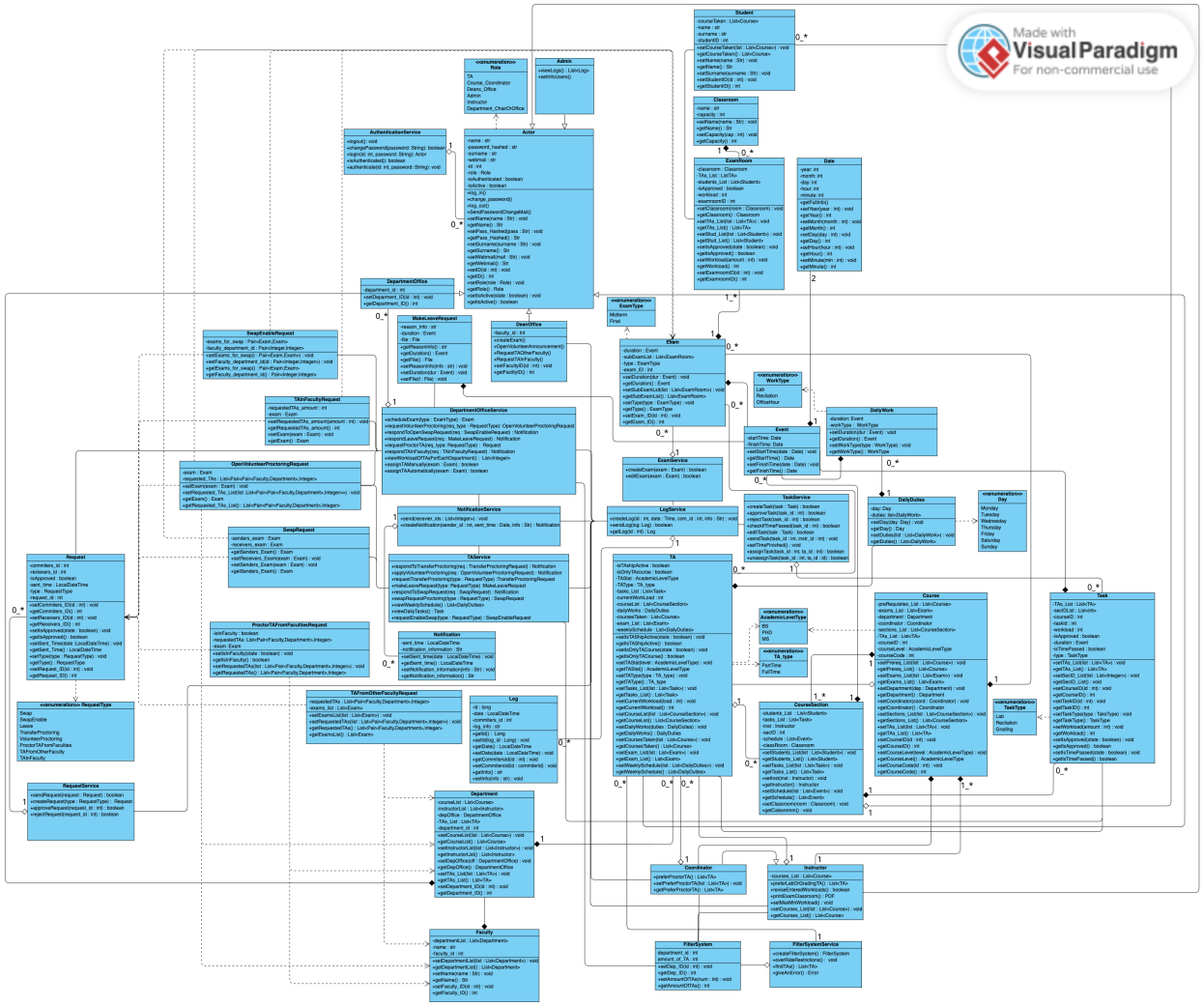
BILKENT UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
CS 319 OBJECT-ORIENTED SOFTWARE ENGINEERING
DELIVERABLE 4
SPRING 2025
GROUP 2 SECTION 1
11.05.2025

Full Name	ID
Perhat Amanlyyev	22201007
Emiralp İlgen	22203114
Anıl Keskin	22201915
İlmay Taş	22201715
Simay Uygur	22203328

