



Bilkent University

Department of Computer Engineering

CS319 Term Project

Fall 2022

Section 1

Group 1B

Design Report

Group members:

Ayşe Kelleci 21902532

Yusuf Doğan 21702970

Zeynep Hanife Akgül 22003356

Kardelen Ceren 22003017

Melisa Tanrıku 21703437

Instructor:

Eray Tüzün

Contents

1. Introduction	4
1.1. Purpose of the System	4
1.2. Design Goals	4
1.2.1. Increased Productivity	4
1.2.2. Functionality	4
1.3. Trade-Offs	5
1.3.1. Rapid development vs. Functionality	5
1.3.2. Usability vs. Functionality	5
2. System Architecture	6
2.1. Subsystem Decomposition	6
2.1.1. Interface Layer	7
2.1.2. Application Layer	8
2.1.3. Data Management Layer	8
2.2. Hardware/Software Mapping	9
2.3. Persistent Data Management	10
2.4. Access Control and Security	11
2.5. Boundary Conditions	12
2.5.1. Initialization	12
2.5.2. Termination	12
2.5.3. Failure	12
3. Low-Level Design	13
3.1. Final Object Design	13
3.2. Application Layer	14
3.3. Data Management Layer	15
3.4. Design Patterns	17
3.4.1. Singleton Design Pattern	17
3.4.2. Template Design Pattern	17
3.5. Packages	18
3.5.1. Controller Package	18
3.5.2. Interface Package	18
3.5.3. Entity Package	18
3.5.4. Database Package	18
3.5.5. External Package	18
3.6. Class Interfaces	19
3.6.1. Application Layer Interfaces	19
3.6.1.1. LoginController	19
3.6.1.2. MainPageController	20
3.6.1.3. ForumController	21
3.6.1.4. FAQController	22

3.6.1.5. DocumentsController	23
3.6.1.6. MyProfileController	24
3.6.1.7. StudentProfileController	24
3.6.1.8. NotificationController	25
3.6.1.9. MessagesController	26
3.6.1.10. CoursesController	27
3.6.1.11. Placement Controller	28
3.6.2. Data Management Layer Interfaces	29
3.6.2.1. Entity Interfaces	29
4. Improvements Summary	40
5. Glossary & References	41

1. Introduction

1.1. Purpose of the System

The purpose of the system is to create a user-friendly website where outgoing students and coordinators can access and handle their Erasmus-related tasks easily. Its difference from the current Erasmus website is, the communication between students and coordinators is done on the website instead of email, all document related tasks (e.g, generating and signing) are done digitally, and they can track their status from the website.

1.2. Design Goals

1.2.1. Increased Productivity

The system will create documents (e.g. Pre-Approval Form) and coordinators will sign them digitally, in order to decrease paper usage. Also, the course selection is done through the system, hence the Course Approval form can be created automatically by our system. Moreover, all users have a to-do list. This list is updated when a user is required to take an action, or, they can add an item to the list if needed. This functionality is useful as the coordinators are able to view what they need to do easily.

1.2.2. Functionality

Our system has many features to ease any process during the Erasmus period. To accomplish this, we included functionalities such as posting on forum, status viewing, to-do list, digitizing the form handling, enabling communication between students and coordinators etc. Also, since the form handling is done through our system, we will enable creating forms automatically and revising them by board members.

1.3. Trade-Offs

1.3.1. Rapid development vs. Functionality

Since we are trying to make our system as functional as possible, the development process is expected to take a long time. We included as much functionality as possible to enhance productivity by making the process easy and quick, so compromise from time is inevitable.

1.3.2. Usability vs. Functionality

Having functionality as one of our main design goals has some drawbacks on usability. More usable is more simple. That is to say, if we compare the usability of more functional and less functional software in which other aspects are equal, we would find less functional one more usable. To narrow this idea to our Erasmus Application software, having only course related operation in the app is usable, however, we trade off usability with functionality by integrating additional features to our software such as, forum, direct messaging, ToDo.

2. System Architecture

2.1. Subsystem Decomposition

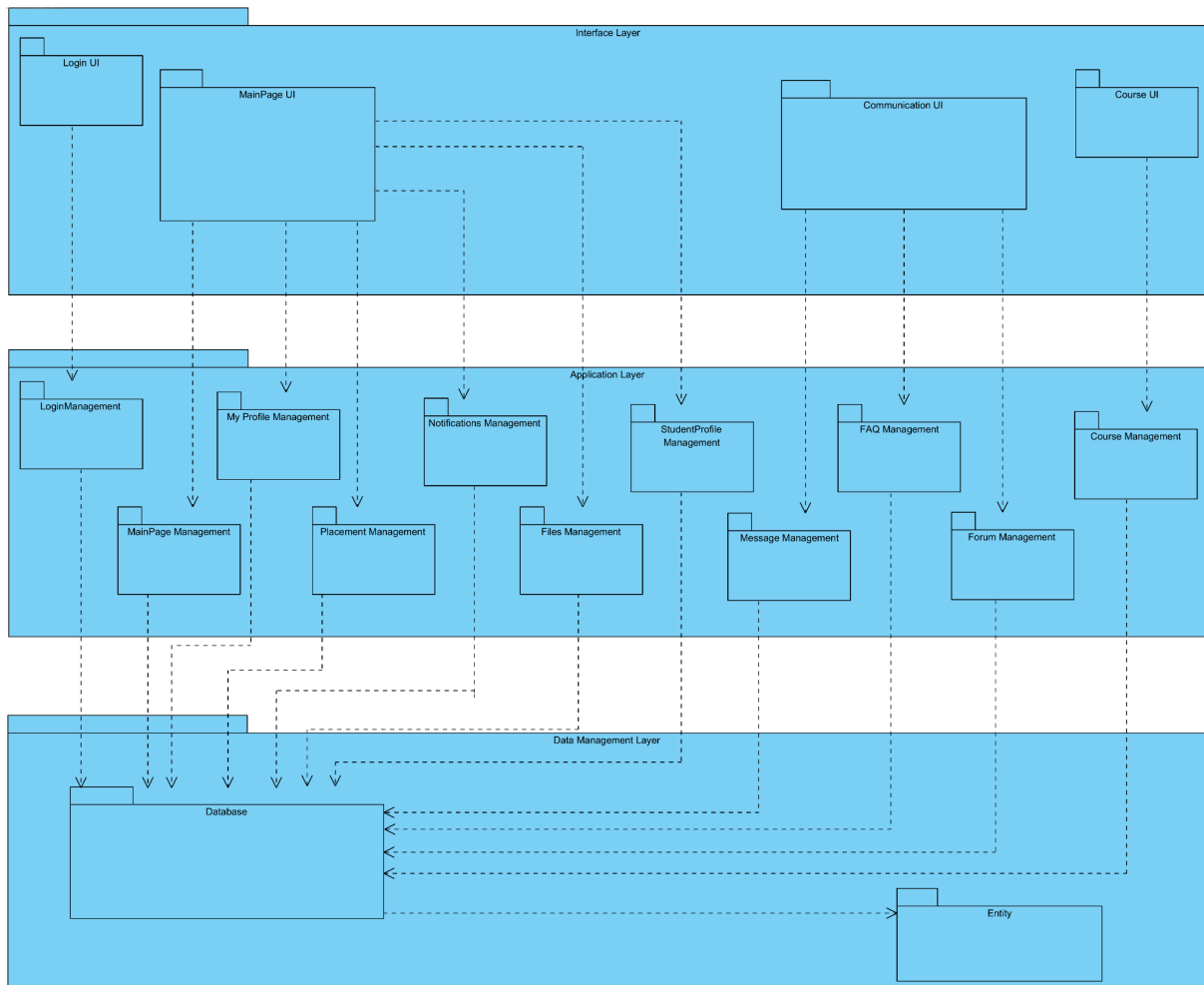


Fig. 2.1.1: Subsystem Decomposition of the System

In this part, we will describe our style of decomposition into subsystems, the reason why we chose it, and the responsibilities of each subsystem. During the decomposition, our goal was to reduce the complexity while facilitating the modification of the system. We decided that three-layer architectural style suited our purpose the best, because of the following reasons [1]:

- Each actor has different functionalities that are assigned to them, so different interfaces for each actor are required. This architectural style enables the development and modification of such for the same application logic.
- Each layer can run on separate server platforms, so in case of failure/customization of any server, the others won't be affected. This property

is significant as there will be a lot of essential data in our system that must not be lost.

