CS 319 – Object Oriented Software Engineering

Deliverable 4

Spring 2025

Team 12

Ahmet Kenan Ataman <22203434>

Berfin Örtülü <21802704>

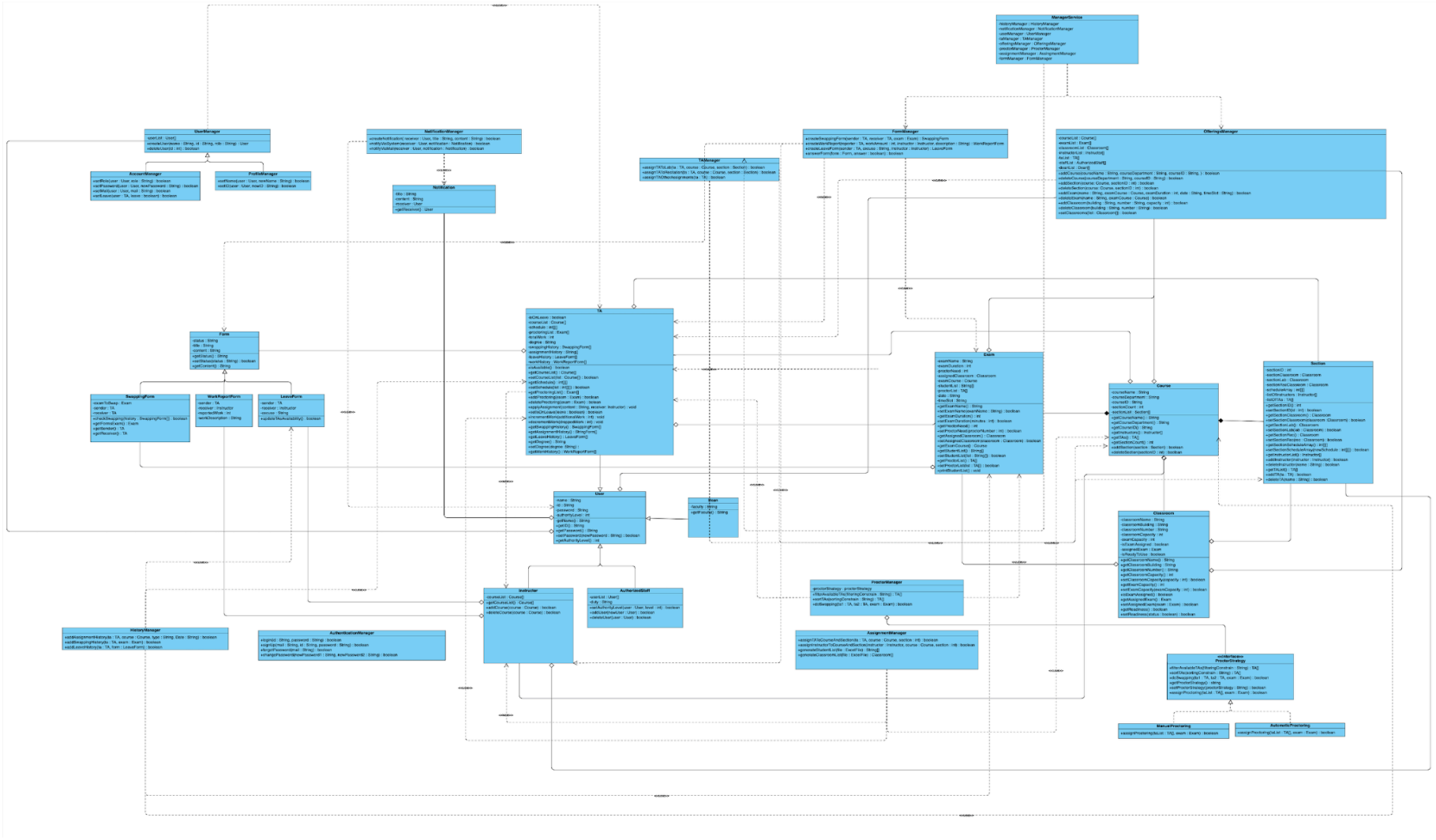
Erdem Uğurlu <22203391>

Gülferiz Bayar <21901442>

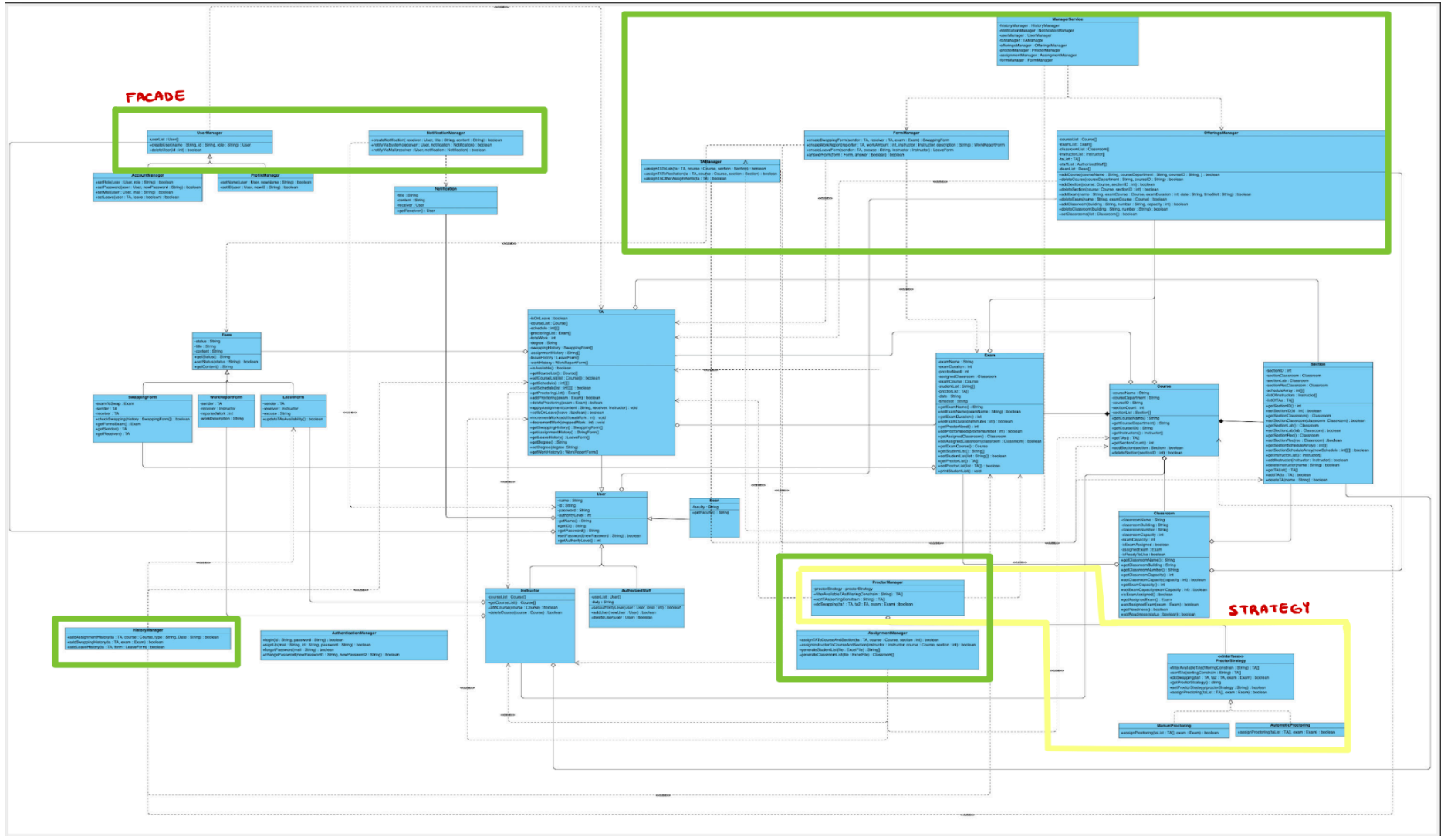
Mehmet Emre Şahin <22201765>

Table of Contents

