Bilkent University

Department of Computer Engineering

CS 319 - Object Oriented Software Engineering

Fall 2023

Section 1

Term Project

BilConnect

(Deliverable 4-5)

Team 3

Members

Abdullah Samed Uslu

22003598

Celal Salih Türkmen

22102498

Emre Akgül

22102310

Murat Çağrı Kara

22102505

Umut Bora Çakmak

22101806

Table of Contents

1. Design Goals and Design Trade-Offs

1.1 Design Goals

BilConnect has different design goals; however, two of them are the most important: usability and maintainability.

Usability

- The app should be easy to learn and use since BilConnect will be deployed into an already saturated market of second-hand sales, with its competing feature being its exclusivity. If the app is frustrating to learn or use, frustration can drive the users away from the app to competitor apps.
- The process from making a post to completing a sale should be easy every step of the way. Messaging is a functionality that will be used repeatedly throughout the process. As such, its UI should be easy to learn and use.
- The tag system should be intuitive, and primary/secondary tags should be obvious. Searching for a post should likewise be easy and without any frustrations.

Maintainability

- The code should be maintainable. Considering the app will serve limited functionalities and will likely not grow in its functionalities, the biggest concern should be maintaining the app.
- The developer team of five people is relatively small for an app that has the potential of several thousand users. Especially considering the app includes several live-service functionalities such as chatting and posting, the site cannot function in a limited manner and requires all its functionalities. To accommodate that, maintenance should be fast.
- In order to provide maintainability, the source code will include lots of controller and abstraction layers to make the architectural components loosely coupled. This design choice can provide flexibility and extensibility for future maintenance.