We will now describe the responsibilities of each subsystem. As mentioned above, there are three layers: Interface Layer, Application Layer and Data Management Layer. We tried to decompose the systems so that the high coherence of each class and low coupling between each layer is achieved. The layers are as follows:

2.1.1. Interface Layer

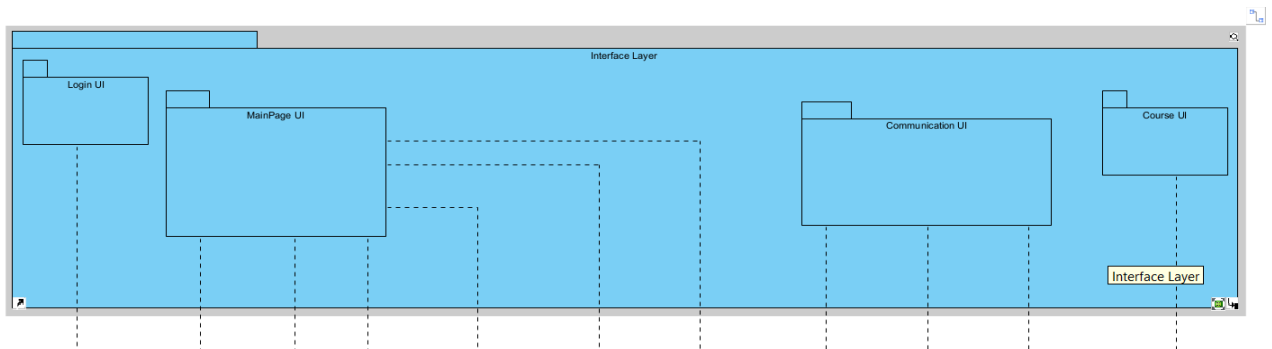


Fig. 2.1.2: Interface Layer of the Subsystem Decomposition of the System

The interface layer is created for every interface a user interacts with. This layer acts like a boundary object as every interface has certain elements that allow actions like clicking buttons or receiving inputs like typing text to corresponding areas. Therefore, some UI components are used for different management layers with abstraction. As interfaces tend to be changed often, creating this layer by separating it from the whole project enables quick adaptations without changes in application and data layers.

2.1.2. Application Layer

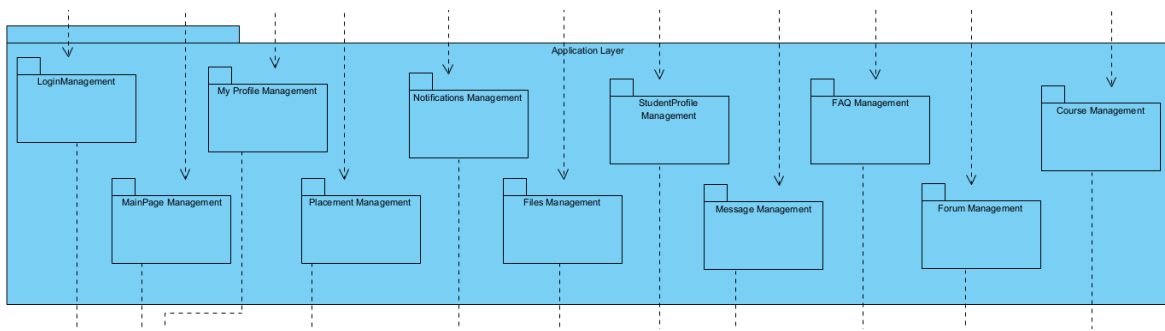


Fig. 2.1.3: Application Layer of the Subsystem Decomposition of the System

The application layer consists of management systems and logic operations, which are handled within back-end development. Every subsystem in this layer corresponds to a function in the application. As the application layer consists of controllers, it acts as a controller object that is dependent on the data layer.

2.1.3. Data Management Layer

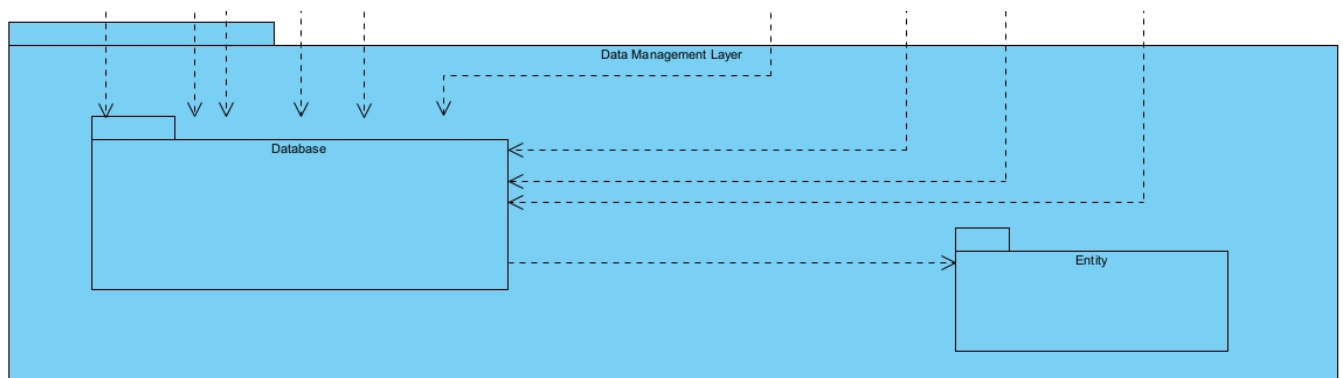


Fig. 2.1.4: Data Management Layer of the Subsystem Decomposition of the System

Data management layer handles the storage of information as well as retrieving them. Database and entity are two subsystems of this layer which contain repository interfaces and persistent entity classes, respectively. Consequently, our system is extensible and flexible as we have decomposed it into three layers. Possible changes in a layer will not affect others which increases reliability.

2.2. Hardware/Software Mapping

The system will run all the components on a single computer including the database and a local server. The system is web-based; therefore, the hardware systems are expected to run a web browser. The followings are the minimum requirements for a computer to be able to run the system which should be provided for both the clients and developers:

- 1.8 GHz x86 or x64 bit processor,
- 30 GB HDD,
- 512 MB RAM, and
- Stable internet connection with a minimum bandwidth of 50 KB/s.

Moreover, our database, Postgres SQL, is local and is kept on a single pc. We will use a variety of libraries in the development process and since some browsers or their older versions do not support all of them, we need to specify which versions are able to support them. The followings are the minimum required versions of browsers to be able to use our system:

- Google Chrome 23
- Internet Explorer 10
- Safari 6
- Opera 15
- Mozilla Firefox 21

Below is the deployment diagram of the system that shows the relationship between software and hardware components.

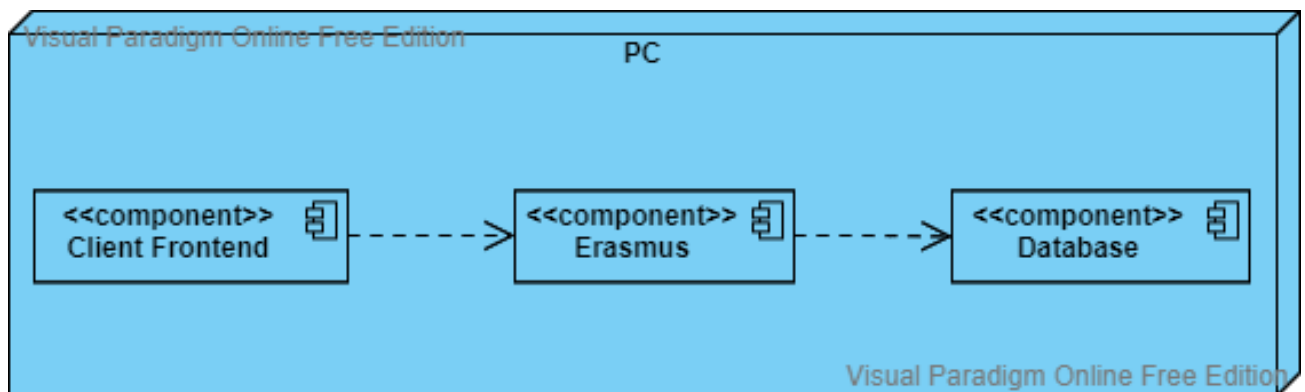


Fig. 2.2.1 Deployment diagram

2.3. Persistent Data Management

For data management, we will use the PostgreSQL database. PostgreSQL database is an open-source relational database management system. Entity objects in our systems and all information related to Erasmus (such as courses, the status of students, FAQ, etc.) are stored in the database. It is convenient to implement an object-oriented web application with the aid of relational systems since they perform better when the number of relations between data tables increases. In our project, we have a lot of dependencies and relations. For instance, course objects should belong to one university, and students have their course list. Also, each todo, notification, and chat object are dependent on at least one user. Therefore, we prefer to use the relational database management system. Moreover, PostgreSQL is the most preferred database system for Django because they are compatible with each other, and connecting them is easy. Since it is a common way, there are also many resources on the web.

