Bilkent University

Department of Computer Engineering

CS319 Term Project

Section 1

Group 1A - "lobby"

Project short-name: Pandemica

Design Report

Group Members:

- Efe Beydoğan (21901548)
- Arda Önal (21903350)
- Mert Barkın Er (21901645)
- Emir Melih Erdem (21903100)
- Gökberk Beydemir (21902638)
- Eren Polat (21902615)

Introduction **3**
Purpose of the System 3
Design Goals 3
User-Friendliness / Usability 3
Reliability / Security 4
Maintainability / Supportability 4
Performance 4

High Level Software Architecture **5**
Subsystem Decomposition 5
Subsystem Explanations 6
Deployment Diagram 10
Hardware/Software Mapping 11
Persistent Data Management 12
Access Control and Security 12
Boundary Conditions 14
Initialization 14
Termination 15
Failure 15

Low-Level Design **15**
Object Design Trade-Offs 15
Maintainability versus Performance 15
Memory versus Performance 15
Security versus Performance 15
Functionality versus Usability 16
Layers 16
User Interface Layer 16
Entity-Controller Diagram (Final Object Design Diagram) 25
Database Layer Repository Class Diagram 51

Design Patterns **52**

Packages and Frameworks **55**

Introduction

Purpose of the System

Our project is designed as a software that will ease everyone's lives in the university during difficult pandemic times. The project will be a medium for students, lecturers, TAs and health personnel to get together during rough times to inform each other of the latest developments regarding the pandemic and take precautions together. The web-based application will provide easy access to resources a person may need during tough pandemic times. These resources include the personal information of the user, as well as weekly reports, guidelines specified by the university, important announcements, statuses of other people in the user's registered courses, and much more. Also, by adding a feedback form we are hoping to receive feedback from the users about the app and increase service quality. The project will include a request form, which users can fill out and send to the university to make suggestions on pandemic related precautions (addition of disinfectants to certain places etc.). Our goal when developing this product is to save the inhabitants of the university from the hassle of finding every bit of information separately on the internet, and let everyone have easy access to crucial information.

Design Goals

The application is supposed to be a user-friendly system, with an easy to use interface. Our goal when developing this software is to ensure it's usability, reliability, security, supportability and maintainability. Also, as we have stated in our Analysis Report, there are a number of performance constraints we strive to fulfill, in order to deliver the best experience for the users of our application. Top two design goals we value the most are usability and security.

User-Friendliness / Usability

We want to make sure that even users who aren't familiar with using computers can find their way around our website, so we will make our app as usable as possible. To this end, we will try our best to let any user who knows how to read and write use the system without a user manual. All the buttons will be aligned and have self-explanatory titles. Every button that is visible to the user will be clickable.

Reliability / Security

The information that is displayed on the site should always be correct and a system crash should not result in any data loss to ensure the reliability of the system. Also, reports that the user generates should not be lost due to internet related issues etc. The system should not crash in input errors and warn the user that the input is incorrect.

The system will have confidential information of every user such as passwords, phone numbers, Bilkent ids, email etc. and this information should not be accessible to anyone who tries to obtain them. Any functionality of the website should work regardless of cybersecurity attacks.

Maintainability / Supportability

The website should be supported for Google Chrome version 68.x or higher, Opera version 80.x or higher and Microsoft Edge version 93.x or higher. The locations of labels and buttons should not change for every operating system and browser. The website should work on devices that have 2 GB RAM or higher. The database should be able to store data that is up to 1 T.

We will implement the system as neatly as possible to make it supportable, so future developers can easily build up on it and add new features. We aim for any changes to be easily added after deployment.

Performance

We will try to make the website as efficient as possible, performance wise. Thus, the users will be able to use it without running into problems very often and they will hopefully have a smooth experience.

Some performance constraints include:

- The system website should load in less than 2 seconds.
- Every user interaction should be processed and if there is output, the result should be displayed in less than 2 seconds.
- The time that takes for the website to retrieve information from the database should be less than 1 second.
- The HES code should be revalidated every 30 seconds.
- If there is a change in the covid status of a user, the change should be represented in less than 2 seconds.

- The course and section information should be validated every 30 seconds and if there is any change, it should be displayed in less than 2 seconds.
- The entire system data should be backed up every hour.

