TA Management System

Deliverable-4
11.05.2025

Course: CS 319

Section: 01

Group No.: 7

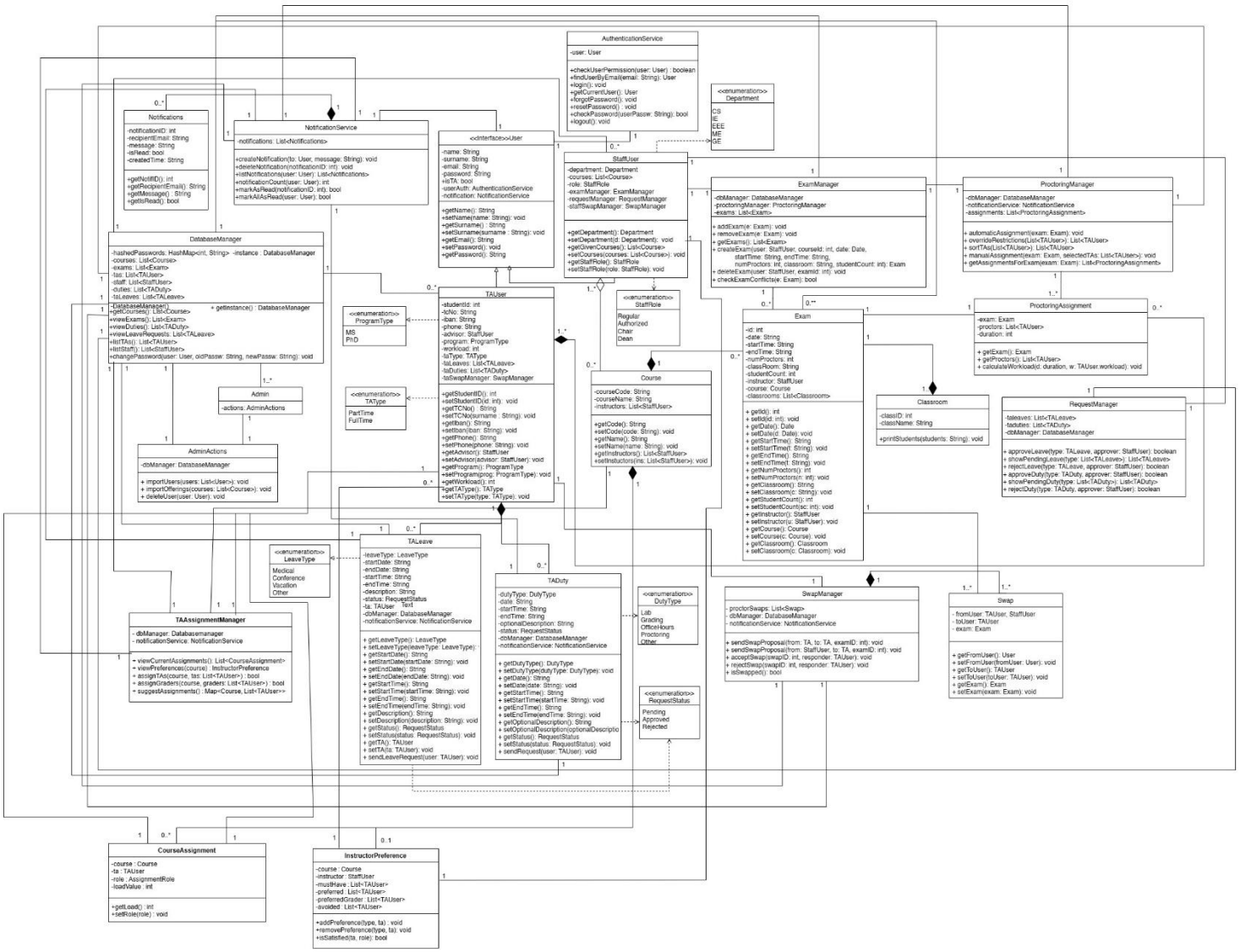
Instructor: Eray Tüzün

Teaching Assistant: Utku Boran Torun

Group Members: Bilge İdil Öziş [22102365]
Burak Kağan Gür [22203365]
Daib Malik [22201013]
Yunus Günay [22203758]

