D4-D5: Design Goals, High-Level Arch. and Class Diagram. Design Patterns

S3T11

BilFind

Serhat Merak **(22002414)**
Mirza Özgür Atalar **(22003455)**
Kaan Türkoğlu **(22102105)**
Ege Karaahmetoğlu **(22102622)**
Orhun Aysan **(22103005)**

Table of Contents

1. Design Goals

1.1. Portability

BilFind utilizes Flutter, a cross-platform development framework, to ensure portability. Flutter allows the application to run on various devices and operating systems, meeting the diverse preferences of Bilkent University community. Flutter reduces the effort required to create and maintain separate codebases for different platforms. This aligns to achieve efficient portability for BilFind. We choose Portability as one of our most important design goals because we want our application to be accessed from every platform and every device. By using Flutter, it is possible to run our application on the Web, Android, iOS, and Desktop.

1.2. Modifiability

BilFind strongly emphasizes writing modular and maintainable code. The codebase is structured into independent modules, enabling easy modification and extension of features to adapt to changes in university policies and user requirements. Our modular codebase enhances BilFind's scalability, ensuring the app can evolve. This flexibility ensures adapting to new features and changes in the university environment. We have implemented the Model-View-Controller (MVC) pattern in our codebase, reinforcing the separation of concerns and providing a clear structure for enhanced maintainability.