High Level Software Architecture

Subsystem Decomposition

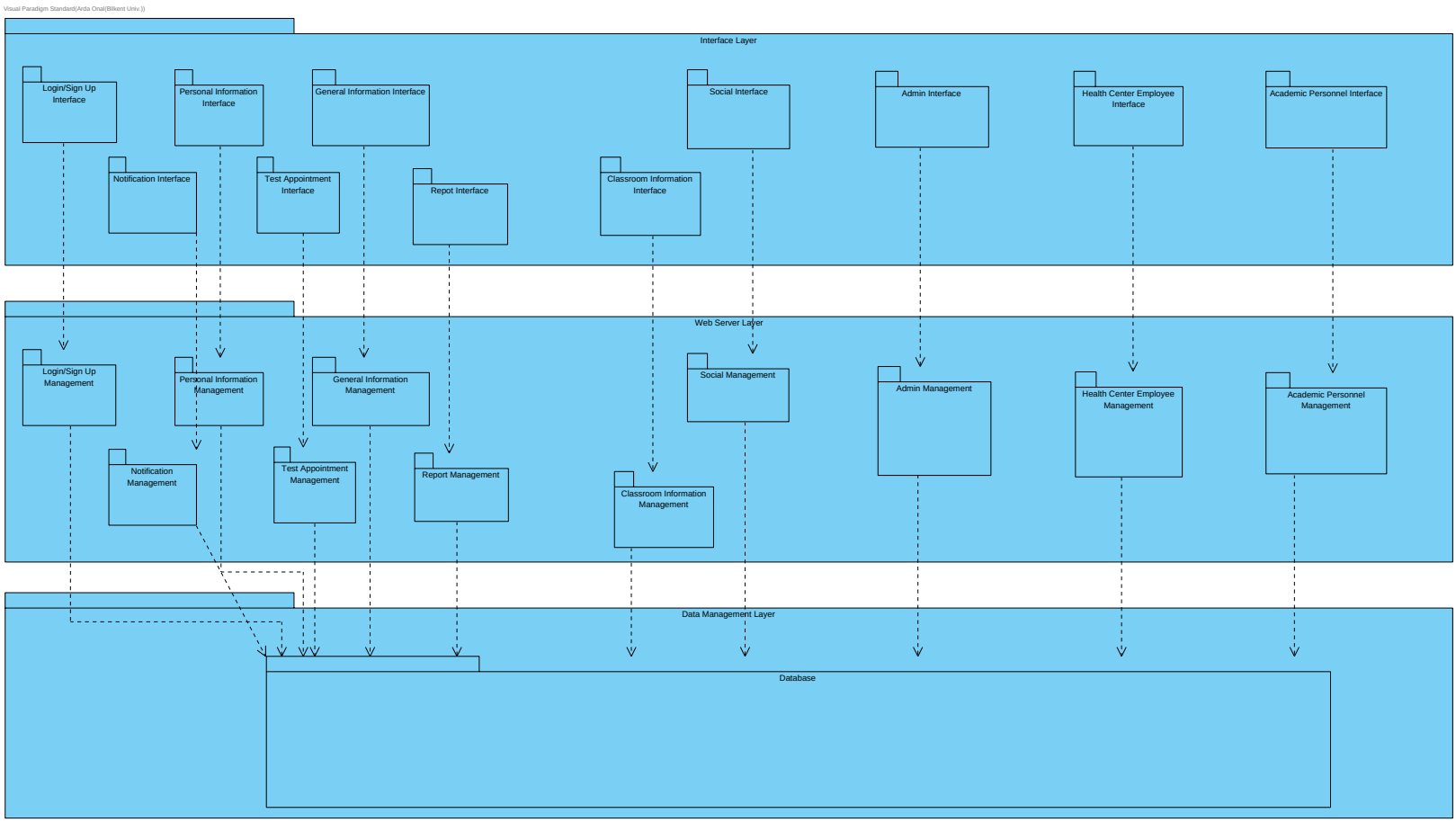

Fig. 1: Subsystem Decomposition Diagram

In the given Subsystem Decomposition Diagram, “Interface Layer” corresponds to the User Interface of our website. This layer constitutes the boundary objects of our project. All of the systems in this layer depict the UI elements users can interact with. We have decided these systems based on our use cases and functionalities we intend on offering to the users of our software. All of these UI elements will be designed according to our “User-Friendliness / Usability” design goal, and we will make sure they meet the standards for easy usage.

The middle “Web Server Layer” contains all the systems for managing systems in the “Interface Layer”. All of these operations will be done in the backend. For example, the “Personal Information Management” system will fetch the information of the user from the database in order to be displayed correctly in the frontend to the user. All of the systems in this layer access the database to fetch information for the users of the software and fulfill their needs. The separation of core functionalities allow for a maintainable and supportable system so version changes and/or developer changes are more easily accommodated.

The final and bottom layer is the “Data Management Layer”. The database will be used to store every important information such as login credentials, vaccination cards, user information etc. This helps with our Performance and Reliability/Security criteria since the database lets us securely store data and quickly fetch it whenever needed.

Subsystem Explanations

Interface & Web Server Layers

Login/Sign Up Interface - Login/Sign Up Management:

Login/Sign Up Interface subsystem contains the login and signup functionalities for the Pandemica website. These functionalities behave the same for every user in the system and the same screen is displayed to every member of the site.

The Login/Sign Up Management subsystem maintains the login and signup functionalities for the site, and fetches necessary information from the database to either log users in or let them sign up to the site. The usernames/emails and passwords of every user are stored in the database to validate their login credentials every time they try to log in.

Notification Interface - Notification Management:

Notification Interface subsystem contains the screen where every user can view their current and old notifications. Any user can either view or delete their own notifications from this screen.

The notification management subsystem is responsible for fetching and displaying notifications whenever the user tries to access them. Also, when a user deletes one of their notifications, this subsystem deals with removing that notification from their screen and disposes of the information contained in the particular notification.

Personal Information Interface - Personal Information Management:

The Personal Information Interface subsystem consists of the screens where users can view their personal information. This information includes the users' Bilkent ID, TC ID/Passport Number, address, phone number, email etc. Also, this screen is where all COVID related information about the individual user can be found, including tests, sickness status and vaccination information/vaccination card.

Personal Information Management subsystem communicates with the Database in the Data Management Layer to receive the stored information about users and relays it to the personal information interface for displaying.

Test Appointment Interface - Test Appointment Management:

The Test Appointment Interface is the screen where every user registered in the system can make a Diagnovir test appointment to be carried out at Bilkent University.

Test Appointment Management subsystem deals with the correct making and relaying of test appointments. Meaning, this subsystem checks for availability to be reflected to the user so users can select among the vacant spots for test appointments. Also, whenever a new appointment is made, the subsystem is responsible for relaying this information to the Health Center Employees so they are aware.

General Information Interface - General Information Management:

The General Information Interface includes the screen where every user can view university wide stats regarding the pandemic, as well as guidelines set by the university, weekly reports created by the system and the announcements made by the university administration.

General Information Management subsystem fetches all of the general information about the pandemic from the database to show to the users, and whenever one of the fields in this screen, such as stats of guidelines, is updated, the management subsystem updates the field accordingly.

Report Interface - Report Management:

Report Interface is the screen where all three report types in the Pandemica system are available for every user to fill in: Violation Report, Request Form and Feedback Form.

Report Management subsystem is tasked with forwarding reports to users who will be viewing them, as well as fetching previous reports filled in by the user from the system so the user can see their previous reports whenever they want. The Violation Reports and Request Forms must be forwarded to the university administration and/or the admins for their evaluation, while the Feedback Forms are forwarded to the maintainers/creators of the application. These tasks and the saving of the reports to the database are carried out by the management subsystem for the reports.

Classroom Information Interface - Classroom Information Management:

Classroom Information Interface is the screen where Student users are able to select their seats in every one of their courses. The interface includes a grid for every classroom where the students can simply select their seat.

The Classroom Information Management subsystem is responsible for determining a particular student’s neighbors in their courses once the student makes a selection for their seat. This subsystem also keeps track of every user’s neighbors and sends the necessary notifications to their neighbors and instructors, should a person test positive for COVID. All of this data is stored in and fetched from the database, which the management subsystem communicates with.

Social Interface - Social Management:

Social Interface is the screen for Student users where they can add friends and view their friends' sickness statuses.

The Social Management subsystem keeps track of a user's friends and communicates with the database to store information such as a user's added friends as well as their outgoing/rejected/accepted friend requests.

Admin Interface - Admin Management:

The Admin Interface contains only the admin specific pages that only Admin users have access to. These pages include the pages for handling all kinds of reports filled in and sent by the users, as well as updating/creating seating plans for classrooms, viewing the neighbor information of every Student registered to the system, and updating announcements, guidelines, stats, weekly reports and sending notifications.

The Admin Management subsystem takes care of every action taken by admins and makes sure admin actions are carried out correctly, communicating with the database when necessary.

Health Center Employee Interface - Health Center Employee Management:

Health Center Employee Interface subsystem includes the screens special to the Health Center employees. Through these screens, health center employees can view and manually update the risk statuses of users, upload physical examination results and enter test results.

The management subsystem for Health Center employees is responsible for communicating with the database to update users' risk statuses and their test results accordingly.

Academic Personnel Interface - Academic Personnel Management:

Academic Personnel Interface is the subsystem where screens specific for instructors are contained. Academic personnel can view their sections' information, and also create/update seating plans if need be.

The Academic Personnel Management subsystem updates the section information continuously so the instructors have access to the most updated

info regarding their classes and students. Also, the seating plans created/updated by the instructors are relayed to the database so other related subsystems have access to this information as well.

Data Management Layer

Database

The database is where all the necessary information on the system is stored for every subsystem to easily access and use whenever needed.

