IHSAN DOGRAMACI BILKENT UNIVERSITY

CS319: Object-Oriented Software Engineering

Deliverable-5 Report

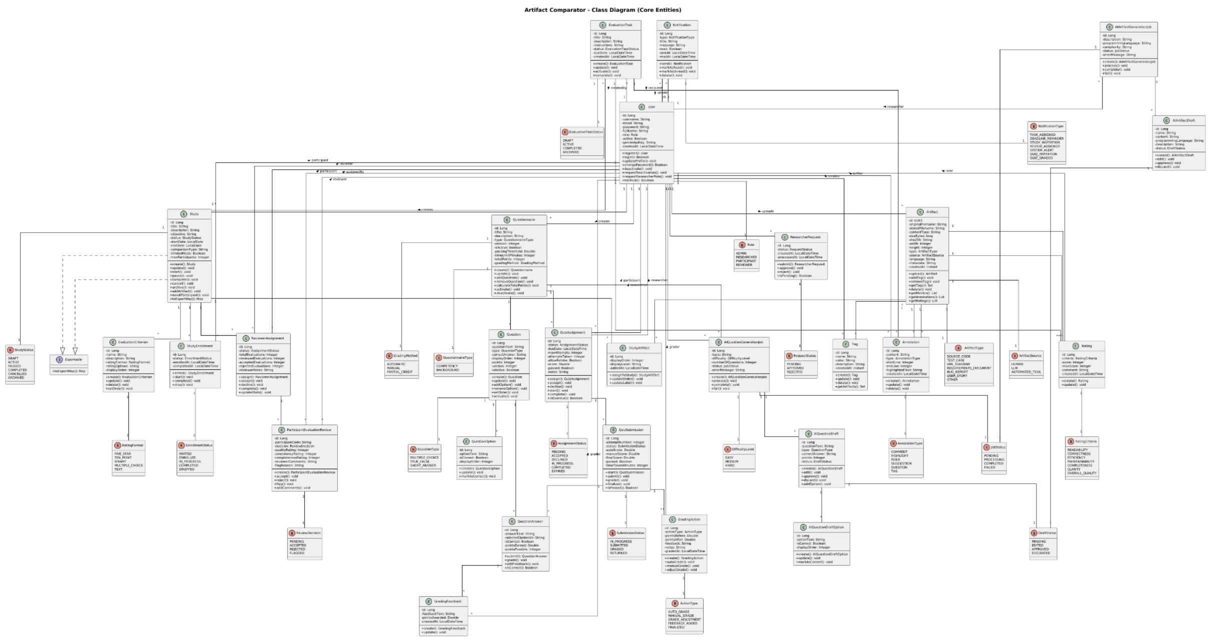
Group: Insight Coding

Group Members:	Mustafa Mert Gülhan	22201895
	Ece Bulut	22202662
	Yasemin Altun	22202739
	Altay İlker Yiğitel	22203024
	Efe Erdoğmuş	22203553

Instructor: Anıl Koyuncu

Submission Date: 5 December, 2025

1. Class Diagram



https://drive.google.com/file/d/16ZP45NpBnxBDr_Ss4I90evNCLGN00iDy/view?usp=sharing

2. Design Patterns

2.1 Facade Pattern

The Facade Pattern provides a simplified and unified interface to a set of complex subsystems, offering a unified approach to accessing these subsystems. Rather than exposing lower-level components such as repositories, HTTP clients, authentication logic, or JSON parsing utilities, a facade concentrates these responsibilities into a single high-level API. This allows the rest of the application to work with a clean, domain-focused interface without needing to understand how the underlying operations are performed. In practice, this reduces coupling, improves modularity, and makes the system far easier to extend or maintain.