2. Subsystem Decomposition Diagram

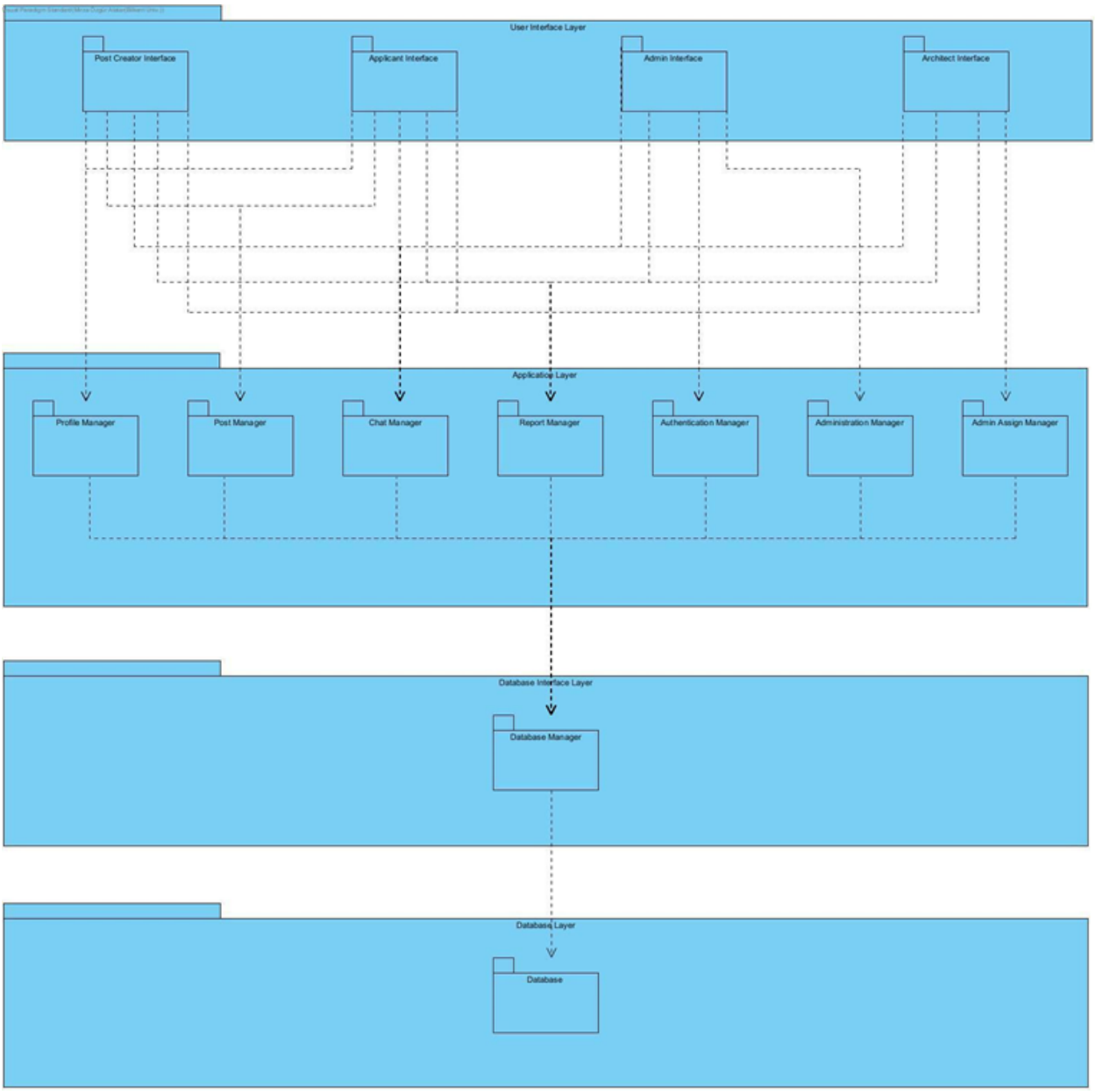

Figure 1 Subsystem Decomposition Diagram

3. Final Class Diagram

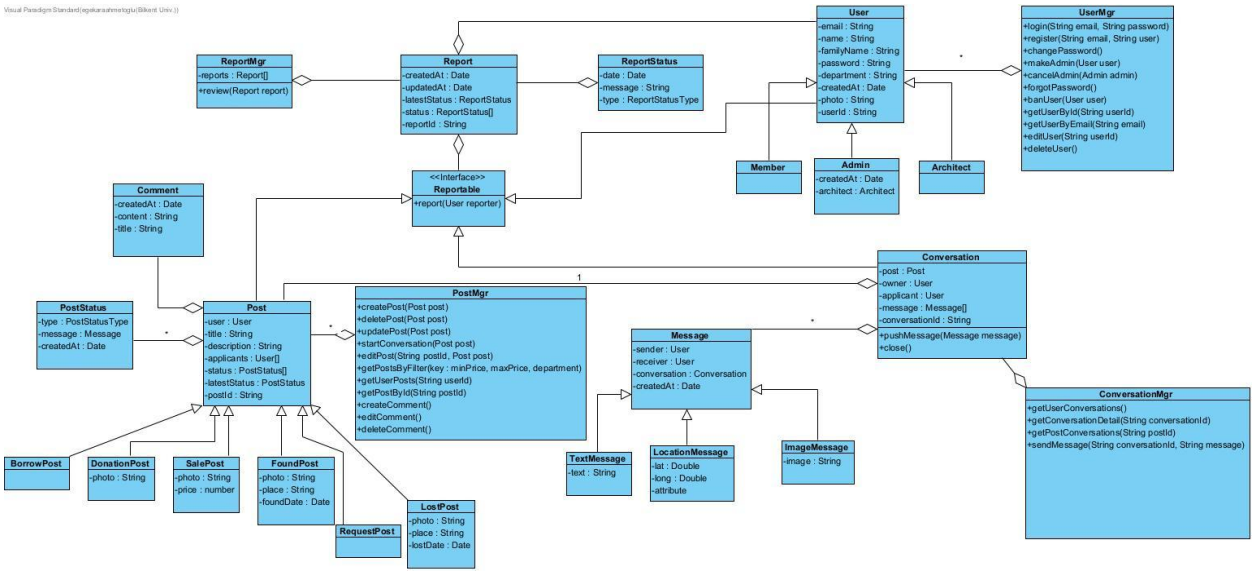

Figure 2 Final Class Diagram

4. Design Patterns

4.1. Singleton Pattern

From a technical standpoint, the Singleton Pattern guarantees that a class has just one instance and offers a global point of access. Through lazy initialization and a private constructor, the class instance is created only when first requested. To further prohibit instances from being created directly by external classes, the class constructor is also made private. Rather than creating new instances, the getInstance() method is used to guarantee that a single instance is kept alive for the lifecycle of the program.