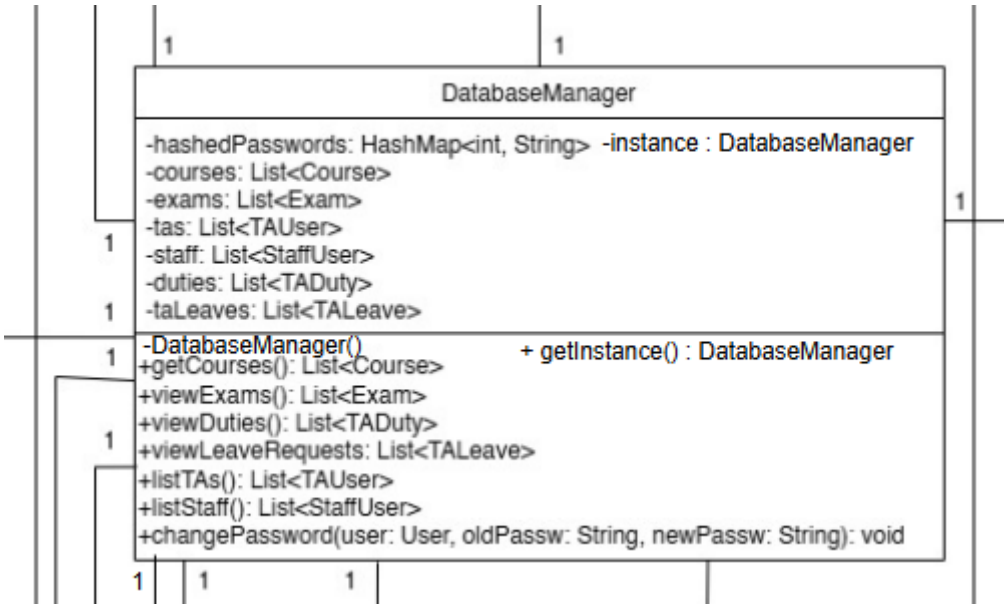
Contents

1. Class Diagram:



2. Software Design Patterns Used:

a. Singleton Design Pattern:



The singleton design pattern is used to make sure there is only one instance at a time of DatabaseManager while holding global access to other classes.

The singleton design pattern is represented in our system in the following way:

Singleton: DatabaseManager class represents the singleton class that has only one instance in the system. This can be seen through the fact that multiplicities of it is 1 at every association end, no other class creates a second DatabaseManager.

Clients: ExamManager, ProctoringManager, SwapManager, RequestManager, AuthenticationService, NotificationService, TAAssignmentManager classes are the clients of DatabaseManager class. Instead of creating multiple instances, each client accesses the shared DatabaseManager by calling DatabaseManager.getInstance().

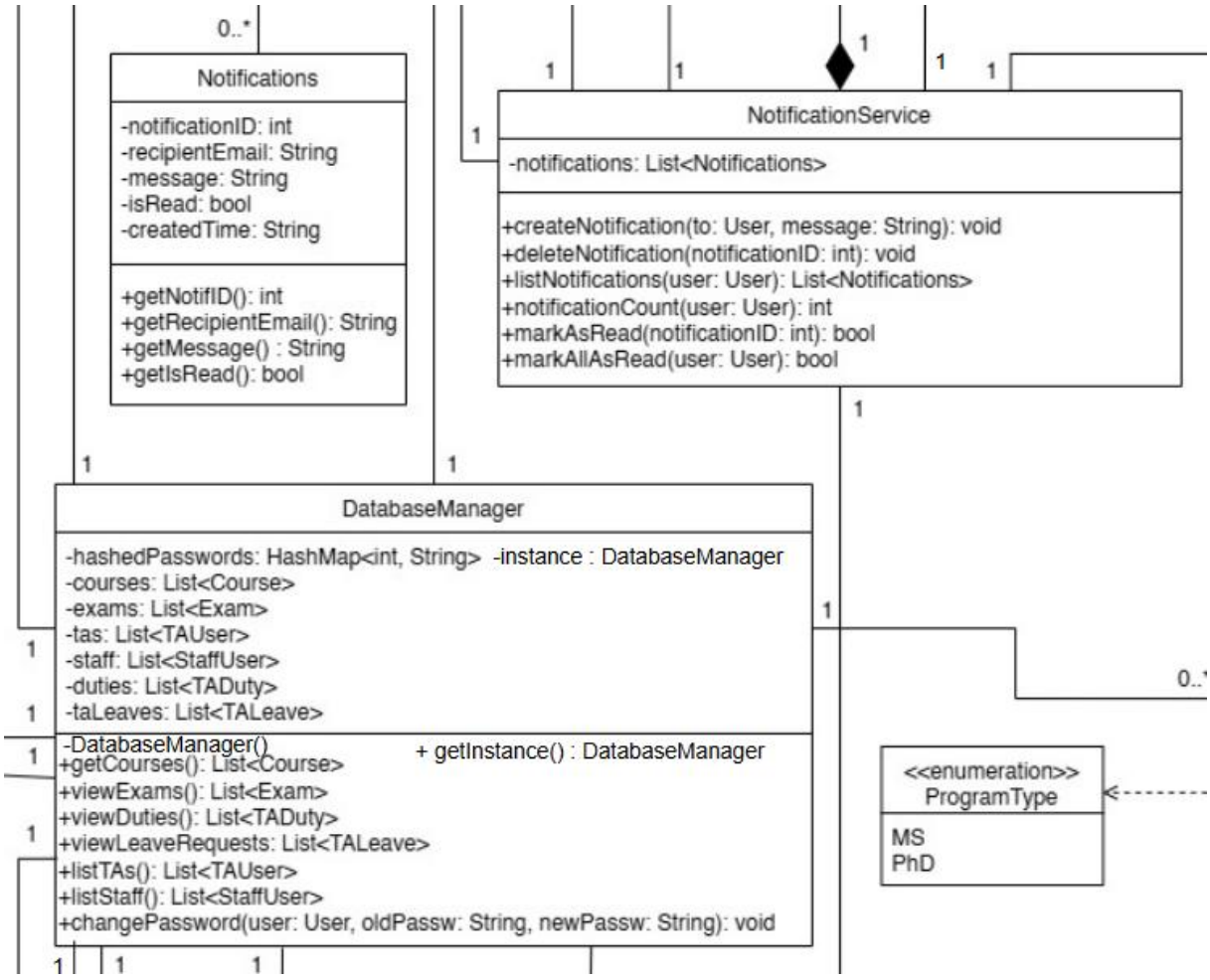
The benefits that were offered by using this design pattern:

- **Centralized resource control:** In any system, creating multiple database connections or data managers can be expensive and unnecessary. The Singleton pattern allows us to maintain a single, shared DatabaseManager instance

that all parts of the application use. This prevents duplication of heavy resources and ensures that there is only one point of access to the underlying data operations.

- **Global consistency:** By having only one DatabaseManager, all the domain manager classes operate on the same dataset and persistence context. This eliminates the risk of accidental data mismatches or synchronization errors that could happen if two or more independent instances existed.
- **Ease of access:** The Singleton pattern simplifies access by allowing any class to retrieve the single instance of DatabaseManager in a single line of code. This makes it very convenient to integrate with the various domain managers without adding extra complexity.

b. Façade Design Pattern:



The façade design pattern is used to provide a simpler interface to a set of subsystems.

We have used this design pattern in many places in the code; the above picture demonstrates one of them.

The façade classes in our system:

AuthenticationService: This class provides a unified interface for the subsystem that consists of the classes: User, TAUser, StaffUser, and DatabaseManager

ExamManager: This class provides a unified interface for the subsystem that consists of the classes: Exam, Course, Classroom, TAUser, and DatabaseManager

ProctoringManager: This class provides a unified interface for the subsystem that consists of the classes: ProctoringAssignment, Exam, TAUser, StaffUser, NotificationService, and DatabaseManager

SwapManager: This class provides a unified interface for the subsystem that consists of the classes: Swap, Exam, TAUser, NotificationService, and DatabaseManager

RequestManager: This class provides a unified interface for the subsystem that consists of the classes: TALeave, TADuty, TAUser, StaffUser, NotificationService, and DatabaseManager

NotificationService: This class provides a unified interface for the subsystem that consists of the classes: Notification, User, and DatabaseManager

These façade classes provide the clients (the other classes that call them such as user interface classes) with a higher-level interface so they do not need to call the subsystems that façade classes use. This way we have a single entry point. The client code (REST controller / CLI) talks only to the façade, never to Exam or Swap directly.

The benefits that were offered by using this design pattern:

- **Reduced complexity for developers:** The Facade pattern simplifies the way external layers interact with the system.

Instead of requiring external components to coordinate multiple domain objects directly, a single method call to the Facade class handles the complete operation. For example, scheduling a swap or assigning a TA requires only one interaction with the Facade.

- **Low coupling between presentation and application layers:** By acting as an intermediary, the Facade isolates the presentation layer (the user interface) from the underlying domain rules and data operations. This separation ensures that changes to internal logic, such as modifying the database schema or improving the exam scheduling algorithm, only affect the Facade without impacting external components.