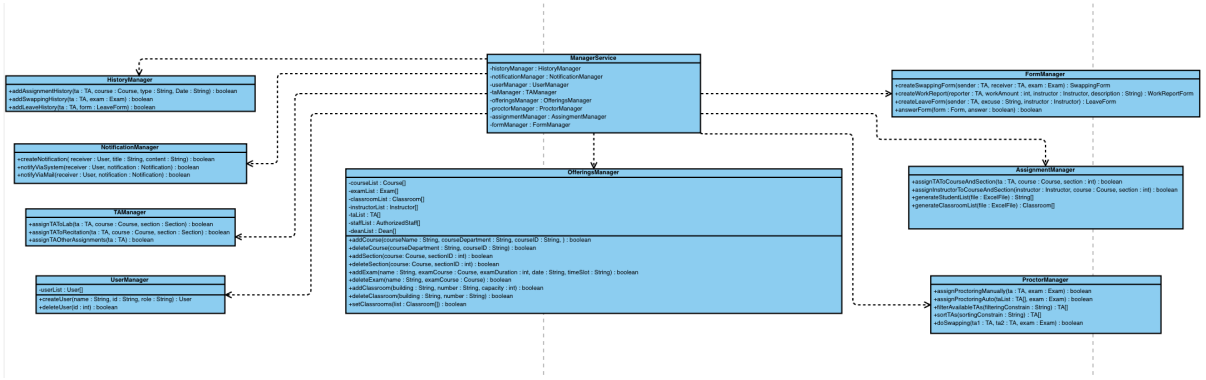
Class Diagram



Class Diagram (Class Diagram with Highlighted Design Patterns)



1. Facade Pattern



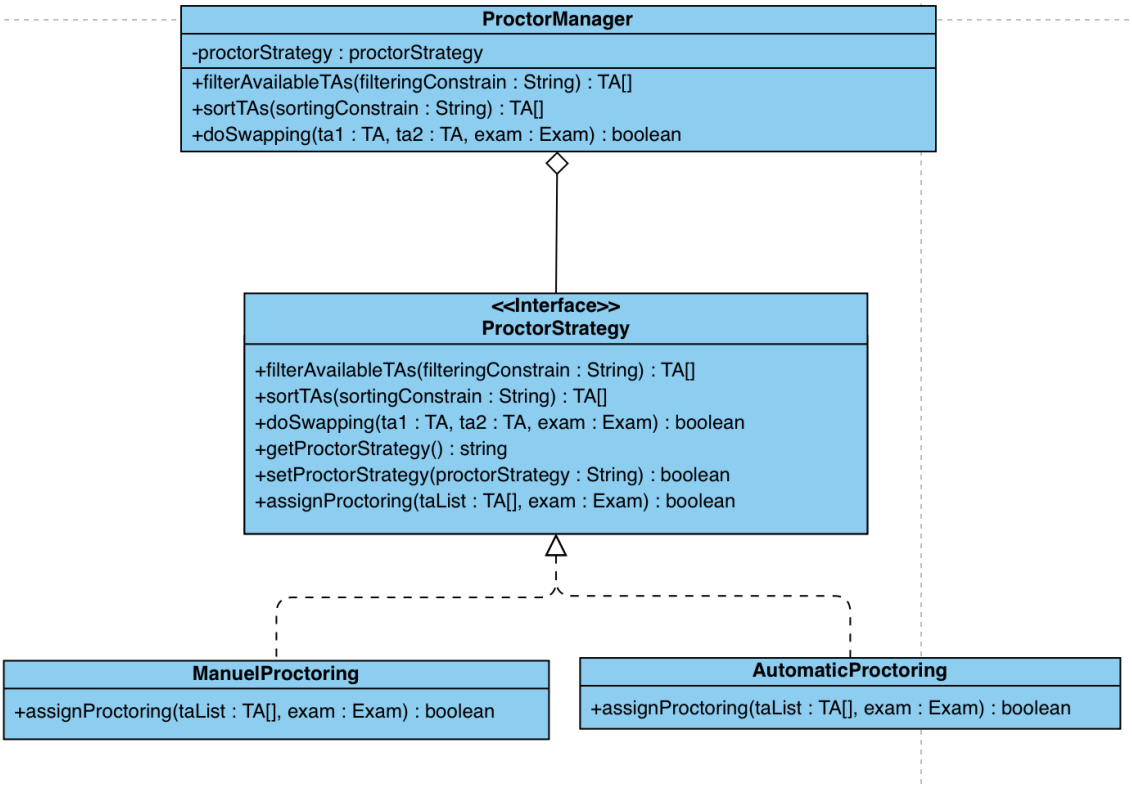
In our project, we have a variety of “Manager” classes (such as UserManager, NotificationManager, HistoryManager, AssignmentManager, ProctorManager, TAManager, OfferingsManager, and FormManager), each responsible for a specific piece of business logic. Rather than instantiating and wiring all of these managers individually in every controller or service, we encapsulate them behind a single ManagerService class that acts as a Facade.