Deployment Diagram

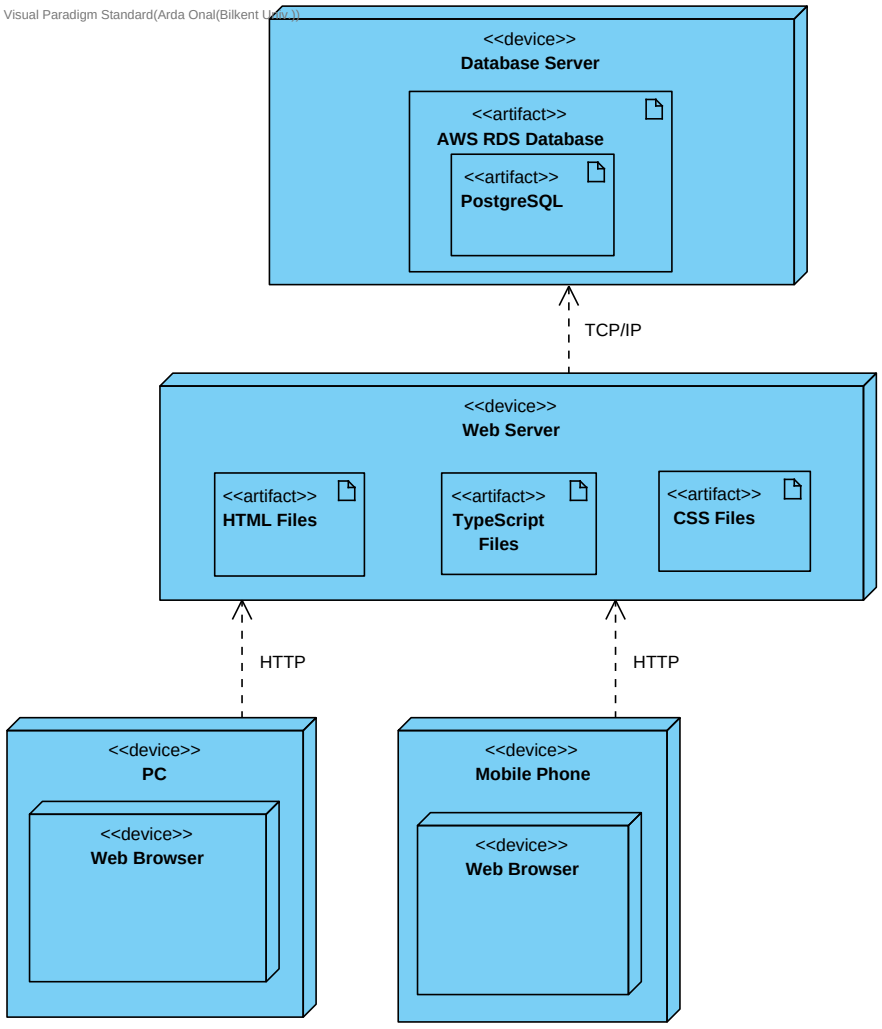

Fig. 2: Deployment Diagram

The PC and Mobile Phone nodes use the following subsystems:

- Login/Sign Up Interface
- Notification Interface
- Personal Information Interface
- Test Appointment Interface
- General Information Interface
- Report Interface
- Classroom Information Interface
- Social Interface
- Admin Interface
- Health Center Employee Interface
- Academic Personnel Interface

Web Server node uses the following subsystems:

- Login/Sign Up Management
- Notification Management
- Personal Information Management
- Test Appointment Management
- General Information Management
- Report Management
- Classroom Information Management
- Social Management
- Admin Management
- Health Center Management
- Academic Personnel Management

Data Server node uses the following subsystems:

- Database

Hardware/Software Mapping

This project is web based which means that supported web browsers are required to use the website. The supported web browsers are Google Chrome version 68.x or higher, Opera version 80.x, Safari 10.x or higher and Microsoft Edge version 93.x or higher as indicated in the Analysis Report. In addition, to host our site 8 GB of RAM and Intel Pentium 4 or later processors will be sufficient. Node.js must be installed on the computer hosting the site since the frontend is built using the Angular framework,

which depends on npm packages. Npm version 8.3.0 is used while building the project. Also, Angular CLI version 13.1.1 and Node version 16.13.1 must be installed. For the backend, Java Spring is used and thus Java 8+ and Spring Framework 5.0.7 release or above is required to get the backend running.

Persistent Data Management

In the implementation of the project, PostgreSQL will be used because some of our team members worked as a backend software engineer part-time in a company and they are the most knowledgeable team members about databases. They are going to lead the backend team and this was his decision. In the database, there will be tables for entities such as students, staff, securities, academic personnel, medical employees, admins, classroom plans, courses, friend requests, neighbors, forms, examination reports, vaccination information, test results, announcements and guidelines. There will also be relational tables that are indicated in the class diagram with many to many relationships. Each table will store the attributes of each entity in the class diagram as database attributes in their table. The primary keys of tables will be selected from their attributes such that it uniquely identifies every tuple in the table for example, the id attribute of the different users etc. Every subsystem in the web server layer will interact with the database subsystem that is located in the data management layer.

Access Control and Security

	Student	Admin	Academic Personnel	Health Center Employee	Security
Login Page	login(id : Integer, password : String) navigateToBilkentStars() navigateToAbout() navigateToFAQ()	login(id : Integer, password : String) navigateToBilkentStars() navigateToAbout() navigateToFAQ()	login(id : Integer, password : String) navigateToBilkentStars() navigateToAbout() navigateToFAQ()	login(id : Integer, password : String) navigateToBilkentStars() navigateToAbout() navigateToFAQ()	login(id : Integer, password : String) navigateToBilkentStars() navigateToAbout() navigateToFAQ()
About Page	navigateToLoginPage() navigateToBilkentStars() navigateToFAQ()	navigateToLoginPage() navigateToBilkentStars() navigateToFAQ()	navigateToLoginPage() navigateToBilkentStars() navigateToFAQ()	navigateToLoginPage() navigateToBilkentStars() navigateToFAQ()	navigateToLoginPage() navigateToBilkentStars() navigateToFAQ()
FAQPage	navigateToLoginPage() navigateToAbout() navigateToBilkentStars()	navigateToLoginPage() navigateToAbout() navigateToBilkentStars()	navigateToLoginPage() navigateToAbout() navigateToBilkentStars()	navigateToLoginPage() navigateToAbout() navigateToBilkentStars()	navigateToLoginPage() navigateToAbout() navigateToBilkentStars()
PersonalInfoPage	getPersonalInfo() getCovidStatus() getNeighborInfo() navigateToClassInfo() getVaccinationInfo() openAddVaccination() getTests()	getPersonalInfo() getCovidStatus() navigateToClassInfo() getVaccinationInfo() openAddVaccination() getTests() navigateToSearchUserInfo()	getPersonalInfo() getCovidStatus() navigateToClassInfo() getVaccinationInfo() openAddVaccination() getTests()	getPersonalInfo() getCovidStatus() getVaccinationInfo() openAddVaccination() getTests() getTestsAppointments() navigateToSearchUserInfo()	getPersonalInfo() getCovidStatus() getVaccinationInfo() openAddVaccination() getTests()
Social Page	getFriends() getRequests() openAddFriend()				
AddFriendPopup	sendFriendRequest(name : String) sendFriendRequest(id : Integer) close()				
AddVaccinationPopup	uploadVaccinationCard() saveVaccinationCard(vaccinationCard : VaccinationCard) cancel()	uploadVaccinationCard() saveVaccinationCard(vaccinationCard : VaccinationCard) cancel()	uploadVaccinationCard() saveVaccinationCard(vaccinationCard : VaccinationCard) cancel()	uploadVaccinationCard() saveVaccinationCard(vaccinationCard : VaccinationCard) cancel()	uploadVaccinationCard() saveVaccinationCard(vaccinationCard : VaccinationCard) cancel()