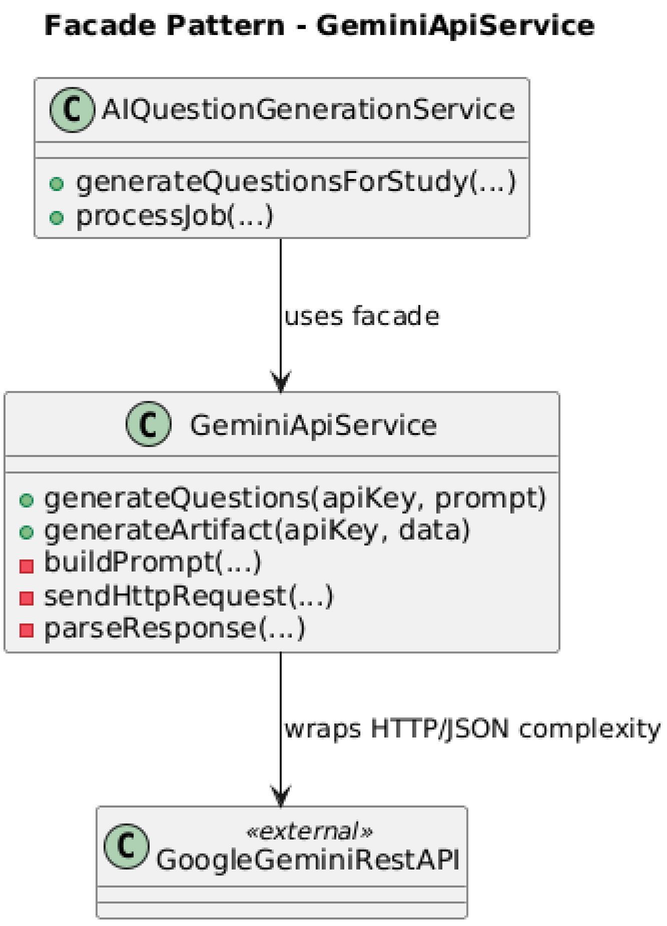
In our project, the Facade Pattern appears in both the backend and the frontend. On the backend, the **GeminiApiService** is a clear example of a facade. All communication with Google's Generative Language API is handled inside this service, including prompt construction, request body creation, header and API key management, JSON (de)serialization, and error handling. Other domain services, such as **AIQuestionGenerationService**, do not interact with HTTP libraries directly; they simply call high-level methods, such as **generateQuestions(...)** or **generateArtifact(...)**. The complexity of network communication is therefore fully encapsulated, and the domain logic operates only with properly structured Java models. This separation is precisely what the

Facade Pattern aims to achieve: a single, easy-to-use entry point to a complicated external subsystem.

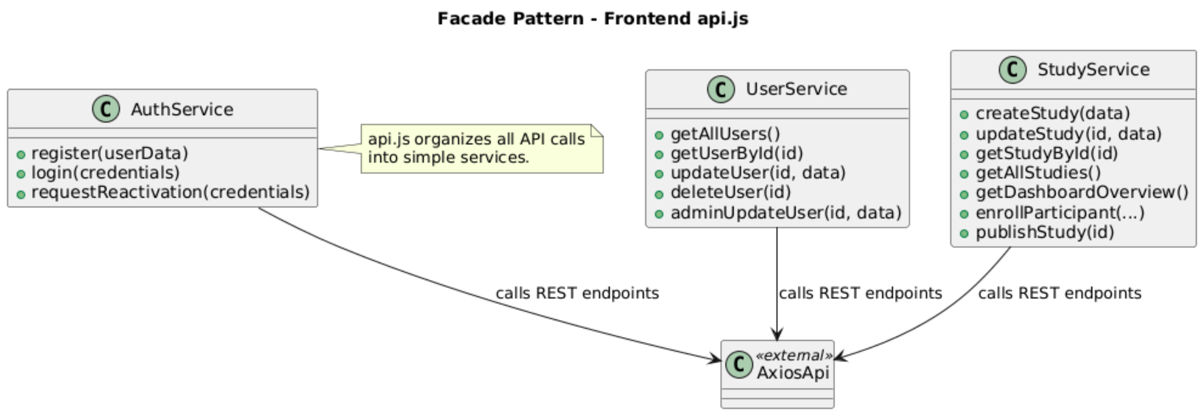
A similar use of the pattern appears on the frontend in the api.js module. This file acts as the application's central gateway for all HTTP communication between the client and the backend. Instead of scattering Axios configuration, token management, URL building, and error handling throughout the UI components, api.js exposes simple and consistent domain-specific methods such as **authService.login()**, **userService.getAllUsers()**, and **studyService.createStudy()**. Behind these method calls lie all of the networking details required to make reliable requests. The Facade Pattern is therefore used here to shield the React components from the complexity of HTTP operations and to provide a single clear interface for each domain module.