2.4. Access Control and Security

As well as security is provided with a secure database PostgreSQL, restraints in user views also support privacy and security. Access control is made based on the user type, and corresponding interfaces and information are displayed to users only if they are allowed to reach out.

	Student	Coordinator	Board Member
Log In	x	x	x
Add Course	x		
Post on Forum	x	x	x
Reply a Post	x	x	x
View Post Replies	x	x	x
View FAQ	x	x	x
Add FAQ		x	
Approve Course		x	x
Direct Message	x	x	x
Message View	x	x	x
Edit Profile	x	x	x
Add ToDo	x	x	x
View User's Profile	x	x	x
View Student List		x	x
Upload File	x	x	x
Download File	x	x	x
Change Password	x	x	x

Table 2.4.1: Access Control Matrix

2.5. Boundary Conditions

2.5.1. Initialization

To be able to run the system, initialization will be done by starting up a local server through a computer's command prompt. Including database and server, everything will be rooted on a local computer on which the application will be runned. To run the latest version of the system which includes the current database data, a latest database file of Postgres SQL should be uploaded to the local computer.

In order to log in to the system, there must be an account assigned to the user in the database by administrators; otherwise, they can only see login and forgot password. If a user is logged in, the same account can also log in from other devices as well. When the user is logged in, all information on the page is initialized from the database.

2.5.2. Termination

If one of the subsystems is terminated, then the whole application terminates as well since the system works as a whole structure. There is no termination condition, such as checking if every data is saved on the database before termination, since data is recorded on the database simultaneously when an action is taken. If an admin initiates the termination process, in order to accomplish zero data loss, all data is saved to the database. Subsystems do not communicate with each other during the termination process, they terminate within themselves. When a user wants to exit from the system, they can click the button "Log out" from the right top of the screen.

2.5.3. Failure

We don't have any case for unexpected crashes. The current data will be lost and none of the functions will work. However, a pop-up will be displayed to the user to explain the situation and the developers will receive a notification through email about the failure.

3. Low-Level Design

3.1. Final Object Design

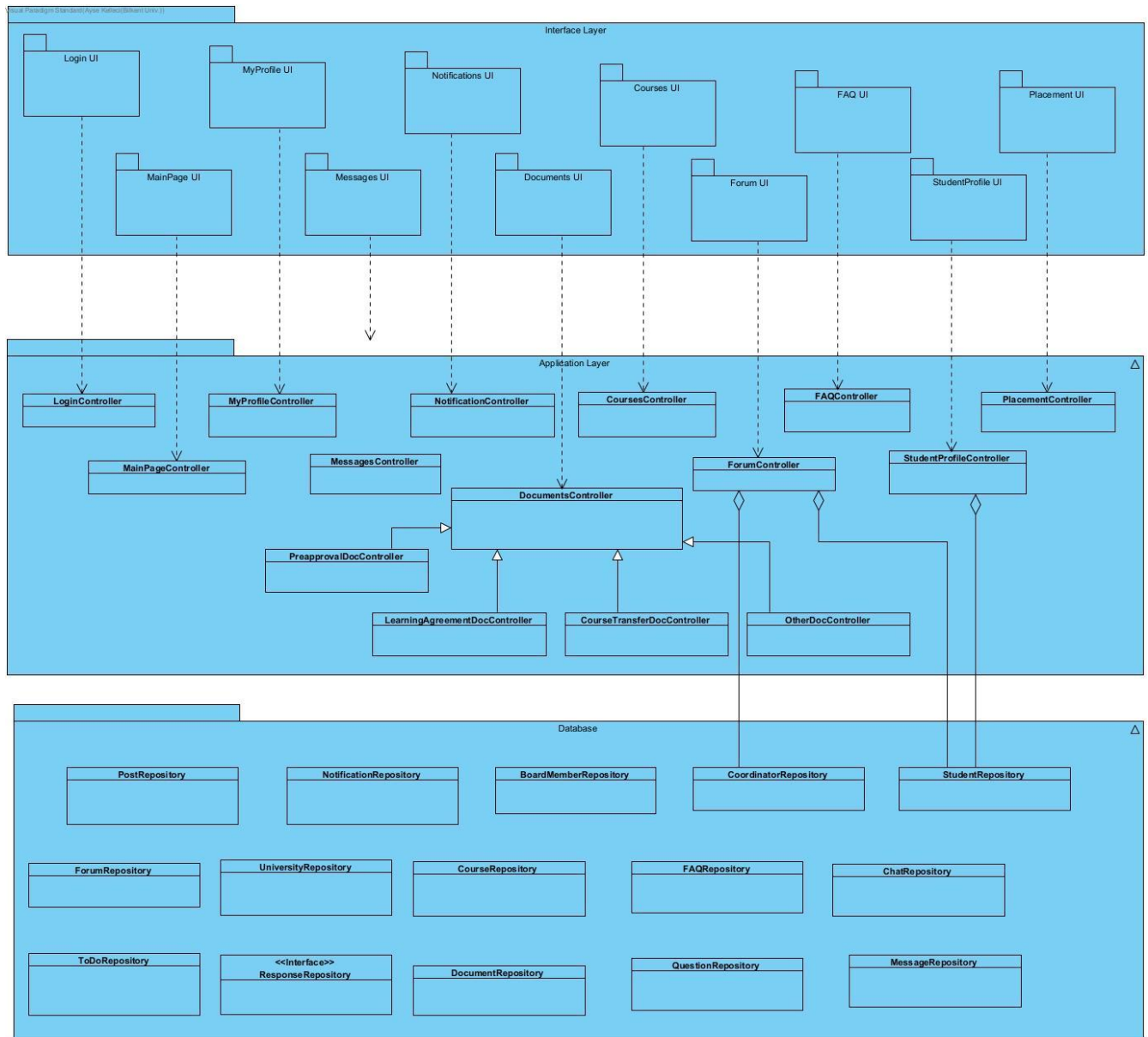


Fig. 3.1.1: Final Object Design Diagram of the System

In the Database layer; studentRepository, coordinatorRepository and boardMemberRepository are connected to each controller (except ForumController and StudentProfileController as shown above) with an aggregation relationship, but for simplicity purposes, not every link is shown in the above diagram.

Each repository is connected to corresponding entities but again for simplicity, the linkages are not shown.