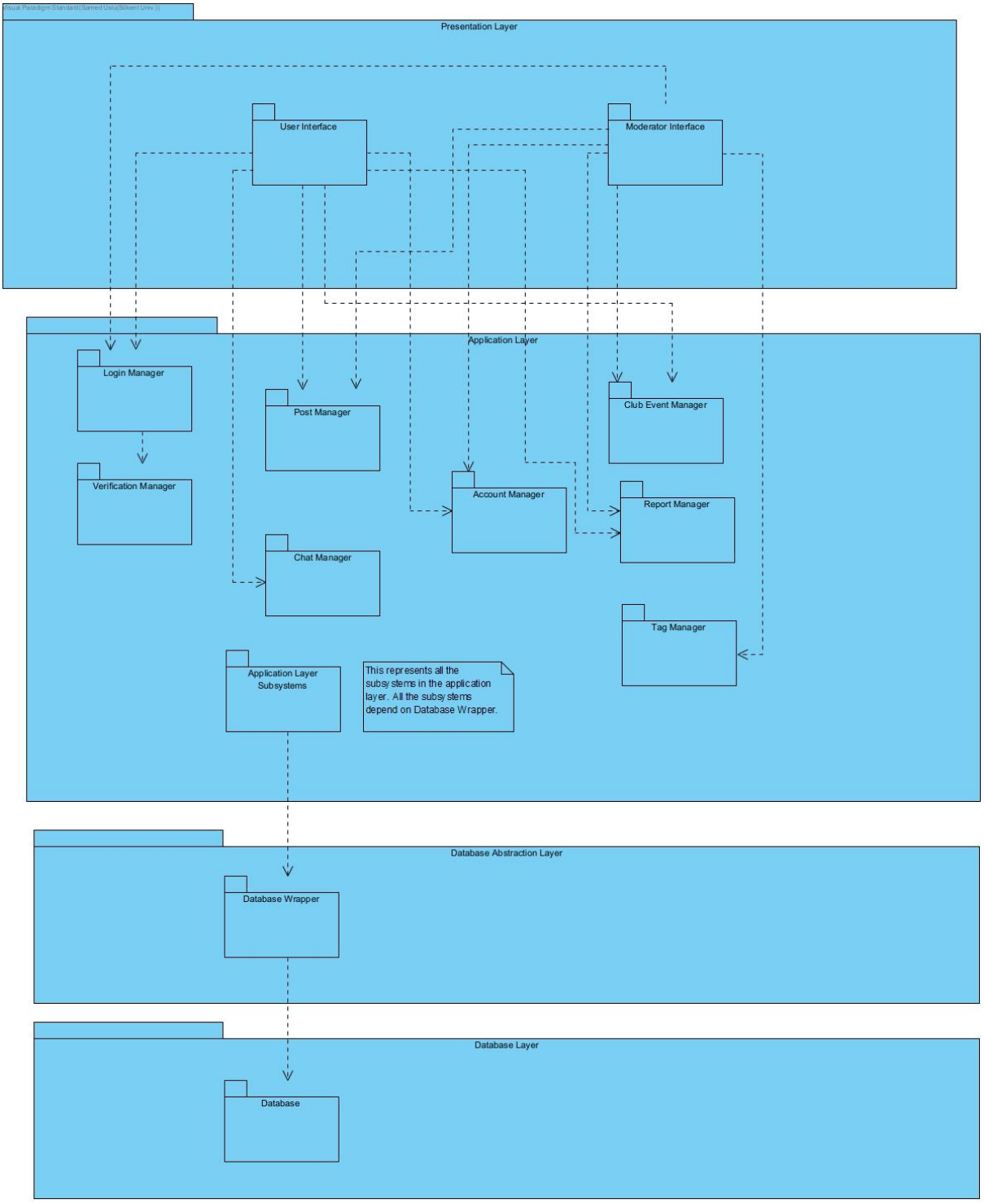
1.2 Design Trade-Offs

Security vs Usability: The app will contain intra-personal messaging and some personal details, which require security, and is exclusive to the Bilkent student body which requires exclusivity. These concerns both contradict usability.

Functionality vs Maintainability: If the app is successful to the extent that justifies having a bigger developer team, the priority of maintainability might get re-evaluated. In such an expansion, the addition of new functionalities might be considered bumping scalability and modifiability up in our priority list.

2. Subsystem Decomposition

The app has 4 subsystems: Presentation Layer, Application Layer, Database Abstraction Layer, and Database Layer.



User Interface: This is the interface that club accounts and user accounts can see. In this interface, users can reach LoginManager, PostManager, ClubEventManager,

ChatManager, AccountManager, and ReportManager to perform various CRUD operations and use various features of BilConnect.

Moderator Interface: This is the interface that the moderator of BilConnect can see, In this interface, the moderator can reach LoginManager, PostManager, ClubEventManager, TagManager, AccountManager, and ReportManager to perform various CRUD operations and use various features of BilConnect.

LoginManager: LoginManager is a controller that controls the registration and login process in the application. It interacts with the VerificationManager and DatabaseWrapper to perform its operations. It can be reached by ModeratorInterface and UserInterface.

VerificationManager: VerificationManager is a controller that controls the verification process, such as the requirement of Bilkent email in the registration part and credential validation in the login part for the LoginManager.

PostManager: PostManager is a controller that controls all CRUD operations for every kind of post in the application. Users can manage their posts and interact with other's posts with PostManager. It interacts with DatabaseWrapper to perform its operations. It can be reached by ModeratorInterface and UserInterface.

AccountManager: AccountManager is a controller that controls the RUD (create part is handled by LoginManager) operations for every kind of account in the application. It interacts with DatabaseWrapper to perform its operations. It can be reached by ModeratorInterface and UserInterface.

ClubEventManager: ClubEventManager is a controller that controls the CRUD operations for every club event. It interacts with DatabaseWrapper to perform its operations. It can be reached by ModeratorInterface and UserInterface.

ReportManager: ReportManager is a controller that controls the CRUD operations for every type of report in BilConnect. It interacts with DatabaseWrapper to perform its operations. It can be reached by ModeratorInterface and UserInterface.

ChatManager: ChatManager is a controller that controls and provides every chat feature to the users in BilConnect. It interacts with DatabaseWrapper to perform its operations. It can be reached by UserInterface.