Table of Contents

A. Class Diagram

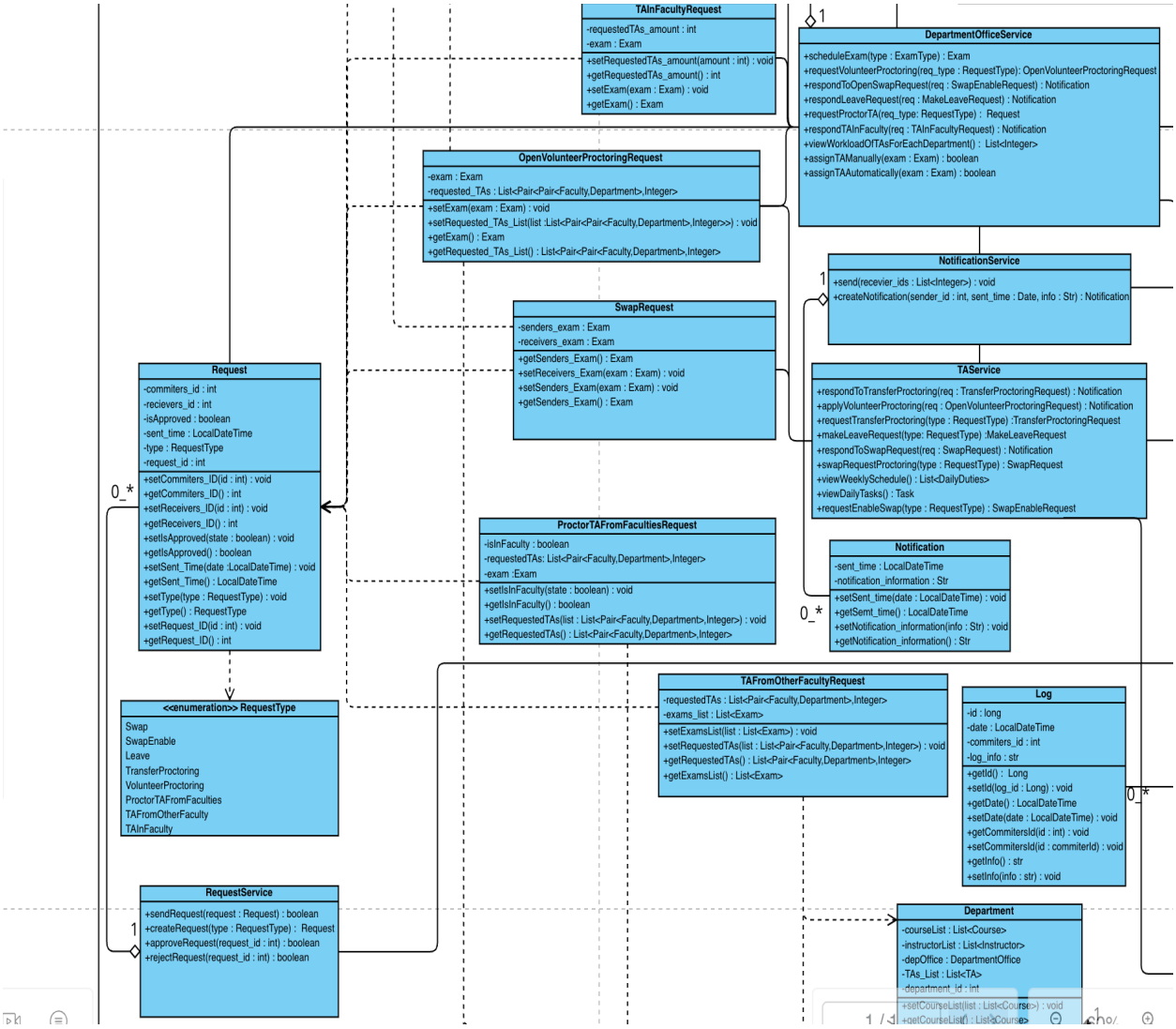


The link for diagram: https://online.visual-paradigm.com/share.jsp?id=333439313030372d3131#diagram:workspace=patxxmja&proj=0&id=11

Drive Link: https://docs.google.com/document/d/1AODbbnLRNEepg2INmynzSk_cDxklM_vusJ21ETte558/edit?usp=sharing

B. Software Design Patterns Used

1. Command Design Pattern for Handling Requests



1. Overview

The **Command Design Pattern** is employed in the TA Management System to encapsulate various request types—such as SwapRequest, OpenVolunteerProctoringRequest, and ProctorTAFromFacultiesRequest—as distinct command objects. This design enables decoupling of the request invocation from its execution logic, thereby enhancing modularity, extensibility, and maintainability of the overall system.

2. Pattern Components and Mapping to Class Diagram

2.1. Command Interface – Request (Abstract Class)

The abstract class Request serves as the base type for all command objects. It defines a uniform structure by providing shared attributes and method contracts, including:

- Fields: requestId, sentTime, isApproved, type
- Methods: setType(), getRequest_ID(), setSentTime(), setIsApproved(), etc.

All concrete command types extend this class, ensuring consistent handling and polymorphism.

