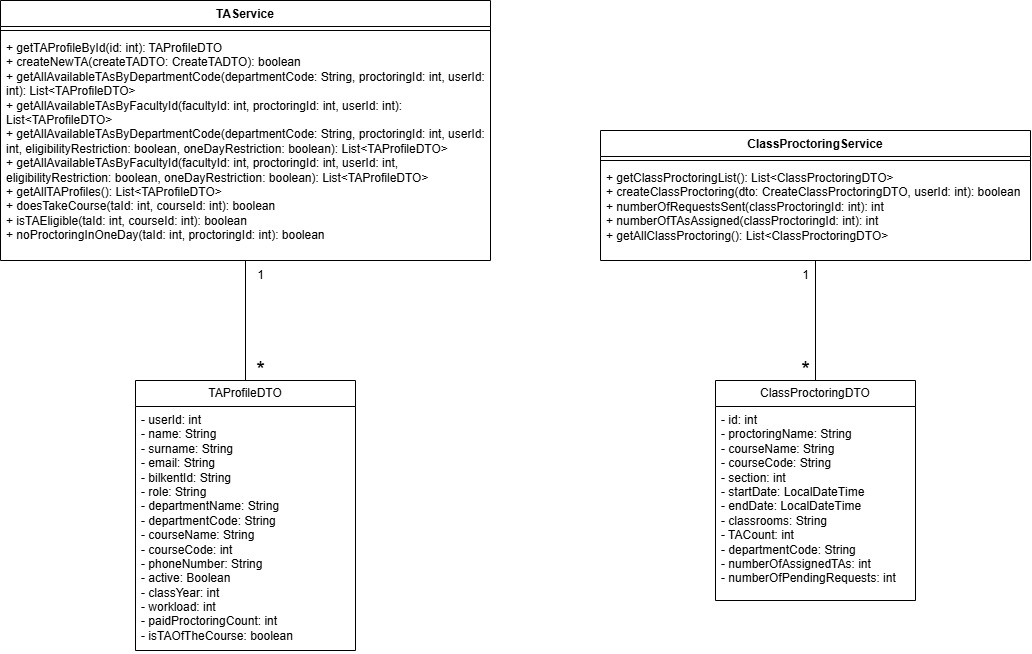
**B) Software Design Patterns Used**



**Data Transfer Object (DTO) Design Pattern**  
  
The Data Transfer Object (DTO) design pattern is a straightforward and highly effective architectural strategy that significantly enhances the structure and maintainability of software applications. By acting as an intermediary data carrier between layers, DTOs help enforce clear separation of concepts. This pattern is especially valuable in applications that demand strong decoupling between layers, optimized performance through minimized data exchange, robust input validation, and enhanced security by exposing only necessary information. Its simplicity, combined with its ability to support scalable and clean application architecture, makes it an essential tool in modern software development.

**Purpose and Application in Our Project**  
DTOs play a critical role in managing the structured and secure exchange of data involving complex workflows between TAs, Instructors, Deans Office and Department Secretary. These workflows include processes like availability updates, proctoring management, workload assignments and more in terms of managing TAs.

Using DTOs reduce complexity since it allows us to simplify big chunks of data into smaller, easier to work DTOs. This works in favor for both backend and front end. It makes the data and the inputs easier to manage and maintain while fetching or posting. Furthermore, it increases performance by sending only the required data.  
  
DTOs enable us to:  
- Encapsulate and validate input/output data clearly and consistently.  
- Shield internal data models from overexposure  
- Define explicit API contracts to ensure well-documented and predictable interfaces.  
- Support scalability and flexibility without impacting core business logic.  
- Promote modularity by minimizing tight coupling between architectural layers.  
  
**Benefits of the DTO Design Pattern:**  
1. Security – Protects sensitive internal data from exposure.  
2. Validation – Supports pre-processing and filtering of input data.  
3. Flexibility – Enables different representations of the same domain model  
4. Efficiency – Ensures lightweight data transmission by carrying only essential fields.  
5. Decoupling – Enhances maintainability and scalability by isolating domain logic from user interfaces.  
  
DTOs are indispensable in our system, which requires high data integrity, robust API design, and efficient communication across diverse roles and services. They contribute to the system's clarity, testability, and long-term scalability and maintenance. It is used to encapsulate data and transfer it efficiently between the different layers of the application, such as the controller, service, and repository layers. DTOs are simple, serializable objects that carry data without any business logic, making them ideal for decoupling internal representations from external interfaces.

In the context of the Bilkent TA & Proctor Management System, we manage a variety of data interactions involving different user roles (TAs, Instructors, Admin, Department Secretary, Deans Office) and operational data (e.g., Availability, Courses, Proctor Assignments). That is because in our project we have lots of more of DTOs other than the ones showed in the example.

**Strategy Design Pattern**

The Strategy Design Pattern is a behavioral design pattern that enables the selection of an algorithm’s behavior at runtime, providing a powerful mechanism for extending functionality without modifying existing code. It achieves this by defining a family of interchangeable algorithms, encapsulating each one within its own class, and ensuring they all adhere to a common interface. This separation of concerns allows clients to choose or switch between algorithms dynamically by decoupling the implementation of behaviors from the objects that utilize them, the Strategy pattern not only simplifies maintenance and testing but also enhances scalability in complex systems requiring role-specific logic or configurable workflows.

**Purpose and Application in Our Project**

In the Bilkent TA & Proctor Management System, we face situations where the system must support multiple implementations of a common behavior. This includes:

Different proctor assignment strategies (e.g., based on availability, load balancing, or preference ranking). The most obvious example for this is that one can assign a ta forcefully or request them to do a proctoring assignment. Another example for that would be request system. Most of the requests are handled by accepting or rejecting, but this is not the case for all, since the needs for action might change for different requests they might have similar but slightly different implementations.

Varying schedule generation rules, each TA can have different schedule items which should not conflict with one another. (leave of absence, taken course schedule, proctoring assignment , etc.)

Role-specific actions are important as well, even though 2 different actors seem to be doing a similar thing their methods and way of doing that might not be the same. In this context, take Department Secretary, Deans Office and Instructors’ exam assignments into consideration.

To accommodate these variations cleanly, we use the Strategy pattern to define an interface for the behavior and implement multiple interchangeable strategies. This allows us to better implement nearly same tasks and be more flexible with our code

**Benefits of the Strategy Pattern**

Flexibility – Easily swap or configure behaviors at runtime.

Add new strategies without changing existing code.

Decoupling – Keeps algorithms separate from the core logic, improving maintainability.

Testability – Each strategy can be independently tested.

Extensibility – Supports role-based behavior, custom workflows, and dynamic configuration.

**Strategic Role in Our Architecture**

In our project, the Strategy pattern is crucial for supporting configurable, reusable, and testable behaviors. It isolates logic that is likely to change (e.g., rules for assigning proctors or generating schedules) from the rest of the system, ensuring that feature updates or new policies can be added with minimal impact on the codebase.

This design pattern reinforces our system’s goals of modularity, flexibility, and role-aware logic, making it easier to evolve in response to new academic rules or user requirements.