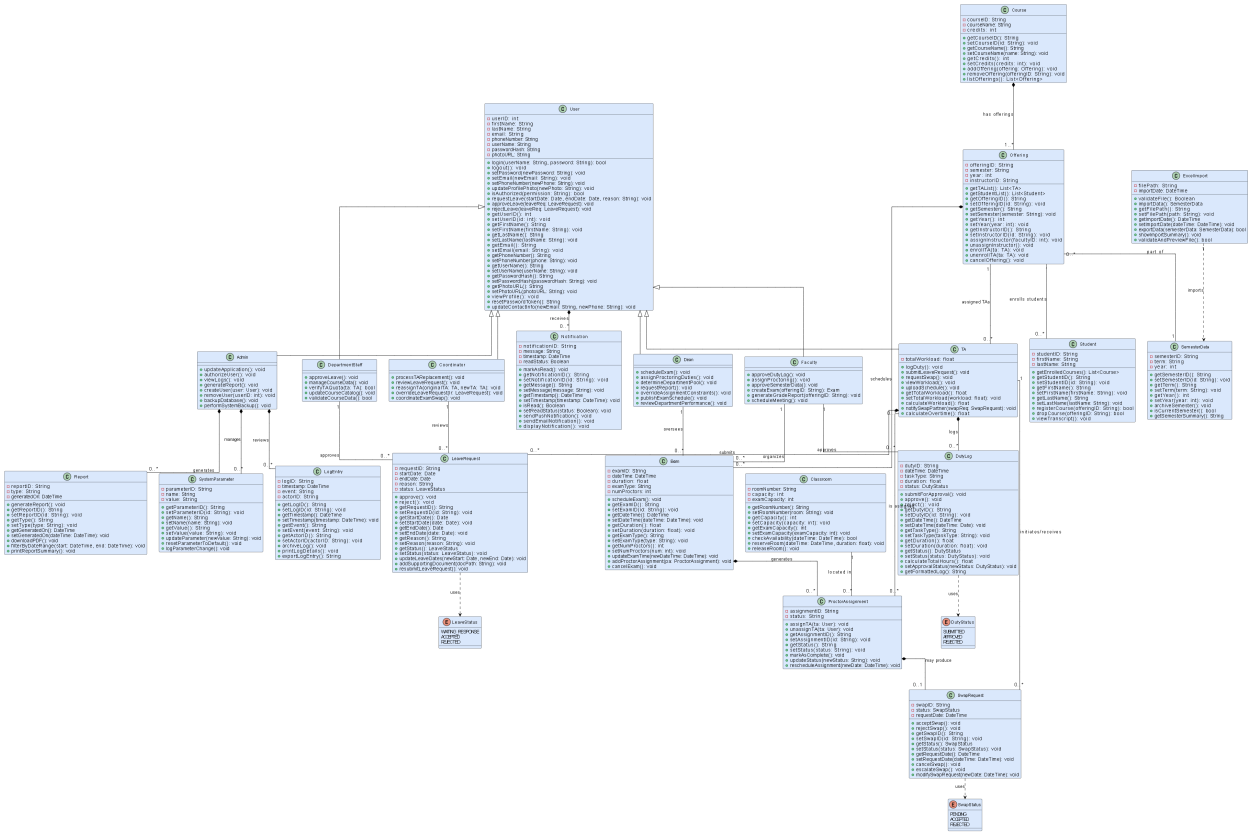
CS319 TA Management System
Deliverable 4

Team 9
Bilkent University - 2024/2025 Spring

Contents

A Class Diagram

The following figure presents the class diagram for the TA Management System.



B Software Design Patterns Used

B.1 Strategy Design Pattern for Proctor Assignment

The **Strategy Design Pattern** is used in our TA Management System project to dynamically handle TA proctor assignments based on the selected assignment mode. Although the strategy implementations reside within a single service class, they reflect the interchangeable behaviors defined by this pattern.