3.2. Application Layer

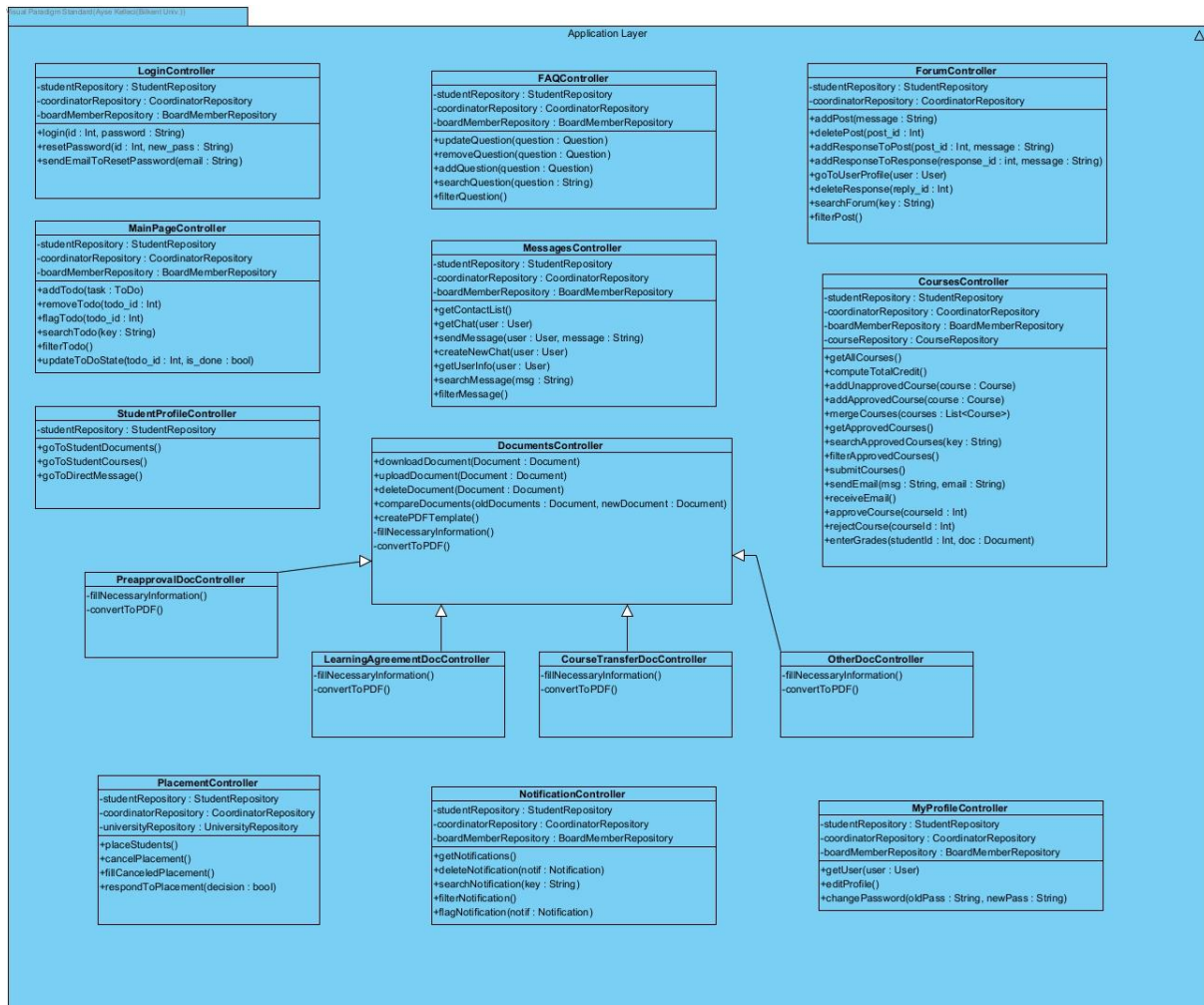


Fig. 3.2.1: Application Layer of the Subsystem Decomposition of the System

Application layer handles the linking of users' input with data management. When necessary, required information is taken by the user to be returned to the data management layer.

3.3. Data Management Layer

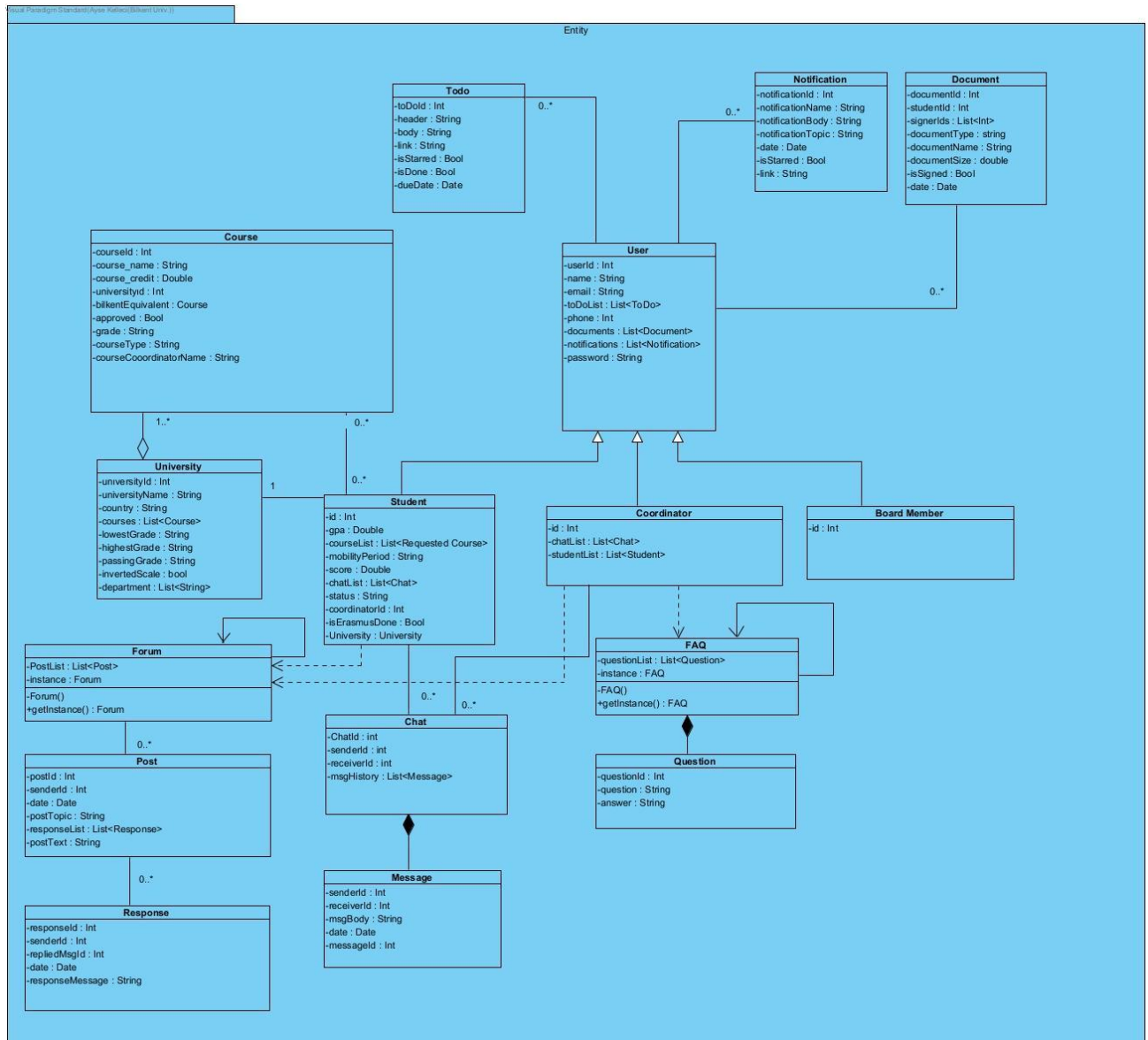


Fig. 3.3.1: Entity Subsystem of the Data Management Layer

Each class in Entity has its own constructors, getter, and setter methods but for simplicity, they are not explicitly written in the above diagram of Entity.

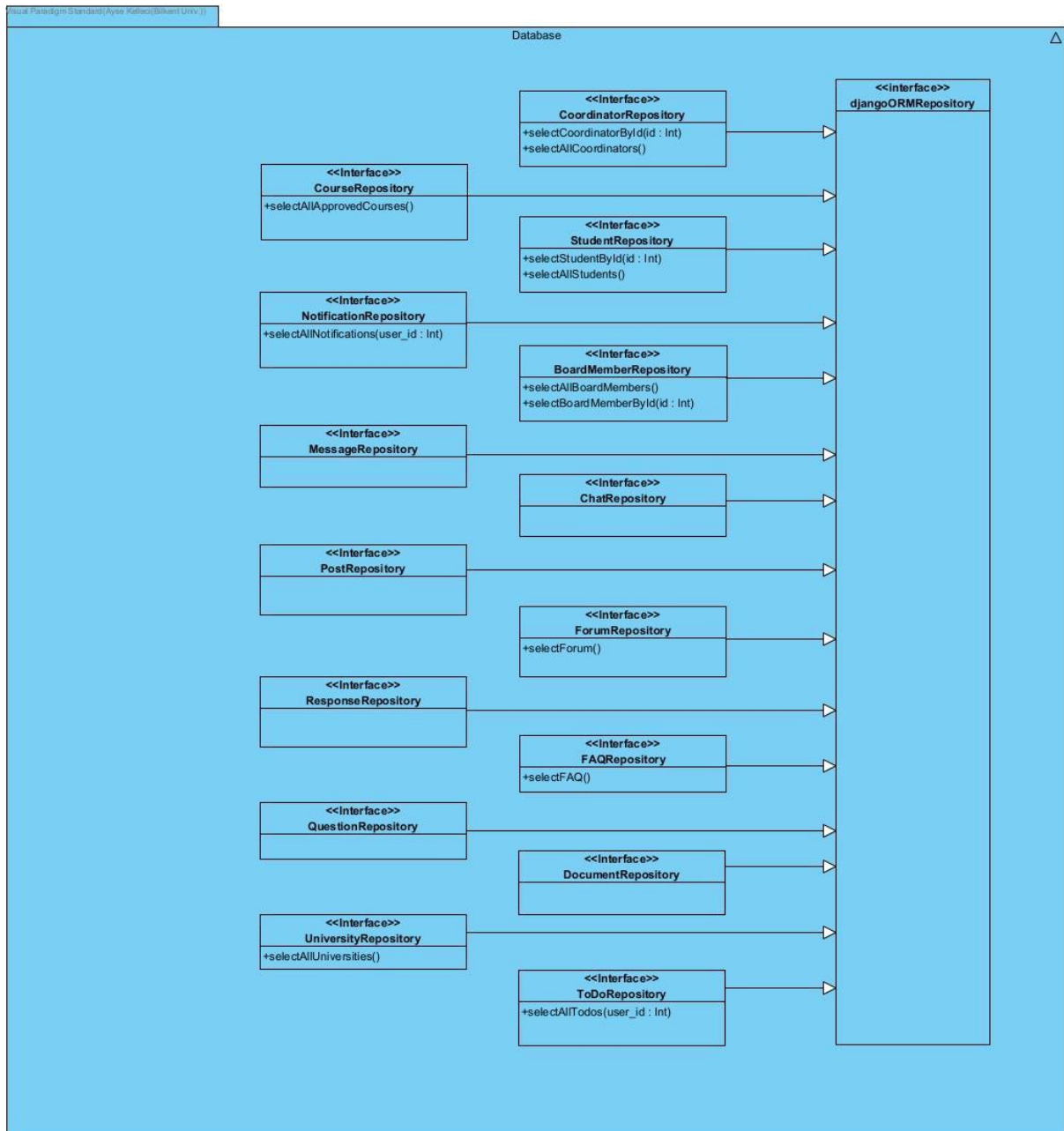


Fig. 3.3.2: Database Subsystem of the Data Management Layer

Data Management layer consists of two subsystems and the database is one of them. This subsystem has repositories for specific categories of data which inherit from djangoORMRepository. The repositories are separated according to which data they handle for a better readability and less complexity.