The “Program.class” singleton is in charge of keeping track of and managing the user's data when they log in. Among the essential elements for user authentication and authorization is the authentication token.

Whether managing user profiles, dashboard functions, or other features, the Program singleton makes sure that the user's token and pertinent data are always accessible in every module or component of our online application. This design decision encourages a centralized approach to user management in addition to making implementation simpler.

We provide a cohesive and manageable framework for handling user-related data by utilizing the Singleton Pattern in our frontend architecture. In addition to making it easier to access important data across the application, this design lays the groundwork for future scalability and modifyability if more user-related functionalities are added.

4.2. Facade Pattern

Facade Design Pattern is a structural design pattern approach that provides a simplified interface to the complex interfaces in a subsystem and makes the user experience easier. In this design, there is a Facade class that provides a simplified interface and integrates all the subsystems in this interface. Subsystems are able to work independently from the Facade because the facade only has the purpose of facilitating the user experience. So, in short, the Facade class becomes a layer between the client and the subsystem and provides a more understandable interface.

In our application, we use the Facade approach in the backend design. Here, we create manager classes for various objects such as user, post, and conversation. These manager classes act as a bridge between
Mongo Database and controller classes that handle incoming HTTP requests. This approach has several benefits. Firstly, the manager classes encapsulate the complexity of directly interacting with Mongo Database, by being a middle layer between controllers and Mongo Database, since the controllers do not require knowing the details of the structure of the database. Secondly, it helps in terms of modularity, as each manager class represents several subsystems that are responsible for a specific aspect of the application, making it easier to maintain and manage the procedure.

The following table shows the usage of facade design patterns in our class diagram. The classes that are related to the facade pattern are highlighted in red.

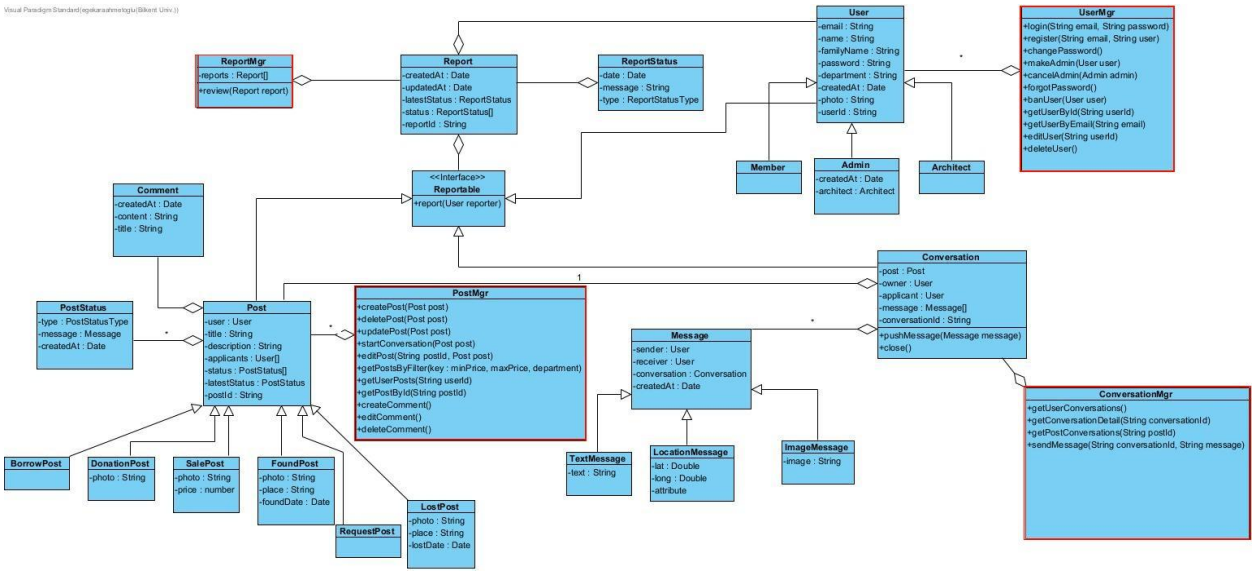

Figure 3 Highlighted Final Class Diagram