The strategy logic is encapsulated within the `FacultyMemberServiceImpl` class, particularly in the following method:

```
public void assignProctor(Long examId, AssignmentType mode, Long taId)
```

This method uses the `AssignmentType` enumeration (with values like `AUTOMATIC_ASSIGNMENT` and `MANUAL_ASSIGNMENT`) to determine which internal strategy method to call. The control flow is:

- `assignAutomatically(examId)` — selects and assigns a TA based on availability and workload.
- `assignManually(examId, taId)` — assigns a specific TA selected by the faculty member.

This approach aligns with the Strategy Pattern by encapsulating algorithms (strategies) as independent methods and selecting one based on input at runtime.

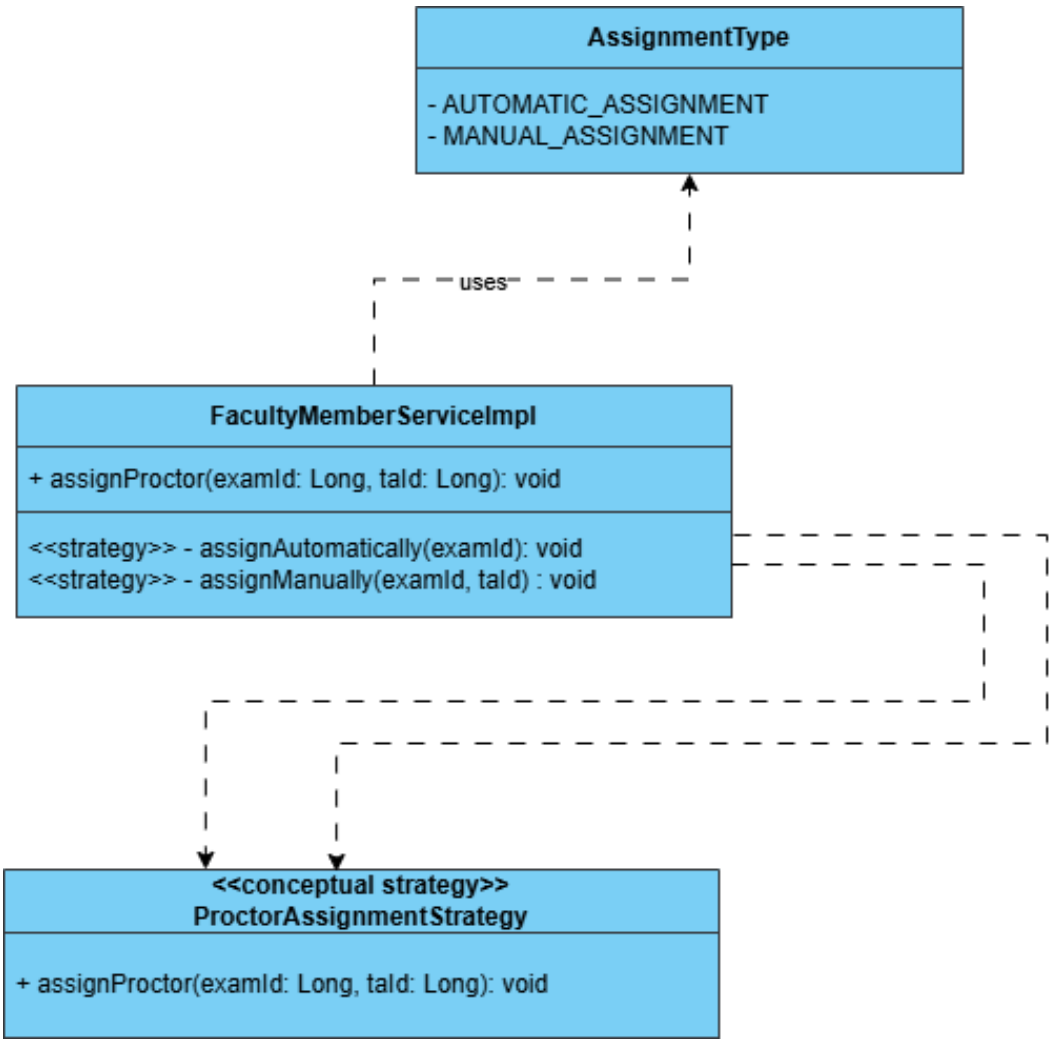


Figure 1: UML Diagram of Strategy Pattern for Proctor Assignment in FacultyMemberServiceImpl

Justification:

- The method signatures clearly separate strategy logic while keeping them interchangeable.
- The `AssignmentType` enum acts as a strategy selector at runtime.