Overall, the Facade Pattern minimizes direct dependencies on external APIs and internal subsystems, centralizing any future changes to these processes. As a result, both the backend and frontend benefit from clearer structure, improved maintainability, and a cleaner separation between domain logic and underlying technical details.

2.1.1 Facade Pattern (GeminiApiService)


Facade Pattern - GeminiApiService
C AIQuestionGenerationService
generateQuestionsForStudy(...)
processJob(...)
uses facade
C GeminiApiService
generateQuestions(apiKey, prompt)
generateArtifact(apiKey, data)
buildPrompt(...)
sendHttpRequest(...)
parseResponse(...)
wraps HTTP/JSON complexity
«external»
C GoogleGeminiRestAPI

2.1.2 Facade Pattern (Frontend api.js)



2.2 Builder Pattern

The Builder Pattern is a creational design pattern that provides a structured and readable way to construct complex objects. Instead of using constructors with long parameter lists that are difficult to read, error-prone, and inflexible, the Builder Pattern allows objects to be created step by step through a fluent API. Each field is explicitly named during construction, which makes the resulting code significantly clearer and easier to maintain. In our project, the Builder Pattern is applied extensively through Lombok's **@Builder** annotation, which automatically generates the underlying builder classes and fluent setter methods.

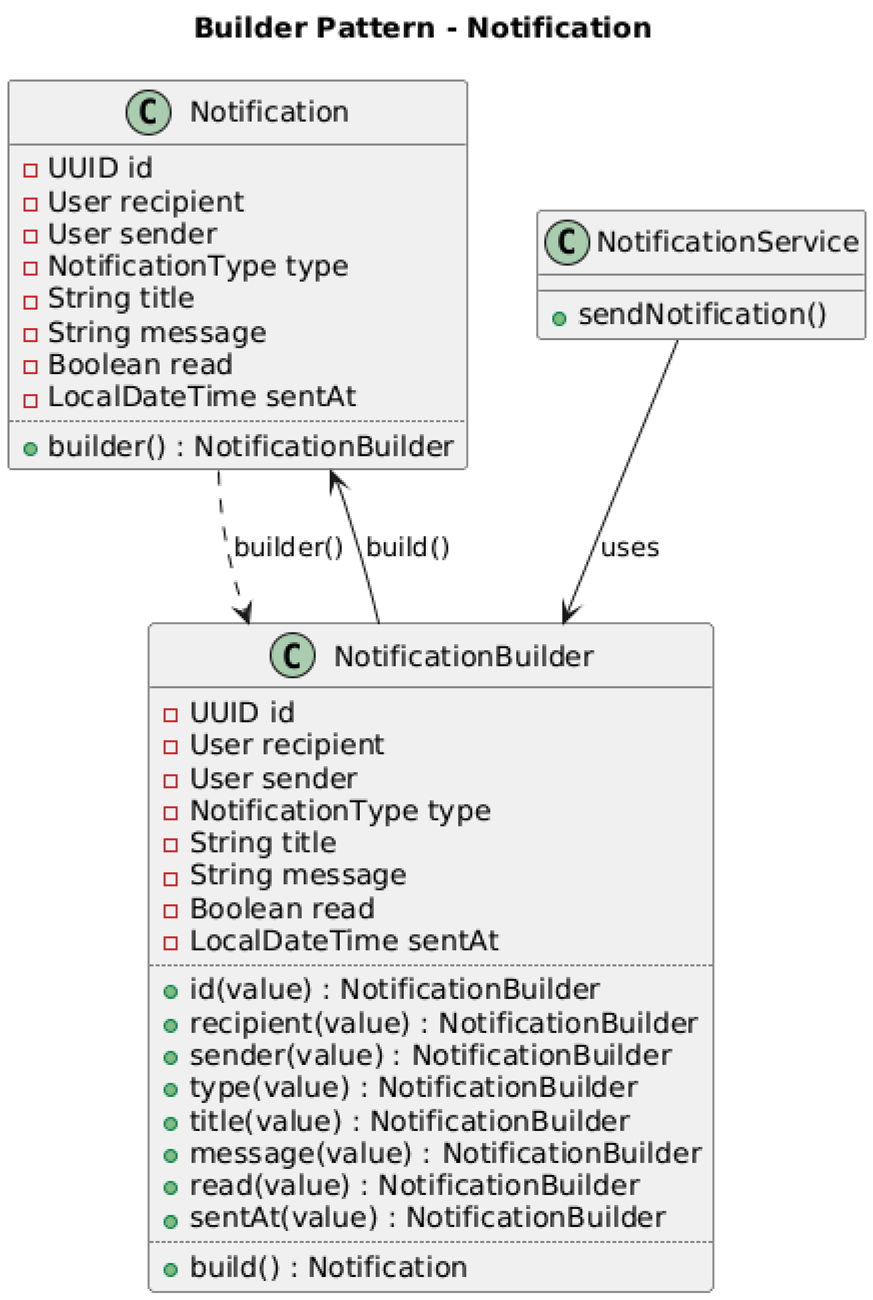
The pattern appears in many parts of the system, particularly in domain models and DTOs, where objects often carry numerous optional or context-dependent fields. Examples include **Notification**, **QuizSubmission**, **ReviewerAssignment**, and DTOs such as **NotificationDTO, EvaluationTaskDTO**, and **ParticipantAssignmentDTO**. These objects frequently require only a subset of their fields to be populated, depending on the use case, making a traditional constructor unsuitable. Builders allow developers to specify only the fields relevant in that context while preserving clarity and avoiding ambiguity. Services that convert between entities and DTOs also rely on builders to construct output objects in a controlled and readable manner.