Sidebar	navigateToPersonalInfo() navigateToClassInfo() navigateToGeneralInfo() navigateToSocial() navigateToReportRequest()	navigateToPersonalInfo() navigateToGeneralInfo() navigateToReportRequest()	navigateToPersonalInfo() navigateToClassInfo() navigateToGeneralInfo() navigateToReportRequest()	navigateToPersonalInfo() navigateToGeneralInfo() navigateToReportRequest()	navigateToPersonalInfo() navigateToGeneralInfo() navigateToReportRequest()
FriendView	getFriendInfo() removeFriend()				
ConfirmationPopup	confirm() cancel()				
RequestView	getRequest() acceptRequest() declineRequest()				
GeneralInfoPage	getUniversityWideStats() getAnnouncements() navigateToGuidelines() navigateToWeeklyReports()	getUniversityWideStats() getAnnouncements() navigateToGuidelines() navigateToWeeklyReports() setUniversityWideStats(universityWideStats : String) addAnnouncement(announcement : String) editAnnouncement() deleteAnnouncement()	getUniversityWideStats() getAnnouncements() navigateToGuidelines() navigateToWeeklyReports()	getUniversityWideStats() getAnnouncements() navigateToGuidelines() navigateToWeeklyReports()	getUniversityWideStats() getAnnouncements() navigateToGuidelines() navigateToWeeklyReports()
WeeklyReportsPage	getWeeklyReports()	getWeeklyReports() setWeeklyReports(weeklyReports : String)	getWeeklyReports()	getWeeklyReports()	getWeeklyReports()
GuidelinesPage	getGuidelines()	getGuidelines() setGuidelines(guidelines : String)	getGuidelines()	getGuidelines()	getGuidelines()
TopNavbar	logOff()	logOff()	logOff()	logOff()	logOff()
SearchBar	searchFunctionality() goToFunctionality()	searchFunctionality() goToFunctionality()	searchFunctionality() goToFunctionality()	searchFunctionality() goToFunctionality()	searchFunctionality() goToFunctionality()
NotificationDropdown	getNotifications()	getNotifications() openSendNotification()	getNotifications()	getNotifications()	getNotifications()
NotificationView	goToNotificationPage() deleteNotification()	goToNotificationPage() deleteNotification()	goToNotificationPage() deleteNotification()	goToNotificationPage() deleteNotification()	goToNotificationPage() deleteNotification()
ClassInfoPage	getCourses()		getCourses()		
CourseSeatView	getCourseName() getSeatNo() getSeatingPlan() openUpdateSeat()	getStudentList()	getCourseName() getSeatingPlan() getStudentList()		
UpdateSeatPopup	getSeatingPlan() selectSeat(seatNo : Integer) saveSeat(seatNo: Integer) cancel()				
ReportRequestPage	navigateToPreviousViolationReports() navigateToPreviousRequestForms() navigateToPreviousFeedbackForms()	navigateToPreviousViolationReports() navigateToPreviousRequestForms() navigateToPreviousFeedbackForms() getAllFormsSubmitted()	navigateToPreviousViolationReports() navigateToPreviousRequestForms() navigateToPreviousFeedbackForms()	navigateToPreviousViolationReports() navigateToPreviousRequestForms() navigateToPreviousFeedbackForms()	navigateToPreviousViolationReports() navigateToPreviousRequestForms() navigateToPreviousFeedbackForms()
ViolationReportView	selectViolationType(type : String) enterViolationMessage(message : String) enterViolationPlace(place : String) submit()	selectViolationType(type : String) enterViolationMessage(message : String) enterViolationPlace(place : String) submit()	selectViolationType(type : String) enterViolationMessage(message : String) enterViolationPlace(place : String) submit()	selectViolationType(type : String) enterViolationMessage(message : String) enterViolationPlace(place : String) submit()	selectViolationType(type : String) enterViolationMessage(message : String) enterViolationPlace(place : String) submit()
RequestFormView	enterTitle(title : String) enterRequest(request : String) submit()	enterTitle(title : String) enterRequest(request : String) submit()	enterTitle(title : String) enterRequest(request : String) submit()	enterTitle(title : String) enterRequest(request : String) submit()	enterTitle(title : String) enterRequest(request : String) submit()
FeedbackFormView	rate(rating : Integer) enterTitle(title : String) enterFeedback(feedback : String) submit()	rate(rating : Integer) enterTitle(title : String) enterFeedback(feedback : String) submit()	rate(rating : Integer) enterTitle(title : String) enterFeedback(feedback : String) submit()	rate(rating : Integer) enterTitle(title : String) enterFeedback(feedback : String) submit()	rate(rating : Integer) enterTitle(title : String) enterFeedback(feedback : String) submit()
SearchUserInfoPage		getUsers()		getUsers()	
UserInfoView		getPersonalInfo() getPersonalInfo(userInfo : String) getCovidStatus() setCovidStatus(covidStatus : integer) getVaccinationInfo() enterTestResult(testResult : integer) deleteUser()		getPersonalInfo() getPersonalInfo(userInfo : String) getCovidStatus() setCovidStatus(covidStatus : integer) getVaccinationInfo() enterTestResult(testResult : integer) deleteUser()	
SendNotificationPopup		addReceiver(receiver : String) setTitle(title : String) setMessage(message : String) sendNotification() cancel()			

Fig. 3: Access Control Matrix

We promise absolute security and reliability in our software. For example, the data that is displayed to the users should always be correct and dependable data, based on official numbers announced by the university. Also, users' personal information will be under protection through

the database. In order to ensure access control, we have different roles in our site and all of these roles are able to access different functionalities.

All of the user types are able to login/sign up, view their notifications, personal information, test appointments and general information regarding the situation of the pandemic in the university. Also, every user type is allowed to send feedback, violation and request reports.

In addition, student roles have access to the social and classroom interfaces. Students are able to add their friends on the site to view their friends' situations as well. Also, students have the ability to enter the seat selection interface, where they can select the seats they are sitting at in every one of their sections. The system will automatically determine their neighbors using this information, and the students will be saved from the hassle of learning the names/ids of all of their neighbors in their classrooms.

Admins have the access to the "Admin Interface" as we put it. This interface includes the functionalities of sending notifications, viewing every user's info, editing the general information about the university such as case numbers etc. and viewing every report created by other users and reviewing them.

Health Center employees can view an interface specifically designed for them, where they can view test appointments made by users, enter test results, upload physical examination reports and manually update users' risk statuses if need be.

Finally, Academic Personnel also have access to a special interface where they can view their sections' information, such as who is allowed in the class or not, as well as seating plans according to the input from the students when they select which seat they are sitting at.

Boundary Conditions

Initialization

The app will be deployed through AWS, from which it will be hosted. The project will be compiled into a jar file, and then loaded into AWS. The database of the project will be hosted on AWS and the backend will be connected to that database, instead of a local database.

Termination

The website subsystems work altogether which means that if a single subsystem is terminated, the whole application terminates. If an admin terminates any subsystem, all data that is being used in that subsystem created by the users are saved to the database.

Failure

The intended failures caused by the developers will return different HTTP codes depending on the situation. Front-end will show a pop-up describing the situation and why the operation failed (e.g. “This student is already your friend so you cannot send a friend request to them.”). Our project will be hosted on AWS, the unintended failures will be handled by AWS.