2.2. Concrete Commands

Each subclass of Request defines a specific operation and encapsulates the associated data:

- SwapRequest: Contains fields such as senders, receivers, and exams.
- OpenVolunteerProctoringRequest: Maintains a list of TAs and exams relevant to volunteer proctoring.
- ProctorTAFromFacultiesRequest and TAFromOtherFacultyRequest: Handle complex mappings between faculties, departments, and exams.

These classes represent discrete, self-contained actions that can be executed independently.

2.3. Invoker – RequestService

The RequestService class is the invoker that manages the execution lifecycle of command objects. It provides methods such as:

- sendRequest(Request request): boolean
- approveRequest(Request request): boolean
- rejectRequest(Request request): boolean

This service delegates command execution to appropriate receivers based on the command type. It performs validation, handles persistence, and ensures separation between command initiation and execution.

2.4. Receivers – TAService, DepartmentOfficeService

Receivers contain the domain-specific logic required to handle each request:

- TAService: Implements methods such as respondToSwapRequest() and respondToTransferProctoringRequest().
- DepartmentOfficeService: Handles volunteer proctoring and TA assignment logic through methods like respondToVolunteerProctoringRequest().

Each receiver operates independently and interacts with the command objects passed through the invoker.

3. Execution Flow Example

A typical request lifecycle can be described as follows:

1. A REST controller receives a command via @RequestBody, e.g., a SwapRequestCommand.
2. The controller forwards it to the invoker (RequestService.sendRequest()).
3. The RequestService validates the request and dispatches it to the appropriate service (TAService.respondToSwapRequest()).
4. The receiver executes the request and triggers any necessary notifications or database updates.

4. Benefits of the Pattern in This Context

Benefit	Description
Encapsulation	Each request type is encapsulated in a separate class, allowing modular design.
Separation of Concerns	Request creation and execution are clearly separated.
Extensibility	New request types can be added by extending the Request class without altering the existing system logic.
Auditability	The Log class records command metadata for traceability and debugging.

5. Spring Boot Adaptation

In the Spring Boot architecture, controllers serve as request entry points and invoke commands through service classes. For instance:

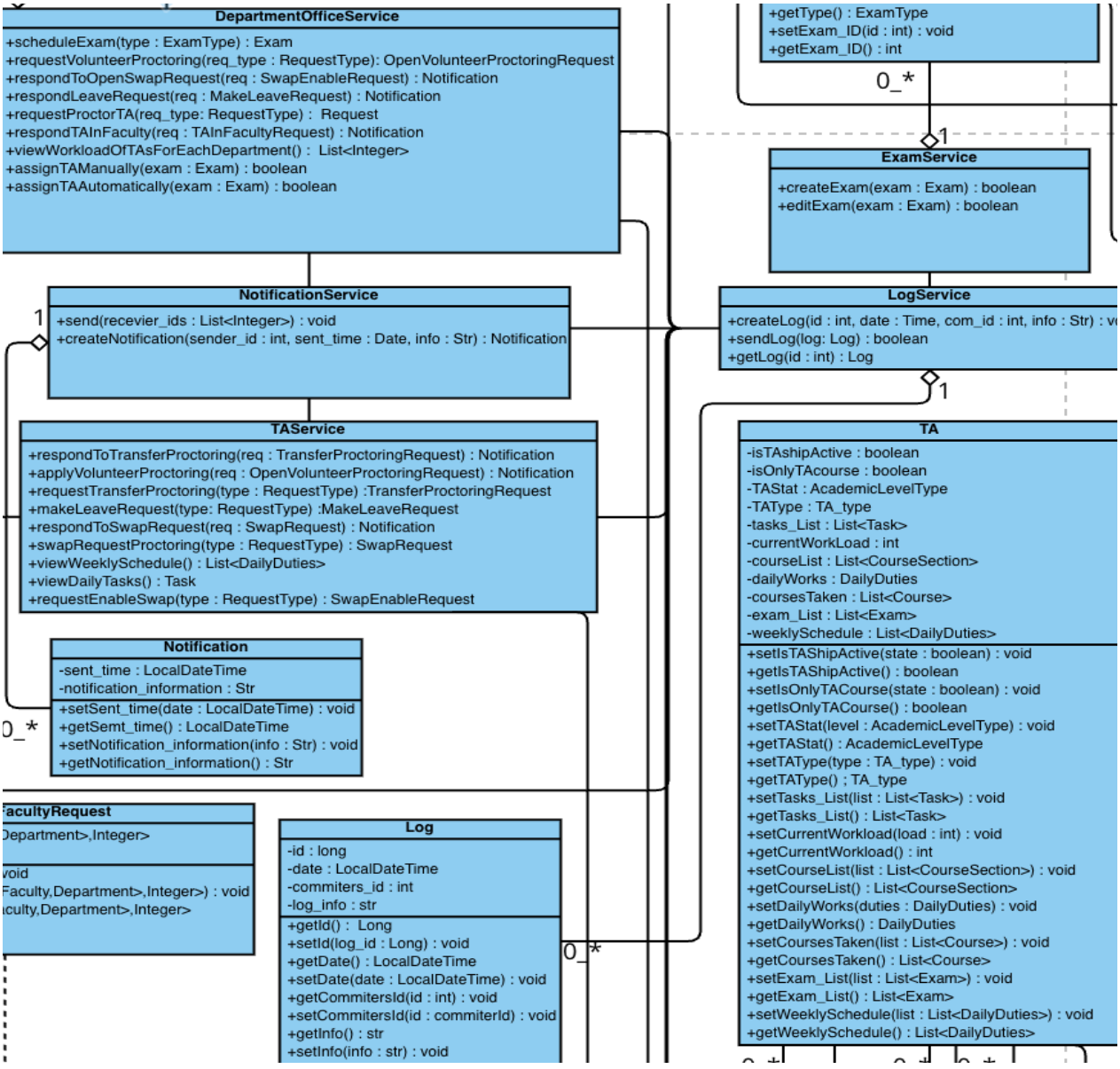
```
@PostMapping("/swap")
public ResponseEntity<?> createSwapRequest(@RequestBody SwapRequestCommand command) {
   requestService.sendRequest(command.toEntity());
   return ResponseEntity.ok().build();
}
```