3.4. Design Patterns

3.4.1. Singleton Design Pattern

Singleton Design ensures that a certain class has only one instance and provides a global access point to that instance [2]. This pattern is used in FAQ and Forum classes as seen in Entity. Usage of only one instance occupies less memory while making the code less complex thus, more readable.

3.4.2. Template Design Pattern

Template design pattern suggests defining a system as the skeleton of operations and leaving the implementation part to child classes. The overall structure is created by the parent classes [3]. The reflection of template design in this project is in the application layer, documentsController class defines fillNecessaryInformation() and convertToPDF() methods as skeletons which are overridden by its child classes. createPDFTemplate() in documentsController class is the caller of these methods.

3.5. Packages

3.5.1. Controller Package

It contains all the controller classes. It also handles interaction between server and other packages.

3.5.2. Interface Package

In the interface part we do not use any package or library. The user interface will be produced by using HTML, CSS and JavaScript.

3.5.3. Entity Package

It is a package that includes all entity objects such as student objects or a course object. Entity package can only interact with controller package.

3.5.4. Database Package

The database package will keep a database class that handles the storage of information as well as retrieving them. The package interacts with controllers for receiving data to be stored and retrieving data to controller operations.

3.5.5. External Package

3.5.5.1. Django: It is a fully featured server-side web framework written in Python, which we utilize in our software.

3.6. Class Interfaces

3.6.1. Application Layer Interfaces

3.6.1.1. LoginController

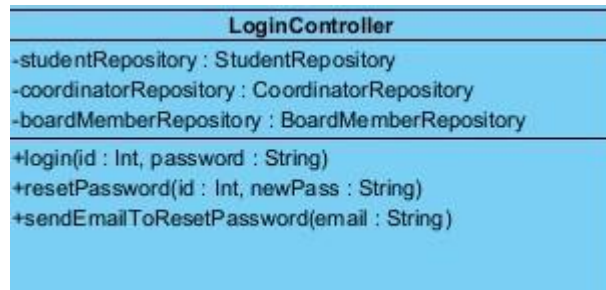


Fig. 3.6.1.1: Login Controller

A class that controls log-in operations.

Operations:

public login(int id, String password):

This operation checks if the given ID and password match with a user on the database and if so, returns the user.

public resetPassword(int id, String newPass):

This operation renews the password with the new password that comes through user email.

public sendEmailToResetPassword(String email);

This operation sends an email to the user for reset password request.

3.6.1.2. MainPageController

MainPageController
-studentRepository : StudentRepository -coordinatorRepository : CoordinatorRepository -boardMemberRepository : BoardMemberRepository
+addTodo(task : ToDo) +removeTodo(todo_id : Int) +flagTodo(todo_id : Int) +searchTodo(key : String) +filterTodo() +updateToDoState(todo_id : Int, isDone : bool)

Fig. 3.6.1.2: Main Page Controller

A class that controls operations related to the to-do list and the status shown on the main page.

Operations:

public getStatus():

This operation gets the status of the user from the database.

public getTodoList():

This operation gets the todo list of the user from the database.

public addTodo(ToDo task):

This operation adds a todo task to the user's todo list.

public removeTodo(int todo_id):

This operation removes a todo task from the user's todo list.

public flagTodo(int todo_id):

This operation flags a todo task in the user's todo list.

public searchTodo(String key):

This operation searches for todo tasks in the user's todo list.

public filterTodo():

This operation filters todo tasks.

public updateToDoState(int todo_id, bool isDone):

This operation updates the todo task's state (whether it's done or not).

3.6.1.3. ForumController

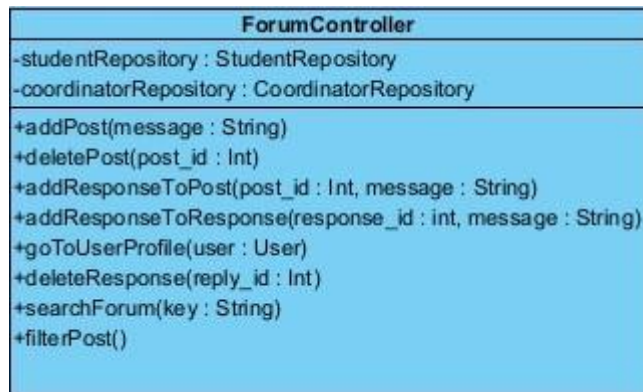


Fig. 3.6.1.3: Forum Controller

A class that controls forum related operations.

Operations:

public getAllPosts():

This operation gets all the posts in the forum.

public getPostDetail(int post_id):

This operation gets details of the post with the given post id.

public addPost(String message):

This operation creates a new post with the given heading.

public deletePost(int post_id):

This operation removes the post with its responses.

public addResponseToPost(int post_id, String message):

This operation adds a response to a post.

public addResponseToResponse(int response_id, String message):

This operation is used to respond to a response of a post.

public goToUserProfile(User user):

This operation leads to a user's profile.

public deleteResponse(int reply_id):

This operation removes the response made to a post or a response.

public searchForum(String key):

This operation searches the given key in the entire forum.

public filterPost():

This operation enables users to filter posts.

3.6.1.4. FAQController

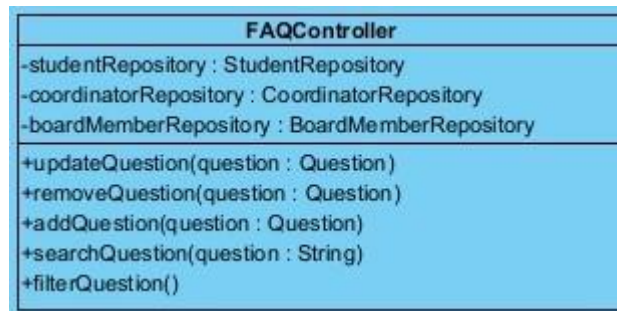


Fig. 3.6.1.4: FAQ Controller

A class that controls FAQ related questions.

Operations:

public getQuestionList():

This operation gets all questions in the FAQ.

public updateQuestion(Question question):

This operation edits question contents.

public removeQuestion(Question question):

This operation removes the given question.

public addQuestion(Question question):

This operation adds a question to the FAQ.

public searchQuestion(String question):

This operation searches questions by a given string.

public filterQuestion():

This operation is used to filter questions.

3.6.1.5. DocumentsController

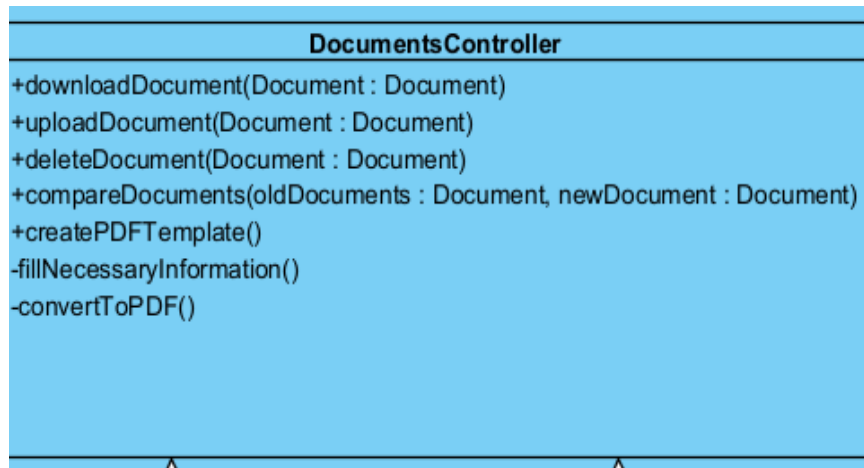


Fig. 3.6.1.5: Document Controller

A class that controls operations related to the documents generated and stored in the system.