One illustrative example is the creation of a **Notification** entity. With the Builder Pattern, the service can construct the object in a clear and expressive manner, specifying each field with fluent method calls such as .**recipient(...)**, **.sender(...)**, **.title(...)**, and .**message(...)**, before completing the process with **.build()**. This stands in contrast to calling a constructor with many positional arguments, where the meaning of each parameter is not immediately obvious,s and errors are more likely. The same benefits of clarity appear in DTO conversion, where builders enable the service layer to map each field individually and conditionally, resulting in clean and maintainable code, even when fields may be null or optional.

The Builder Pattern also contributes to robustness by enabling sensible defaults through Lombok's **@Builder.Default** mechanism. For instance, the **Notification** entity defines its **read** field as **false** by default, ensuring consistent object initialization without requiring every caller to set this value. Additionally, because objects are fully constructed only when **.build()** is invoked, the pattern prevents partially initialized objects from leaking into the system, supporting safer and more predictable APIs.

Overall, the system comprises numerous data structures with optional fields, and builders offer a clean, expressive, and maintainable approach to creating them. Lombok further reduces boilerplate by generating the builder logic automatically, allowing the development team to benefit from the clarity and reliability of the pattern without additional implementation overhead. As a result, the Builder Pattern plays an important role in improving code readability, reducing errors, and supporting clean object construction throughout both the entity and DTO layers of the application.

2.2.1 Builder Pattern (Notification)


Builder Pattern - Notification
Notification
UUID id
User recipient
User sender
NotificationType type
String title
String message
Boolean read
LocalDateTime sentAt
builder() : NotificationBuilder
NotificationService
sendNotification()
builder()
build()
uses
NotificationBuilder
UUID id
User recipient
User sender
NotificationType type
String title
String message
Boolean read
LocalDateTime sentAt
id(value) : NotificationBuilder
recipient(value) : NotificationBuilder
sender(value) : NotificationBuilder
type(value) : NotificationBuilder
title(value) : NotificationBuilder
message(value) : NotificationBuilder
read(value) : NotificationBuilder
sentAt(value) : NotificationBuilder
build() : Notification