Low-Level Design

Object Design Trade-Offs

Maintainability versus Performance

The application will be coded using OOP, therefore will have objects for almost every portion of the project. While this will make performance a bit lower, maintainability will be better since developers will be able to make changes easier, saving developers hours that they would spend trying to understand the system.

Memory versus Performance

Since the project will be coded using OOP, there will be a lot of memory usage caused by the amount of classes. However, AWS has memory optimization that lowers the memory usage which we will be using, increasing our memory optimization at the cost of performance.

Security versus Performance

The application will use JSON Web Tokens for authorization to preauthorize functionalities based on the user’s type. The tokens will be generated per user when that user logs in, and it will be stored in the local storage and sent whenever a user tries to use a functionality. This increases security, making sure that the users can only do things they are allowed to. However, checking a user can do an action every single time they try to make an action decreases performance.

Functionality versus Usability

Our project is supposed to be used by everyone in Bilkent University, ranging from the professors to the non-academic staff. This forces the project to have many functionalities that are not available for everyone since there are many user types with many different user authorities, which decreases the usability of the software.

Layers

User Interface Layer

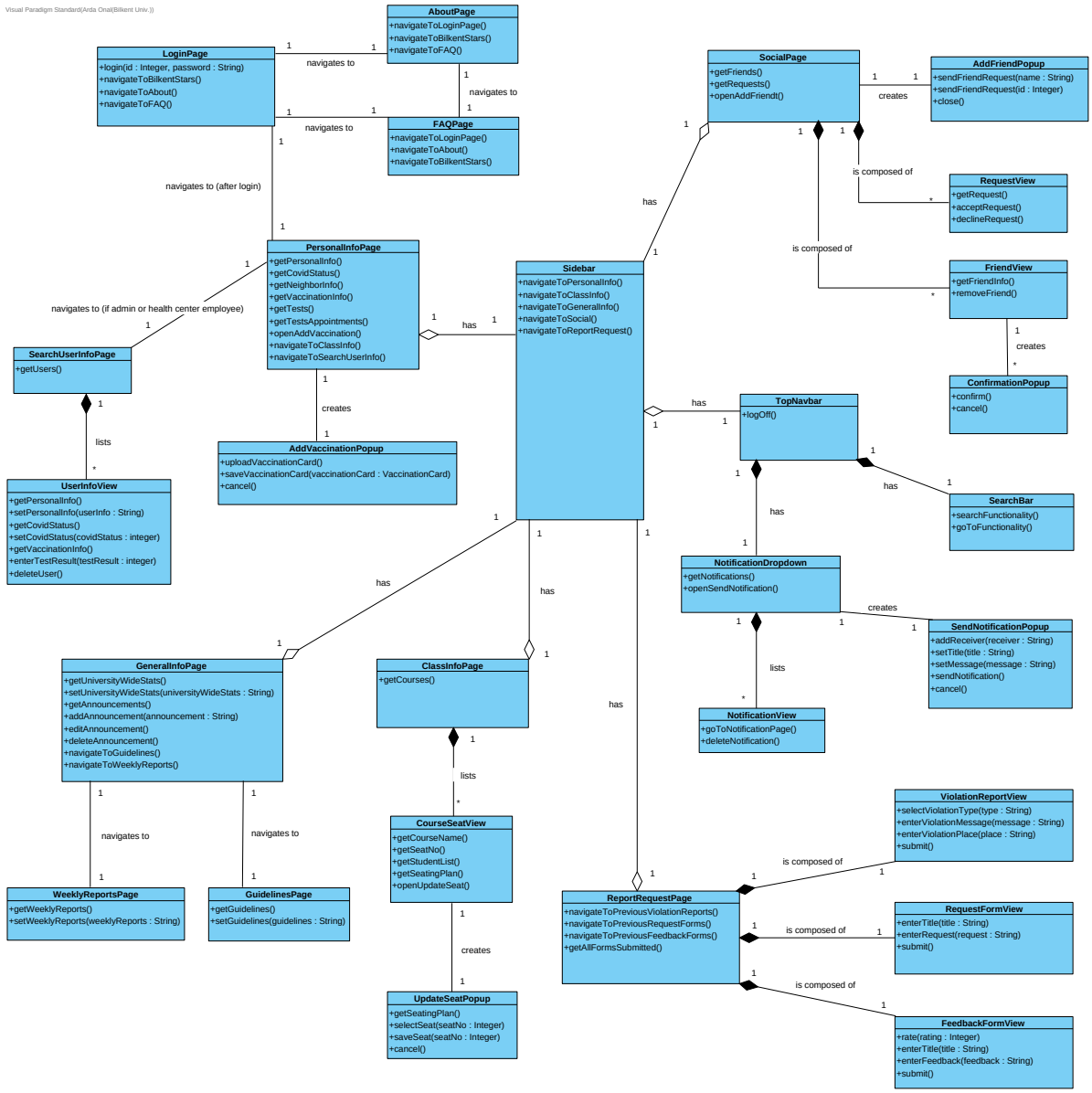

Fig. 4: User Interface Layer Class Diagram

User Interface Class Explanations

LoginPage

The LoginPage class constructs the view for the login page, which is the initial page if the user is not logged in. Users may choose to log in or navigate to other pages that do not require authentication.

Methods:

login(id : Integer, password : String): Tries to login by sending the inputted credentials to the backend.

Other methods are used to navigate to Bilkent Stars webpage, About page, or FAQ page.

AboutPage

The AboutPage class constructs the view for the about page, which contains information on the project and the team.

Methods:

Methods are used to navigate to the Login page, Bilkent Stars webpage, or FAQ page.

FAQPage

The FAQPage class constructs the view for the FAQ page, which contains frequently asked questions and their answers.

Methods:

Methods are used to navigate to the Login page, About page, or Bilkent Stars webpage.

Sidebar

The Sidebar class constructs the view for the sidebar that is used to navigate through the main pages of the application. It is contained in all pages that require authentication.

Methods:

Methods are used to navigate to the Personal Info page, Class Info page, General Info page, Social page, or ReportRequest page.

TopNavbar

The TopNavbar class constructs the view for the top navbar that contains the search functionality bar, notification bell, and the log off button. It is contained in all pages that require authentication.

Methods:

logOff(): Logs off.

SearchBar

The SearchBar class constructs the view for the search bar used to search for a specific functionality in the application.

Methods:

searchFunctionality(): Displays the results of the functionality search.

goToFunctionality(): Navigates to the page that contains the functionality. Activated when clicked on a search result.

NotificationDropdown

The NotificationDropdown class constructs the view for the notification dropdown that is used to view the list of notifications received.

Methods:

getNotifications(): Gets the notifications received.

openSendNotification(): Creates the Send Notification Popup. Activated when clicked on the Send Notification button. Specific to admin users.

NotificationView

The NotificationView class constructs the view for a single notification item in the notifications list.

Methods:

goToNotificationPage(): Navigates to the page related to the notification. Activated when clicked on the notification item.

deleteNotification(): Deletes the notification item from received notifications.

SendNotificationPopup

The SendNotificationPopup class constructs the view for the send notification popup that is used by admin users to send notifications.

Methods:

addReceiver(receiver : String): Used to enter the receiver of the notification.

setTitle(title : String): Used to set the title of the notification.

setMessage(message : String): Used to set the message of the notification.

sendNotification(): Used to submit the notification and send it to the backend.

cancel(): Closes the popup.