Operations:

public getAllDocuments():

This operation gets all the documents in the system.

public downloadDocument(Document document):

This operation enables document downloading.

public uploadDocument(Document document):

This operation enables document uploading.

public deleteDocument(Document document):

This operation removes the document from the user's files section.

public compareDocuments(Document oldDocument, Document newDocument):

This operation is used to compare two documents.

public createPDFTemplate():

This operation creates pdf templates of three type of documents (learning agreement, course transfer, pre approval).

public fillNecessaryInformation():

This operation fills necessary user information in the template.

public convertToPDF:

This operation is used to convert the document to a pdf.

3.6.1.6. MyProfileController

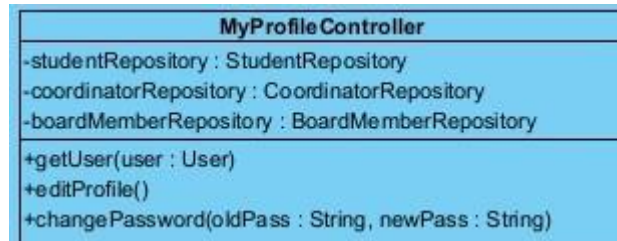


Fig. 3.6.1.6: My Profile Controller

This class enables the user to control his/her own profile.

Operations:

public getUser(User user):

This getter operation receives the user.

public editProfile():

This operation enables users to edit profiles.

public changePassword(String oldPass, String newPass):

This operation changes the password of the user.

3.6.1.7. StudentProfileController

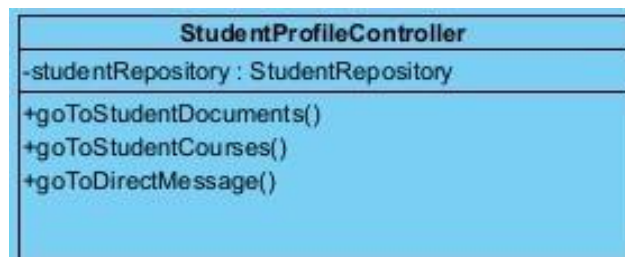


Fig. 3.6.1.7: Student Profile Controller

A class that controls a student profile as seen by the coordinator.

Operations:

public goToStudentFiles():

This operation leads the user to student files.

public goToStudentCourses():

This operation leads the user to student's courses.

public goToDirectMessage():

This operation leads the user to direct messages.

3.6.1.8. NotificationController

NotificationController
-studentRepository : StudentRepository -coordinatorRepository : CoordinatorRepository -boardMemberRepository : BoardMemberRepository
+getNotifications() +deleteNotification(notif : Notification) +searchNotification(key : String) +filterNotification() +flagNotification(notif : Notification)

Fig. 3.6.1.8: Notification Controller

A class that controls operations related to the notifications.

Operations:

public getNotifications():

This operation gets all notifications.

public deleteNotification(Notification notif):

This method deletes the given notification.

public searchNotification(String key):

This operation searches notifications with the given key.

public filterNotification():

This operation filters notifications.

public flagNotification(Notification notif):

This operation is used to flag a notification.

3.6.1.9. MessagesController

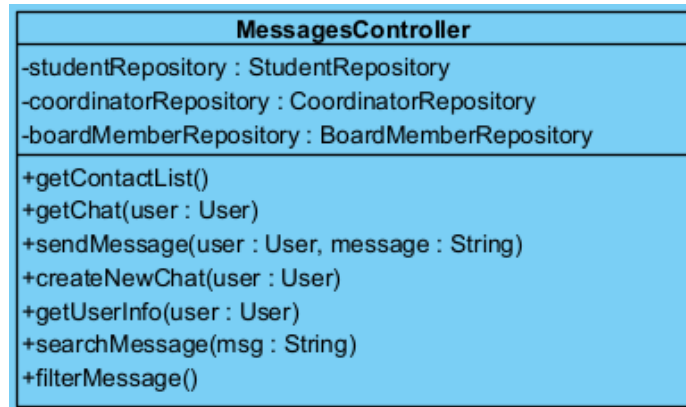


Fig. 3.6.1.9: Message Controller

A class related to operations related to the messages system.

Operations:

public getContactList():

This operation gets the user's contact list.

public getChat(User user):

This operation gets the whole chat with the received user parameter.

public sendMessage(User user, String message):

This operation is used to send a message to other users in the existing chat.

public createNewChat(User user):

This operation is to create a new chat with a new user.

public getUserinfo(User user):

This method gets user info.

public searchMessage(String msg):

This operation searches a specific word/words in messages.

public filterMessage():

This operation filters the messages.

3.6.1.10. CoursesController

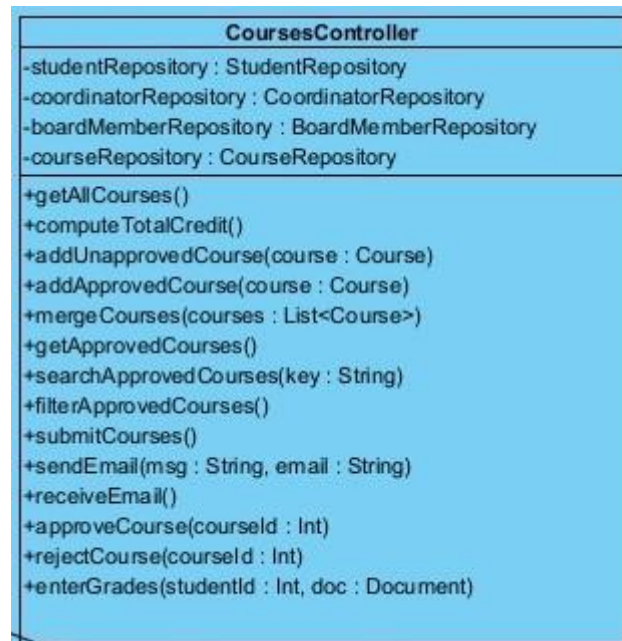


Fig. 3.6.1.10: Courses Controller

A class that controls the course related operations.

Operations:

public getAllCourses():

This operation gets all the courses in the system.

public compute TotalCredit():

This operation computes total credits of added courses.

public addUnapprovedCourse(Course course):

This operation adds an unapproved course.

public addApprovedCourse(Course course):

This operation adds an approved course.

public mergeCourses(List<Course> courses):

This operation merges desired courses so that they can be equivalent to a course in Bilkent.

public getApprovedCourses():

This operation gets previously approved courses.

public searchApprovedCourses(String key):

This operation is used to do search courses by a keyword.

public filterApprovedCourses():

This operation filters approved courses.

public submitCourses():

This operation submits added courses.

public sendEmail(String msg, String mail): This operation is used to send emails to instructors regarding unapproved must courses.

public receiveEmail(): This method receives email from corresponding mail addresses of faculty members on approval of must courses.

public approveCourse(int courseId): With this operation an added course is approved.

public rejectCourse(int courseId): With this operation an added course is rejected.

public enterGrades(int studentId, Document doc): This operation lets the coordinator enter grades to student documents.

3.6.1.11. Placement Controller

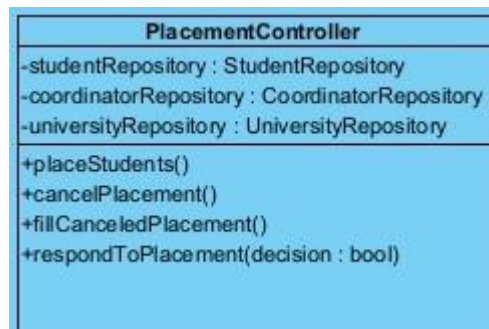


Fig. 3.6.1.11: Placement Controller

A class to control student placement related operations.

Operations:

public placeStudents():

This operation places students according to their preferences and scores.

public cancelPlacement():

This operation cancels assigned placement.

public fillCancelledPlacement():

This operation fills canceled placements with the next student.

public respondToPlacement(bool decision):

This operation responds yes/no to placement.

3.6.2. Data Management Layer Interfaces

3.6.2.1. Entity Interfaces

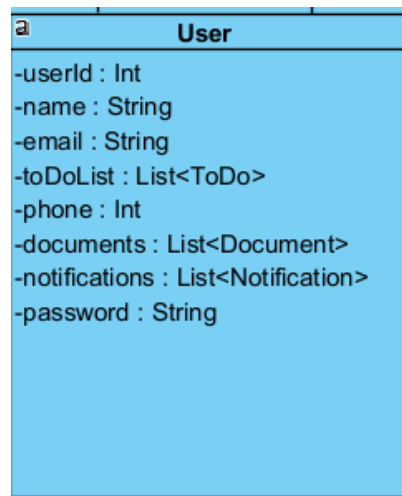


Fig. 3.6.2.1: User Entity

This class represents any registered user.