- SwapRequestCommand is a DTO that maps to the domain model (SwapRequest).
- The RequestService processes the command as a polymorphic Request.

6. Summary of Component Roles

Design Pattern Role	Class(es)
Command Interface	Request (abstract class)
Concrete Commands	SwapRequest, OpenVolunteerProctoringRequest, TAFromOtherFacultyRequest, etc.
Invoker	RequestService
Receiver	TAService, DepartmentOfficeService
Logger (Optional)	Log (for command persistence and audit trail)

2. Singleton Design Pattern for Service Classes



.

1. Overview

The **Singleton Design Pattern** ensures that a class has only one instance throughout the application lifecycle and provides a global point of access to it. In the TA Management System, this pattern is conceptually applied to a set of **core service classes** responsible for orchestrating application-wide, stateless operations, such as TA assignment, exam scheduling, logging, and notification delivery.

2. Candidate Singleton Services

The following classes serve as centralized service layers and implicitly behave as singletons:

- `TAService`: Manages TA assignment workflows and request processing.
- `ExamService`: Handles exam-related operations and data retrieval.
- `LogService`: Responsible for recording system events and request metadata.
- `NotificationService`: Sends notifications to users upon state transitions or approvals.
- `DepartmentOfficeService`: Coordinates department-specific validation and assignment logic.

Although these classes do not explicitly implement the `getInstance()` method commonly found in traditional Singleton implementations, their usage aligns with Singleton principles due to the Spring Boot runtime behavior.

3. Singleton Pattern Realization in Spring Boot

In the context of the **Spring Boot framework**, Singleton behavior is inherently managed by the **Spring IoC (Inversion of Control) container**. Service classes annotated with `@Service` are automatically instantiated as **Singleton-scoped beans** by default.

```
@Service
public class TAService {
   public void respondToSwapRequest(SwapRequest request) {
      // Business logic
   }
}
```

- When the application starts, Spring creates a single instance of each `@Service` class.
- These instances are **injected** into other components (e.g., controllers, other services) via **constructor injection** or **field injection** using `@Autowired`.

```
@Autowired
private TAService taService;
```

This approach ensures **singleton semantics** without manually managing instance creation or synchronization.

5. Technical Justification for Singleton Usage

Aspect	Justification
State Management	Services are stateless and do not maintain user/session-specific data.
Consistency Guarantee	A single shared instance ensures uniform access and behavior across modules.
Performance Efficiency	Avoids repeated instantiation overhead by reusing a single instance.
Testability and Modularity	Allows mock injection during unit testing using Spring's dependency injection mechanisms.

5. Class Diagram Observation

Although the UML diagram does not explicitly show singleton constructs (e.g., private constructors or static `getInstance()` methods), the role and usage of the service classes are consistent with the Singleton design intent. Each service is injected into multiple classes (such as `RequestService`, controllers, or receivers) and is expected to function as a central coordination point for a specific domain concern.

6. Summary

Component	Pattern Role	Singleton Implementation Detail
`TAService`	Singleton Service	`@Service` + Spring-managed singleton instance
`NotificationService`	Singleton Service	Global notification dispatch logic, injected via DI
`LogService`	Singleton Service	Logging logic shared across all modules
`DepartmentOfficeService`	Singleton Service	Single instance manages departmental validation logic
`ExamService`	Singleton Service	Centralized source of truth for exam operations

By leveraging the Spring container's built-in support for Singleton beans, the system avoids manual singleton management while maintaining the benefits of the pattern in a clean, modular, and scalable architecture.