Benefits:

- Facilitates easy addition of new strategies (e.g., random or priority-based assignment).
- Simplifies maintenance by isolating strategy logic.
- Conforms to the Open/Closed Principle — new strategies can be added with minimal change.

B.2 Observer Design Pattern for Notifications

The **Observer Design Pattern** is applied in the TA Management System to enable decoupled and extensible notification mechanisms when system state changes. This is used in the `FacultyMemberServiceImpl` class, particularly in methods like `approveLeaveRequest`, `rejectLeaveRequest`, `uploadDutyLog`, and `reviewDutyLog`.

Each of these methods results in user notifications, which are delivered through different channels (in-app alerts and/or email). This follows the classic observer model where multiple observers (notification services) respond to a subject's state change (e.g., duty approved).

Example – Leave Request Rejection:

When a leave request is rejected via:

```
public LeaveRequest rejectLeaveRequest(Long requestId)
```

the system notifies the TA through:

- `notificationService.notifyUser(...)` – for in-app notification.
- `mailSender.send(...)` – for email notification.

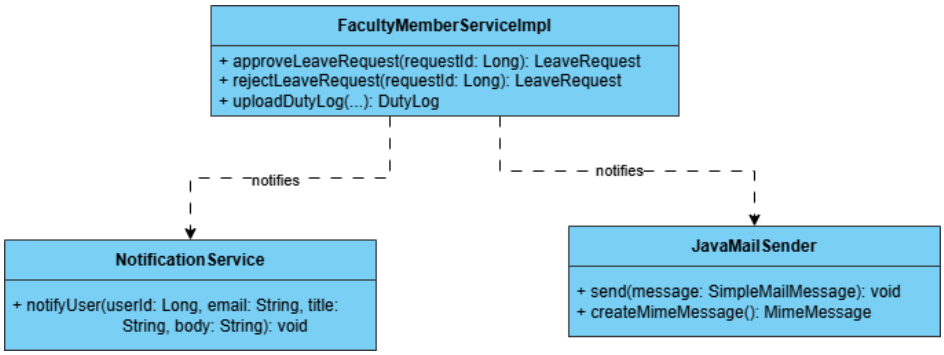


Figure 2: Observer Pattern in FacultyMemberServiceImpl for Notification Mechanism

Roles:

- **Subject:** Business logic methods (e.g., leave approval, duty upload).
- **Observers:** `notificationService` and `mailSender` respond to the subject's state changes.

Benefits:

- Promotes loose coupling between business logic and notification mechanisms.
- Allows extensibility — new notification channels (e.g., SMS, push) can be added without modifying core logic.
- Encourages separation of concerns.

B.3 Singleton Design Pattern for TA Class

The **Singleton Design Pattern** is inherently applied to service and component classes in the TA Management System through Spring Boot's default bean scope. This includes the `TAServiceImpl`, which handles all TA-related business logic.

Under the hood, Spring ensures that only a single instance of each service class is created and shared throughout the application. This is particularly relevant for the `TAServiceImpl` class, which is used repeatedly by controllers and other services.

Example Usage:

```
@Service
@RequiredArgsConstructor
public class TAServiceImpl implements TAService {
    ...
}
```

When Spring initializes this service with the `@Service` annotation, it maintains exactly one instance of `TAServiceImpl` (singleton) across the application context.

Benefits:

- Ensures consistent access to TA-related logic across the application.
- Reduces memory footprint by preventing redundant service instances.
- Simplifies dependency injection and promotes efficient resource management.

This Spring-managed Singleton Pattern is a core design convention that ensures clean architecture and efficient resource use.