Attributes:

private int userId: ID associated with the user in the database.

private String name: Full name of the user.

private String email: Email address of the user.

private List<ToDo> todoList: List of the user's ToDo items.

private int phone: Phone number of the user.

private List<Document> documents: List of the user's documents.

private List<Notification> notifications: List of the user's notifications.

private String password: Password of the user.

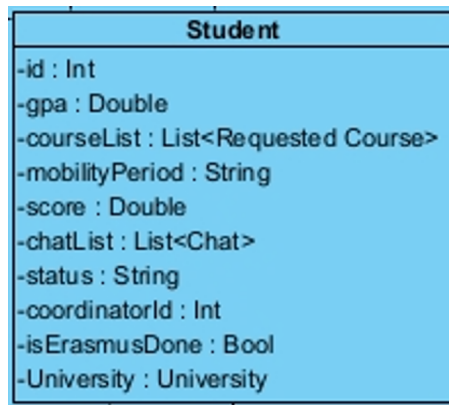


Fig. 3.6.2.2: Student Entity

This class represents a student user.

Attributes:

private int id: The Bilkent student id of the user (not the database id).

private double gpa: The cumulative GPA of the student.

private List<Requested Course> courseList: The list of courses the student is planning to take or taking.

private String mobilityPeriod: The mobility period of the student.

private double score: Total score of the student for placement purposes.

private List<Chat> chatList: All the chats of the student.

private String status: The status of the student.

private int coordinatorId: The id of the student's coordinator.

private bool isErasmusDone: A boolean variable for whether the student has completed their mobility period or not.

private University university: The university the student is placed in.

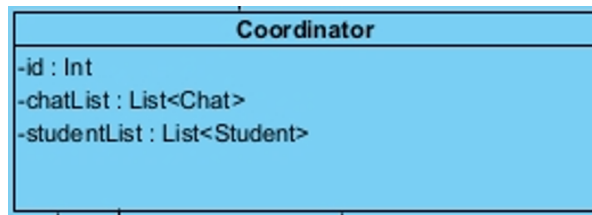


Fig. 3.6.2.3: Coordinator Entity

This class represents the coordinator type users.

Attributes:

private int id: Bilkent id of the coordinator

private List<Chat> chatList: List of chats the coordinator has, represented as Chat objects.

private List<Student> studentList: List of students represented as Student objects.



Fig. 3.6.2.4: Board Member Entity

This class represents a faculty board member user.

Attributes:

private int id: Bilkent id of the board member.

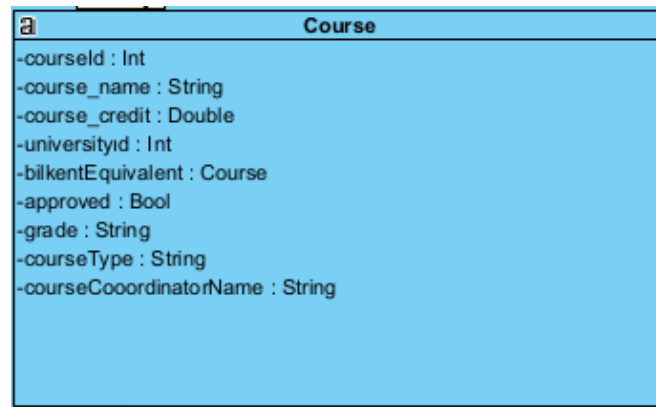


Fig. 3.6.2.5: Course Entity

This class represents courses.

Attributes:

private int courseId: Database id of the course.

private String course_name: The name of the course.

private double course_credit: The ECTS credits of the course.

private int universityId: Database id of the university student is going.

private Course bilkentEquivalent: This course's equivalent in Bilkent, if applicable.

private bool approved: A boolean variable for whether the course is approved by the coordinator or not.

private String grade: The grade the student received.

private String courseType: The course type, such as mandatory or elective.

private String courseCoordinatorName: The course coordinator's name for Bilkent University courses.

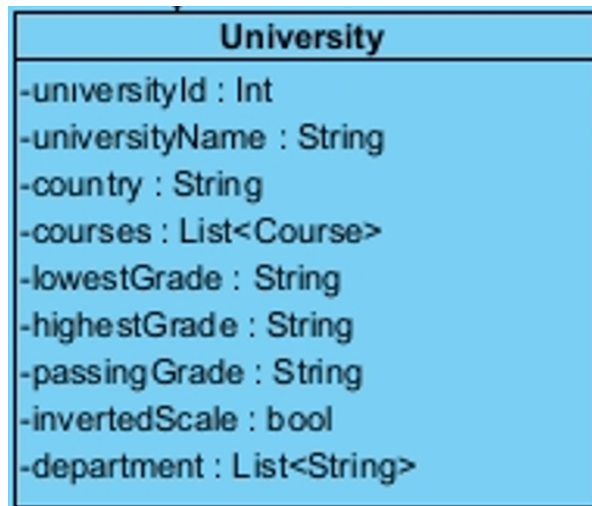


Fig. 3.6.2.6: University Entity

This class represents foreign universities.

Attributes:

private int universityId: Database id of the university.

private String universityName: Name of the university as string.

private String country: Name of the country in which the university is located.

private List<Course> courses: The list of approved courses in that university.

private String lowestGrade: The lowest grade for the course.

private String highestGrade: The highest grade for the course.

private String passingGrade: The minimum grade to pass the course.

private bool invertedScale: It shows whether scale is ordered from the highest grade to the lowest grade, or the opposite.

private List<String> department: The applicable departments in the university.

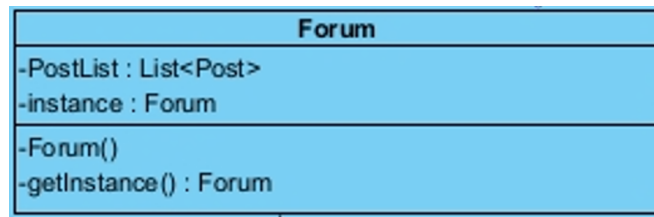


Fig. 3.6.2.7: Forum Entity

This class represents the Forum.

Attributes:

private List<Post> PostList: List of posts in the Forum represented as Post objects.

private Forum instance: This attribute holds one instance of this class, in line with the Singleton design pattern.

Operations:

private Forum(): This private constructor initializes the one Singleton instance of this class.

public Forum getInstance(): This operation returns the one Singleton instance of this class.

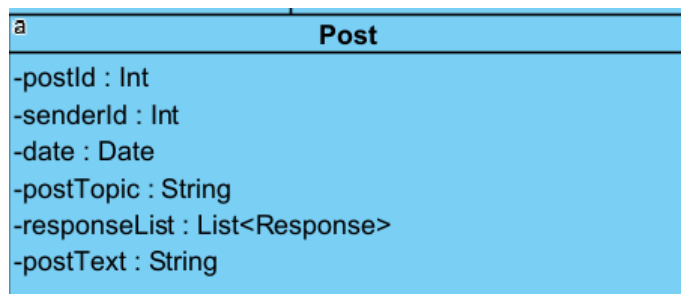


Fig. 3.6.2.8: Post Entity

This class represents a single post on Forum.

Attributes:

private int postId: Database id of a single post.

private int senderId: Database id of the poster user.

private Date date: Date object representing the date on which the post is posted on the forum.

private String postTopic: The subject of the post.

private List<Response> responseList: A list that consists of Response objects which represent replies to posts.

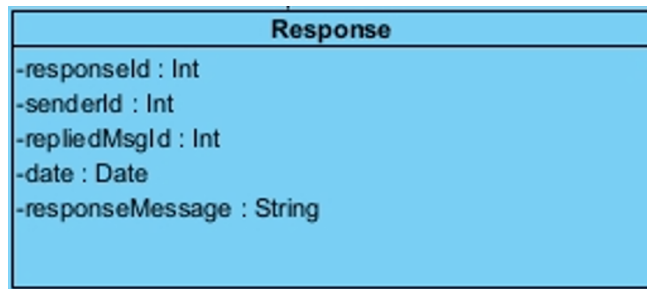


Fig. 3.6.2.9: Response Entity

This class represents a response to a post or another response on Forum.

Attributes:

private int responseld: Database id of the response.

private int senderId: Database id of the user who posted a reply.

private int repliedMsgId: Database id of the post that is being replied with a Response object.

private Date date: Date object representing on which date the reply is posted.

private String responseMessage: Text body of the response.