PersonalInfoPage

The PersonalInfoPage class constructs the view for the personal info page, which is the initial page if the user is logged in. Contains information on the personal information of the user including covid status, neighbor info, vaccination info, and tests.

Methods:

getTestsAppointments(): Gets the list of test appointments. Specific to the Health Center Employee role.

openAddVaccination(): Creates the Add Vaccination Popup. Activated when clicked on the Add Vaccination button.

navigateToClassInfo(): Navigates to the Class Info page. Activated when clicked on the Update Seats button.

navigateToSearchUserInfo(): Navigates to the Search User Info page. Specific to the Admin and Health Center Employee roles.

Other methods are trivial getter methods for displayed information.

AddVaccinationPopup

The AddVaccinationPopup class constructs the view for the add vaccination popup that is used to upload a vaccination file.

Methods:

uploadVaccinationCard(): Used to upload a vaccination file from the device.

saveVaccinationCard(vaccinationCard : VaccinationCard): Sends the uploaded vaccination card to the backend.
cancel(): Closes the popup.

SearchUserInfoPage

The SearchUserInfoPage class constructs the view for the search user info page, which contains the list of users registered to the system and their information. It is specific to the Admin and Health Center Employee roles.

Methods:
Trivial getter method for displayed information.

UserInfoView

The CourseSeatView class constructs the view for a single user item in the taken users list. The displayed information and operations are distinct for the Admin and Health Center Employee roles.

Methods:
enterTestResult(testResult : integer): Used by health center employees to add the test result of the user.
deleteUser(): Used by admins to delete the user from the system.
Other methods are trivial getter/setter methods for displayed information.

GeneralInfoPage

The GeneralInfoPage class constructs the view for the general info page, which contains information on the university wide statistics, announcements, and navigation options to the guidelines and weekly reports. Admin users can modify the information displayed.

Methods:
Methods to add, edit, and delete announcements. Specific to admin users.
navigateToGuidelines(): Navigates to the Guidelines page.
navigateToWeeklyReports(): Navigates to the Weekly Reports page.
Other methods are trivial getter methods for displayed information.

WeeklyReportsPage

The WeeklyReportsPage class constructs the view for the weekly reports page, which contains the weekly reports of university wide statistics. Admin users can modify the information displayed.

Methods:

Trivial getter/setter method for displayed information.

GuidelinesPage

The GuidelinesPage class constructs the view for the guidelines page, which contains the guidelines. Admin users can modify the information displayed.

Methods:

Trivial getter/setter method for displayed information.

ClassInfoPage

The ClassInfoPage class constructs the view for the class info page, which contains the list of courses taken/given and seat information.

Methods:

Trivial getter method for displayed information.

CourseSeatView

The CourseSeatView class constructs the view for a single course-seat item in the taken/given courses list.

Methods:

getStudentList(): Gets the list of students in the course, along with their Covid statuses. Specific to the Academic Personnel role.

openUpdateSeat(): Creates the Update Seat Popup. Activated when clicked on the Update Seat button. Specific to the Student role.

Other methods are trivial getter methods for displayed information.

UpdateSeatPopup

The UpdateSeatPopup class constructs the view for the update seat popup that is used to select and update the seat in a course.

Methods:

selectSeat(seatNo : Integer): Used to select the seat with the given seat no from the seating plan.

saveSeat(seatNo : Integer): Sends the selected seat no to the backend.

cancel(): Closes the popup.

Other methods are trivial getter methods for displayed information.

SocialPage

The SocialPage class constructs the view for the social page, which contains information on the user's friends and friend requests.

Methods:

openAddFriend(): Creates the Add Friend Popup. Activated when clicked on the Add Friend button.

Other methods are trivial getter methods for displayed information.

FriendView

The FriendView class constructs the view for a single friend item in the friends list.

Methods:

removeFriend(): Creates the Confirmation Popup for removing the friend item from the friends list. Activated when clicked on the Remove button.

Other methods are trivial getter methods for displayed information.

ConfirmationPopup

The ConfirmationPopup class constructs the view for the confirmation popup that is used to confirm the friend removal.

Methods:

confirm(): Confirms the friend removal.

cancel(): Closes the popup.

RequestView

The RequestView class constructs the view for a single friend request item in the pending requests list.

Methods:

acceptRequest(): Accepts the friend request item.
declineRequest(): Declines the friend request item.
Other methods are trivial getter methods for displayed information.

AddFriendPopup

The AddFriendPopup class constructs the view for the add friend popup that is used to send a new friend request.

Methods:

sendFriendRequest(name : String): Sends a friend request to the user with the specified name, if it exists.
sendFriendRequest(id : Integer): Sends a friend request to the user with the specified id, if it exists.
close(): Closes the popup.

ReportRequestPage

The ReportRequestPage class constructs the view for the report-request page, which is composed of three form views. Used to submit a violation report, request form, or feedback form. Admins can view all the forms submitted to the system.

Methods:

navigateToPreviousViolationReports(): Views the previous violation reports.
navigateToPreviousRequestForms(): Views the previous request forms.
navigateToPreviousFeedbackForms(): Views the previous feedback forms.
getAllFormsSubmitted(): Gets all the forms submitted to the system. Specific to the Admin role.

ViolationReportView

The ViolationReportView class constructs the view for violation report. It is in a form layout.

Methods:

selectViolationType(type : String): Used to select the violation type.
enterViolationMessage(message : String): Used to enter the violation message.
enterViolationPlace(place : String): Used to enter the violation place.
submit(): Submits the form and sends it to the backend.

RequestFormView

The RequestFormView class constructs the view for request form. It is in a form layout.

Methods:

enterTitle(title : String): Used to enter the request title.
enterRequest(request : String): Used to enter the request.
submit(): Submits the form and sends it to the backend.

FeedbackFormView

The FeedbackFormView class constructs the view for feedback form. It is in a form layout.

Methods:

rate(rating : Integer): Used to give a rating out of five.
enterTitle(title : String): Used to enter the feedback title.
enterFeedback(feedback : String): Used to enter the feedback.
submit(): Submits the form and sends it to the backend.

Entity-Controller Diagram (Final Object Design Diagram)

