

The Observer Pattern in this hotel management system establishes a publish-subscribe mechanism that enables automatic communication and coordination between different hotel departments and stakeholders when important events occur. The pattern uses three main subjects (NotificationSubject, RoomStatusSubject, and MessageSubject) that maintain lists of observers and automatically notify them when state changes happen, such as when room status updates, new messages arrive, or notifications are created. Each observer type (ReceptionistObserver, HousekeepingObserver, MaintenanceObserver, GuestObserver, and ManagementObserver) represents a different stakeholder in the hotel system that needs to be informed about specific changes - for example, when a room's cleaning status changes, the HousekeepingObserver is automatically notified so housekeeping staff can see updated room lists, while the ReceptionistObserver gets notified about new guest messages so reception can respond promptly. This pattern eliminates the need for constant polling or manual checking of status changes, ensures real-time updates across all hotel departments, and maintains loose coupling between different parts of the system since subjects don't need to know the specific details of who is listening to them.



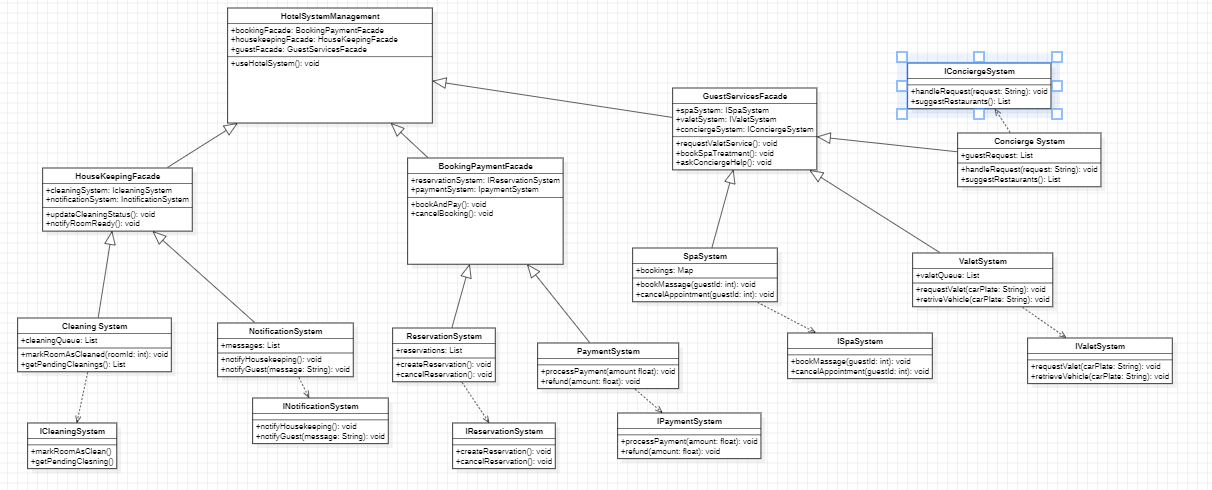
Strategy (Artemisa Hasalami,Brikena Papadhopuli,Ester  Pashtranjaku)

In this hotel management system, the Strategy pattern is applied to support dynamic and interchangeable behaviors such as billing, payment, and room assignment. Each behavior is represented by an interface (e.g., BillingStrategy, PaymentStrategy), and multiple concrete classes implement these interfaces to provide specific algorithms (e.g., PremiumBilling, CashPayment, VIPRoomAssignment). These strategy objects are used through context classes that hold them via aggregation, allowing the system to switch behaviors at runtime without modifying the context logic. This approach follows object-oriented principles such as the Open/Closed Principle and improves flexibility, scalability, and testability of the system.



**Facade (Artemisa Hasalami,Brikena Papadhopuli,Ester Pashtranjaku)**

The Facade pattern is used to simplify and organize access to complex hotel subsystems such as reservation, payment, cleaning, valet, spa, and concierge services. Instead of exposing these subsystems directly to higher-level classes, the system introduces facades like BookingPaymentFacade, HousekeepingFacade, and GuestServicesFacade. Each facade composes its related subsystems using composition, meaning the subsystems are fully managed and encapsulated within the facade. Additionally, interfaces are defined for both subsystems and facades to support abstraction and dependency injection, allowing easy testing and future extensions. This design reduces coupling between components, promotes clean architecture, and improves maintainability by hiding internal complexity.



**Singleton (Erta Llenga, Esta Çekrezi)**

The Singleton design pattern is implemented for three critical services in the hotel management system: SharedConfiguration, StatsDashboard, and Logging. This pattern ensures that each of these classes has only one instance throughout the application's lifecycle, accessed through a static getInstance() method while preventing direct instantiation via private constructors. SharedConfiguration manages system-wide settings like hotel policies, authentication parameters, and operating hours that must remain consistent across all system components. StatsDashboard provides centralized statistics and reporting capabilities, ensuring all users see the same real-time occupancy rates, revenue projections, and performance metrics. Logging centralizes all system activities, audit trails, and user actions into a single, consistent log system. This pattern is ideal for these services because they represent shared system resources that require global consistency, prevent resource duplication, maintain data integrity, and provide centralized control - ensuring that when a Manager checks occupancy rates, an Admin configures system settings, or any user action gets logged, they all interact with the same unified instances rather than conflicting separate objects.