Fig. 3.6.2.10: Chat Entity

This class represents a single chat.

Attributes:

private int ChatId: Database id of the chat.

private int senderId: Database id of the user who is the sender side of the chat.

private int receiverId: Database id of the user who is the receiver side of the chat.

private List<Message> msgHistory: A list consisting of message objects in the chat.

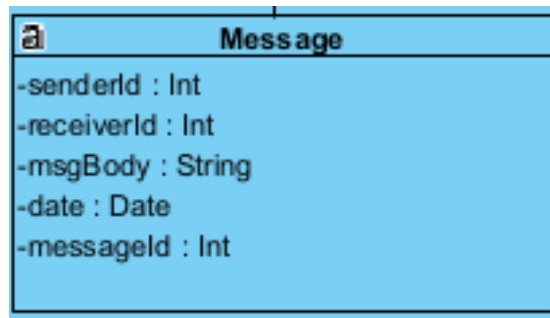


Fig. 3.6.2.11: Message Entity

This class indicates messages in the system.

Attributes:

private int senderId: The id of the message sender.

private int receiverId: The id of the message receiver.

private String msgBody: String containing message content.

private Date date: The date message is sent.

private int messageId: Database id of the message.

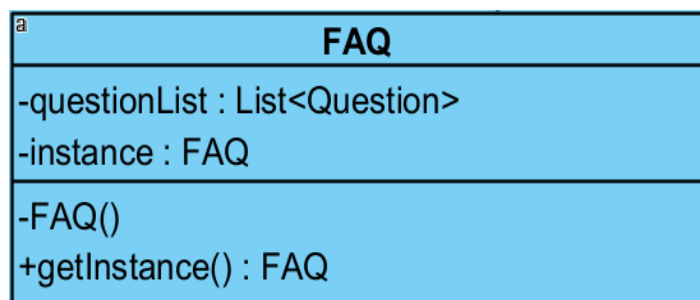


Fig. 3.6.2.12: FAQ Entity

This class illustrates the FAQ in the system.

Attributes:

private List<Question> questionList: The list of question objects that will be stored in database

private FAQ instance: It is a single FAQ object that will be stored in database

Operations:

private FAQ(): This private constructor initializes the one Singleton instance of this class.

public FAQgetInstance(): This operation returns the one Singleton instance of this class.

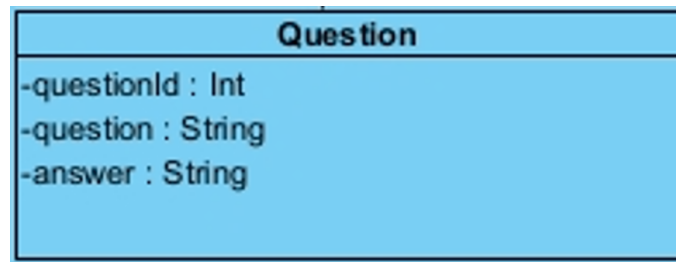


Fig. 3.6.2.13: Question Entity

This class represents questions in the FAQ.

Attributes:

private int questionId: Unique id for a question to store in the database.

private String question: String containing the question.

private String answer: String containing the answer to the question.

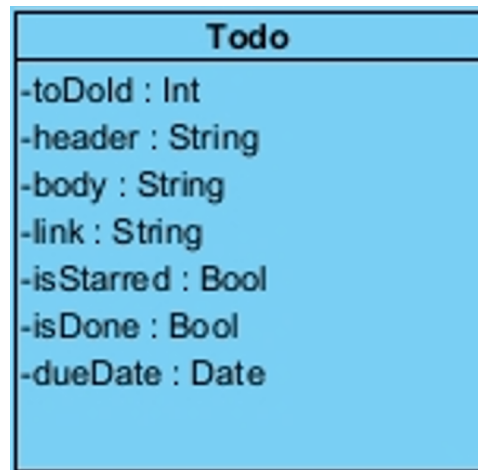


Fig. 3.6.2.14: To-do Entity

This class indicates ToDo items in the system.

Attributes:

private int toDoId: Unique ToDo id stored in database.

private String header: String including the title of ToDo.

private String body: String including actual content of ToDo.

private String link: String that contains the link associated with the ToDo.

private bool isStarred: a boolean to store if the ToDo has a star or not.

private bool isDone: a boolean to store if the ToDo is done or not.

private Date dueDate: Due date of ToDo, if it is user created ToDo then user can define a due date.

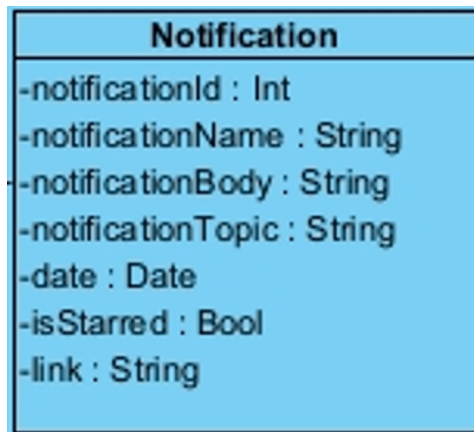


Fig. 3.6.2.15: Notification Entity

This class represents the notifications in the system.

Attributes:

private int notificationId: Unique notification id stored in database.

private String notificationName: Heading of the notification.

private String notificationBody: The actual content of notification.

private String notificationTopic: The subject of the notification such as, messages, course approval.

private Date date: The date of the notification when the user received it.

private bool isStarred: boolean that stores whether the notification is starred or not.

private String link: The link that includes the corresponding interface user should reach when clicked on the notification.

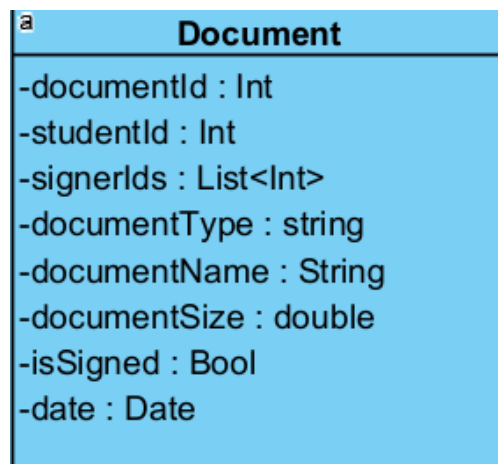


Fig. 3.6.2.16: Document Entity

This class represents the documents in the system.

Attributes:

private int documentId: unique id of document to store in database.

private int studentId: student id that exists in all documents.

private List<int> signerIds: id integer list of the user that has signing authority.

private String documentType: the type of document to store in the database.

private String documentName: document name that is defined by the user to store in the database.

private double documentSize: the size of the document.

private bool isSigned: A boolean if the document is signed or not.

private Date date: The date object which represents the date on which the file is uploaded.

4. Improvements Summary

- In the Boundary Conditions section, Initialization, Failure, and Termination headings are updated. Initial conditions to run the system have been added. Explanation of how the system acts while the termination process has been detailed more.
- Persistent Data Management explanation was changed based on given feedback. Persistent data was explained, and some examples of relations and dependencies on database tables were given.
- Interface layer was changed, and abstraction was used to use the same component for different UIs.
- Instead of Reliability, we now have Increased Productivity as a design goal.
- A deployment diagram has been added.
- More details about Hardware/Software Mapping have been added such as hardware requirements for clients and developers, where the database is kept and where the server is run.
- Missing figure captions have been added.

5. Glossary & References

- [1] By: IBM Cloud Education, "What is three-tier architecture," *IBM*. [Online]. Available:
<https://www.ibm.com/cloud/learn/three-tier-architecture#:~:text=Three%2Dtier%20architecture%20is%20a,associated%20with%20the%20application%20is>. [Accessed: 29-Nov-2022].
- [2] "Singleton," *Refactoring.Guru*. [Online]. Available:
<https://refactoring.guru/design-patterns/singleton>. [Accessed: 29-Nov-2022].
- [3] "Template method design pattern," *GeeksforGeeks*, 18-Oct-2021. [Online]. Available: <https://www.geeksforgeeks.org/template-method-design-pattern/>. [Accessed: 29-Nov-2022].
- [4] "Web Browser System Requirements Overview," *Ghacks*, 21-May-2012. [Online]. Available:
<https://www.ghacks.net/2012/05/21/web-browser-system-requirements-overview/>. [Accessed: 28-Nov-2022].
- [5] "What are the hardware and software requirements to run a website?," *Quora*. [Online]. Available:
https://www.quora.com/What-are-the-hardware-and-the-software-requirements-to-run-a-website/answer/Rohit-K-Choudhary?ch=10&oid=101070104&share=4cf084b1&srid=lilzw&target_type=answer. [Accessed: 28-Nov-2022].