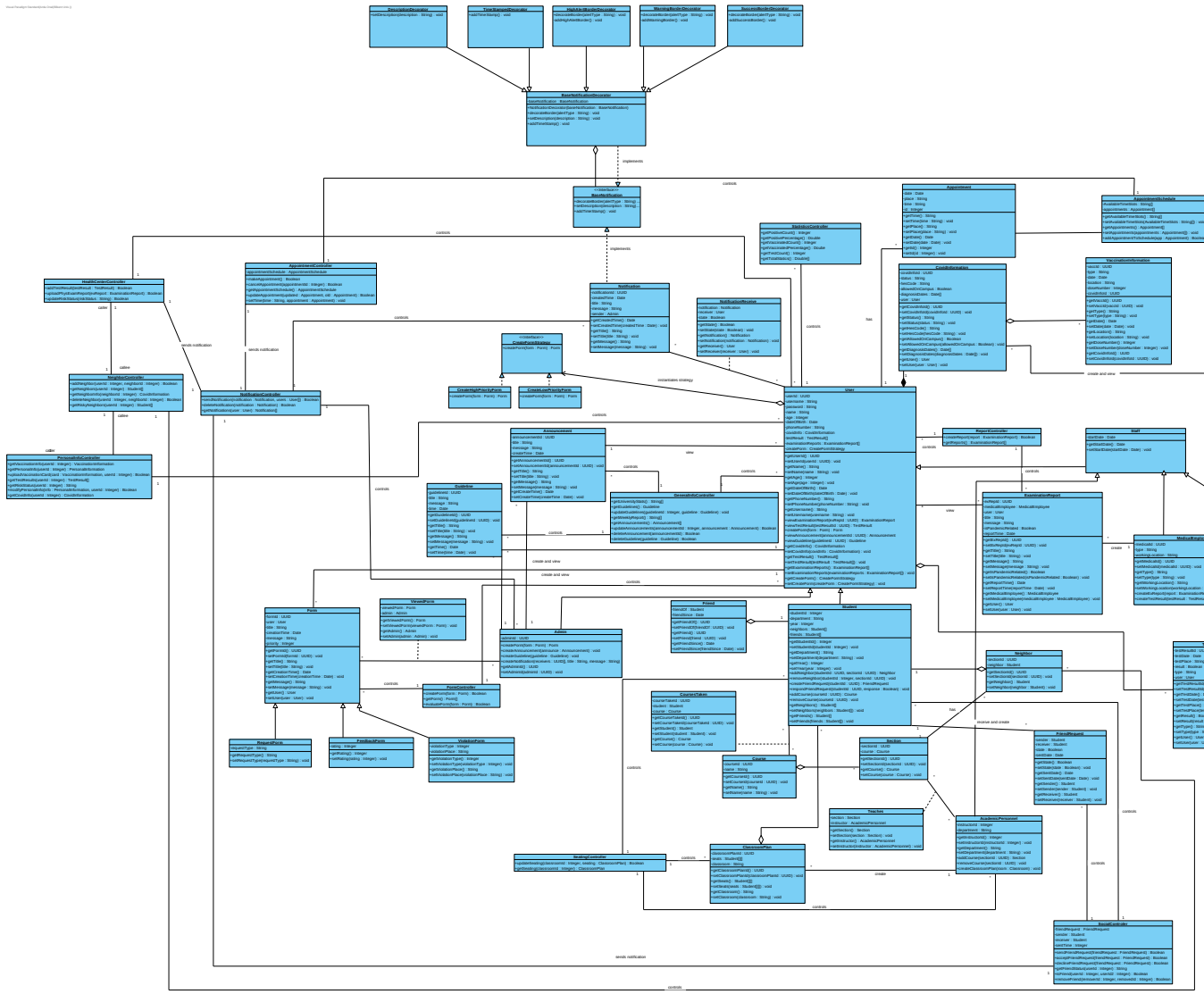

Fig. 5: Entity-Controller Diagram

Entity Class Explanations

Notification

The Notification class represents the notifications that will be sent to the users when for example one of their neighbors becomes positive. Admins or the system can send notifications and Users can receive notifications.

Attributes:

UUID notificationId: Represents the primary key of the Notification table in the Database.
Date createdTime: Represents the creation time of the Notification.
String title: Represents the title of the Notification.
String message: Represents the message of the Notification.
Admin sender: Represents the sender of the Notification. If the system sends the notification, the value is set to Null.

Methods:

All the methods of this class are trivial getter and setter methods.

NotificationReceive

The notificationReceive class is used to represent which Users receive which Notifications. In the database there will be a table to represent the many to many relation between the User and the Notification tables. This class will be used to instantiate objects from the NotificationReceive table.

Attributes:

Notification notification: Represents a notification that is being received by a specific User.
User receiver: Represents a specific User that is receiving a notification.
Boolean state: If the notification is read, the state is true else it is false.

Methods:

All the methods of this class are trivial getter and setter methods.

User

The User class is used to represent the users in the project.

Attributes:

UUID userId: Represents the primary key of the User table in the database.
String username: Represents the username of the User.
String password: Represents the password of the User.
String name: Represents the name of the User.
Integer age: Represents the age of the User.
Date dateOfBirth: Represents the date of birth of the User.
String phoneNumber: Represents the phone number of the User.
CovidInformation covidInfo: Represents the covid information of that User.
TestResult[] testResult: Represents all of the test results this User had.
ExaminationReport[] examinationReports: Represents all of the examination reports this User had.

Methods:

viewExaminationReport(UUID exRepId): Views a specific examination report of this user with the UUID exRepId. Returns an ExaminationReport.
viewTestResult(UUID testResultId): Views a specific test result of this user with the UUID testResultId. Returns a TestResult.
createForm(Form form): Creates a form with the form object. Returns a Form. Implementation depends on the create form strategy.
viewAnnouncement(UUID announcementId): Views an announcement with the UUID announcementId. Returns an Announcement.
viewGuideline(UUID guidelineId): Views a Guideline with the UUID guidelineId. Returns a Guideline.
This class also has the trivial getter/setter methods for its attributes.

CreateFormStrategy <<interface>>

The CreateFormStrategy interface is used to represent the create form strategy, which is implemented differently from role to role. Students can

have the low priority create form behavior whereas other roles have the one with the higher priority. Designed according to the strategy pattern.

Methods:

createForm(Form form): Creates a form with the form object. Returns a Form. (Abstract)

CreateHighPriorityForm

The CreateHighPriorityForm class is used to implement the create high priority form behavior. Implements the CreateFormStrategy class. Users except students use this create form behavior. Designed according to the strategy pattern.

Methods:

createForm(Form form): Creates a form with the form object. Returns a Form. Algorithm for creating high priority forms.

CreateLowPriorityForm

The CreateLowPriorityForm class is used to implement the create low priority form behavior. Implements the CreateFormStrategy class. Students use this create form behavior. Designed according to the strategy pattern.

Methods:

createForm(Form form): Creates a form with the form object. Returns a Form. Algorithm for creating low priority forms.

Guideline

The Guideline class that represents the guidelines related to the pandemic.

Attributes:

UUID guidelineId: Represents the primary key of a guideline from the guideline database table.

String title: Represents the title of the guideline.

String message: Represents the message of the guideline.

Date time: Represents the date of creation of the guideline.

Methods:

All the methods of this class are trivial getter and setter methods.

Announcement

The Announcement class that represents the announcements related to the pandemic.

Attributes:

UUID announcementId: Represents the primary key of the announcements from the announcement database table.
String title: Represents the title of the announcement.
String message: Represents the message of the announcement.
Date time: Represents the date of creation of the announcement.

Methods:

All the methods of this class are trivial getter and setter methods.

Form

The Form class is the parent of all the form types which includes the common attributes among them.

Attributes:

UUID formId: Represents the primary key of the forms from the form database table.
User user: Represents the user that created the form.
String title: Represents the title of the form.
String message: Represents the message of the announcement.
Date creationTime: Represents the date of creation of the form.
Integer priority: Represents the priority of the form.

Methods:

All the methods of this class are trivial getter and setter methods.

ViewedForm

The ViewedForm class is for the database table that represents which admin has viewed which form. Since the view relation between Form and Admin is many to many, a database table is required to store the relation. The purpose of this class is that different admins will view different forms and when they log back in, we have to store what they have viewed previously.

Attributes:

Form viewedForm: Represents the viewed form of an admin.

Admin admin: Represents the admin that has viewed a form.

Methods:

All the methods of this class are trivial getter and setter methods.

RequestForm

The RequestForm class is the child of the form class which represents the request forms that were created by any user.