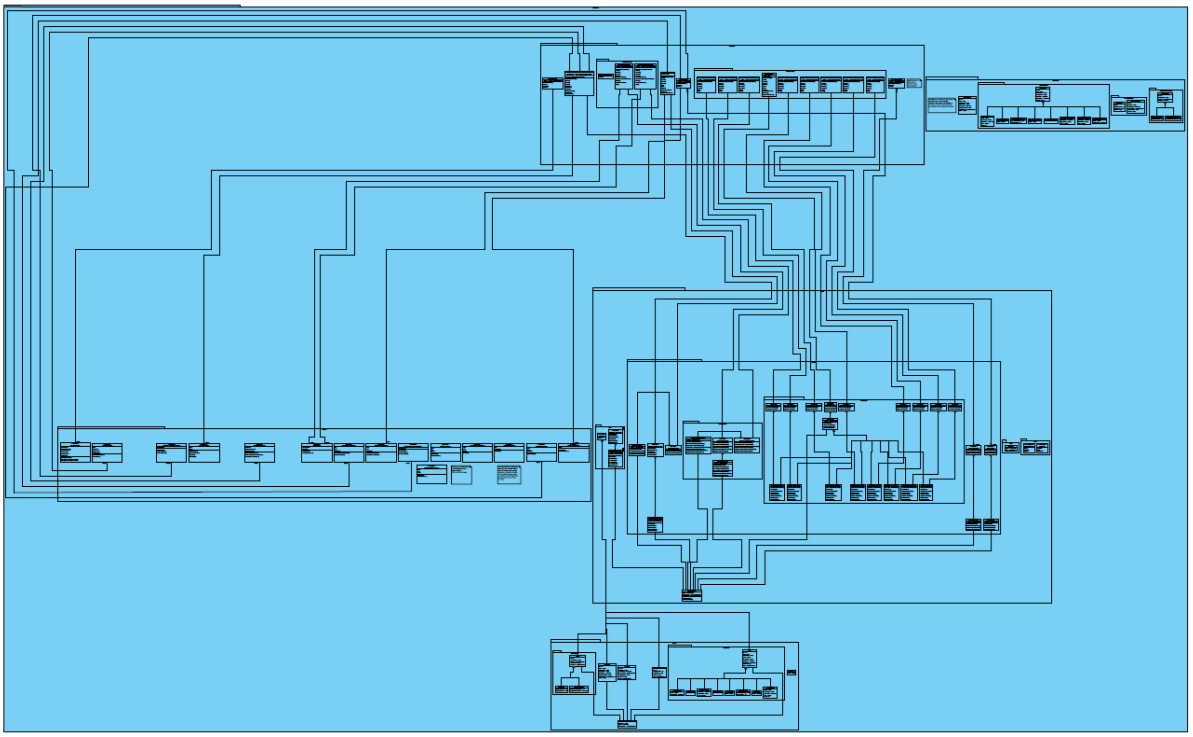
TagManager: TagManages is a controller that controls every CRUD operation for the post tags in BilConnect. It interacts with DatabaseWrapper to perform its operations. It can be reached by ModeratorInterface.

DatabaseWrapper: DatabaseWrapper provides abstraction between the application layer and the database layer. It helps to provide loosely coupled subsystem decomposition to provide maintainability.

Database: The database is where all useful information and data are stored in BilConnect. It can communicate with the application layer with the help of DatabaseWrapper.

3. Class Diagram

The class diagram is so big, it is provided as a separate PDF file in the folder. It is named ClassDiagram2.pdf.



4. Design Patterns

4.1 Repository Design Pattern

In the application, the repository pattern is used to create a clean separation between the database operations and logic layers, ensuring each repository class has a single responsibility. The interface *IEntityBaseRepository<T>* defines a set of methods for common CRUD operations. Using *EntityBaseRepository<T>*, we isolate the application's business logic from the data access layer. This assures each repository is concerned only with data access logic for any entity type. It also makes the code more maintainable. We use generics (T) for the interface. This implementation provides a reusable way to interact with different entities in the application. Therefore, each entity in the application has its repository that inherits from *EntityBaseRepository<T>*.

4.2 Facade Design Pattern

The main focus of the Facade pattern is to offer a simplified interface for a group of interfaces within a subsystem. This simplification aims to enhance usability and decrease complexity for clients. The app implements a Facade design pattern for every model using services, interfaces, and controllers. For example, let’s focus on the *Post* model. *PostsService* communicates with the database and abstracts the behavior related to posts. It acts as a subsystem, handling data storage and retrieval complexity. *IPostsService* defines the interface for interacting with the service, allowing flexibility to change implementations. *PostsController* communicates with *PostsService* and is involved in the decision-making process. It serves as a higher-level component that uses the *PostsService* to fulfill requests coming from the client. The controller serves as a facade in this scenario. It provides a simplified interface to the complex behavior of the *PostsService*. Clients, which may be components in the presentation layer or external systems, interact with the *PostsController* to perform actions related to posts without needing to know the details of database interactions. The *PostsService* class acts as a subsystem, encapsulating the details of working with the database and defining a higher-level interface through *IPostsService*. Clients (in this case, the *PostsController*) interact with the *PostsService* through the abstraction provided by the interface. This design is beneficial because it promotes the separation of concerns, making the source code cleaner and easier to maintain. Clients interact with the simplified interface of the facade without needing to deal with the complexities of the subsystem. Moreover, it facilitates the task division between team members by reducing complexity and compromising a clean design.