The ManagerService holds direct references to each manager and exposes high-level, unified methods (for example, createUserAndNotify() or importAndAssign()) that internally delegate tasks to the appropriate managers in the correct sequence. This approach allows external code to interact with a single, simple interface instead of juggling multiple classes and their complex interactions.

The UML diagram illustrates this clearly: ManagerService is the facade that aggregates the various subsystem managers through composition (shown as solid lines in the diagram). All internal complexity is hidden behind this unified entry point. By calling one facade method, our controllers stay lean and focused, the system remains loosely coupled, and adding or changing workflow steps becomes far simpler and less error-prone.

This design ensures that any modifications to the internal manager logic do not ripple through the entire system, as the ManagerService provides a stable and clean API for external consumers.

2. Strategy Pattern



In our exam-management system, proctoring assignment represents a flexible “variation point” where the behavior depends on the selected policy, such as manual or automatic assignment. To implement this flexibility cleanly, we define an interface named ProctorStrategy. This interface declares methods for common proctoring operations like filtering available teaching assistants, sorting them, performing swapping tasks, and most importantly, assigning proctors via the assignProctoring(taList, exam) method. These methods provide a unified contract that ensures all concrete strategies implement the expected behavior.

We implement two concrete classes: ManuelProctoring and AutomaticProctoring. Both classes implement the ProctorStrategy interface and provide their own versions of the assignProctoring method, each offering a specific proctoring logic (manual or automatic). This allows each strategy to encapsulate its unique proctoring behavior while adhering to the shared interface.

At runtime, the ProctorManager class holds a composition relationship with a ProctorStrategy instance, meaning it directly owns and manages the strategy object (represented by the filled diamond in the UML diagram). The ProctorManager delegates all proctoring-related operations to its current strategy implementation via the interface. For instance, when a proctoring task needs to be performed, ProctorManager calls assignProctoring(taList, exam) on its strategy object without knowing or depending on whether the logic is manual or automatic.

The UML diagram below illustrates this design: ProctorStrategy is an interface, and ManuelProctoring and AutomaticProctoring implement this interface. This is shown by dashed lines with open triangles, indicating interface realization rather than standard class inheritance. The ProctorManager maintains a composition relationship with the strategy, shown by a solid line with a filled diamond, emphasizing that it owns the strategy instance directly.

This design keeps the ProctorManager decoupled from specific proctoring logic, ensuring that new strategies can be introduced easily without any changes to the manager's core code. By overriding the assignProctoring method in each concrete strategy, we ensure that the correct proctoring logic is executed based on the active strategy, while maintaining a clean and modular architecture.