Attributes:

String requestType: This attribute is used to classify the requests such as a disinfectantRequest, cleaningRequest, maskRequest etc. The types can be extended if necessary.

Methods:

All the methods of this class are trivial getter and setter methods.

FeedbackForm

The FeedbackForm class is the child of the form class which represents the feedback forms that were created by any user.

Attributes:

Integer rating: A rating from 0 to 5 to rate the application.

Methods:

All the methods of this class are trivial getter and setter methods.

ViolationForm

The ViolationForm class is the child of the form class which represents the violation forms that were created by any user.

Attributes:

String violationType: Represents the type of violation such as not wearing a mask, not obeying the social distance rule etc.

String violationPlace: Represents the violation location.

Methods:

All the methods of this class are trivial getter and setter methods.

Admin

The admin class which represents the admin user type.

Attributes:

UUID adminId: Represents a primary key from the admin database table which is used to identify admin user types.

Methods:

createForm(Form form): Allows the admins to create forms. The method takes a form as a parameter. Returns a Form.

createAnnouncement(Announcement announce): Allows the admins to create announcements. The method takes an announcement as a parameter. Return type is void.

createGuideline(Guideline guideline): Allows the admins to create guidelines. The method takes a guideline as a parameter and the return type is void.

createNotification(UUID[] receivers, String title, String message): Allows the admins to create notifications using the parameters title and message which will be sent to the parameter receivers which is a User id array. Returns the notification.

This class also has the trivial getter/setter methods for its attributes.

Friend

The friend class that represents the friend statuses of two students.

Attributes:

Student friendOf: Represents the friend of the student.

Date friendSince: Represent the time passed since they became friends.

Methods:

All of the methods of this class are trivial getters/setters.

CoursesTaken

Represents the courses taken by a student.

Attributes:

UUID courseTakenId: Represents the primary key of this relation.

Student student: Represent the student who takes the course.

Course course: Represents the course the student takes.

Methods:

All of the methods of this class are trivial getters/setters.

Course

Represents the courses the students take.

Attributes:

UUID courseId: Represents the primary key of a course.

String name: Represents the name of the course.

Methods:

All of the methods of this class are trivial getters/setters.

ClassroomPlan

Represents the seating plan of a classroom.

Attributes:

UUID classroomPlanId: Represents the primary key of a classroom plan.

Student[][] seats: Represents the seating places of the students.

String classroom: Represents the name of the classroom.

Methods:

All of the methods of this class are trivial getters/setters.

CovidInformation

Represents the covid information of a user.

Attributes:

UUID covidInfoId: Represents the primary key of a covid information entity.

String status: Represents the status of a user (e.g. "Healthy").

String hesCode: Represents the HES code of a user.

Boolean allowedOnCampus: Represents if a user is allowed on campus or not.

Date[] diagnosisDates : Represents the date on which the user was diagnosed.

User user: Represents the user who owns this covid information.

Methods:

All of the methods of this class are trivial getters/setters.

Student

Represents a student.

Attributes:

Integer studentId: Represents the Bilkent Id of a student.
String department: Represents the department of a student.
Integer year: Represents the year of a student.
Student[] neighbors: Represents the neighbors of a student.
Student[] friends: Represents the friends of a student.

Methods:

addNeighbor(UUID studentId, UUID sectionId): Creates a neighbor relationship between the current student and the student with UUID studentId in section with UUID sectionId. Returns a Neighbor.
removeNeighbor(UUID studentId, UUID sectionId): Removes a neighbor relationship between the current student and the student with UUID studentId in section with UUID sectionId.
createFriendRequest(UUID studentId): Creates a friend request between this student and the student with UUID studentId. Returns a FriendRequest.
respondFriendRequest(UUID studentId, Boolean response): Responds to a friend request created by the student with UUID studentId.
addCourse(UUID courseId): Adds a course to a student. Returns a Course.
removeCourse(UUID courseId): Removes a course from a student.
This class also has the trivial getter/setter methods for its attributes.

Section

Represents a section of a course.

Attributes:

UUID sectionId: Represents the primary key of a section.
Course course: Represents the course of this section.

Methods:

All of the methods of this class are trivial getters/setters.

Teaches

Represents the relationship between academic personnel and section, where an academic personnel teaches a section.

Attributes:

Section section: Represents the section.

AcademicPersonnel instructor: Represents the instructor of this section.

Methods:

All of the methods of this class are trivial getters/setters.

VaccinationInformation

Represents the vaccination information of a user.

Attributes:

UUID vaccId: Represents the primary key of VaccinationInformation.

String type: Represents the name of the vaccination.

Date date: Represents the date of the vaccination.

String location: Represents the location of the vaccination.

Integer doseNumber: Represents how many shots this user had.

UUID covidInfoId: Represents the UUID of the covid information entity which vaccination information is connected to.

Methods:

All of the methods of this class are trivial getters/setters.

Staff

Represents the staff working at Bilkent.

Attributes:

Date startDate: Represents the start date of this staff.

Methods:

All of the methods of this class are trivial getters/setters.

ExaminationReport

Represents the examination report a medical employee creates after an examination.

Attributes:

UUID exRepId: Represents the primary key of the examination report.

MedicalEmployee medicalEmployee: Represents the medical employee who created the examination report.
User user: Represents the user who had the examination.
String title: Represents the title of the examination report.
String message: Represents the message of the examination report.
Boolean isPandemicRelated: Represents if the examination report is related to COVID or not.
Date reportTime: Represents the time of the examination report.

Methods:
All of the methods of this class are trivial getters/setters.

Neighbor

Represents the neighbor relationship between two students.

Attributes:
UUID sectionId: Represents the section's id where the two students are neighbors.
Student neighbor: Represents the neighboring student.
Methods:
All of the methods of this class are trivial getters/setters.

FriendRequest

Represents the friend request a student sends to another student.

Attributes:
Student sender: Represents the student sending the request.
Student receiver: Represents the student receiving the request.
Boolean state: Represents the request's state.
Date sentDate: Represents the time at which the request was sent.

Methods:
All of the methods of this class are trivial getters/setters.

AcademicPersonnel

Represents the academic personnel at Bilkent.

Attributes:
Integer instructorId: Represents the instructors id.
String department: Represents the department of the instructor.

Methods:

addCourse(UUID sectionId): Adds a section to an instructor. Returns a Section.

removeCourse(UUID sectionId): Removes a section from an instructor.

createClassroomPlan(Classroom room): Creates a classroom plan for a section.

This class also has the trivial getter/setter methods for its attributes.

MedicalEmployee

Represents the medical employees working at the Health Center.

Attributes:

UUID medicalId: Represents the Id of the medical employees.

String type: Represents the type of the medical employee.

String workingLocation: Represents the location of the medical employee.

Methods:

createExReport(ExaminationReport report): Creates an examination report. Returns ExaminationReport.

createTestResult(TestResult testResult): Creates a test result. Returns TestResult.

This class also has the trivial getter/setter methods for its attributes.

TestResult

Represents the test result of a user.

Attributes:

UUID testResultId: Represents the primary key of TestResult.

Date testDate: Represents the date of the result.

String testPlace: Represents the place of the test.

Boolean result: Represents the result of the test.

String type: Represents the type of the test.

User user: Represents the user who had the test.

Methods:

All of the methods of this class are trivial getters/setters.

Security

Represents the security staff at Bilkent.

Attributes:

UUID securityId: Represents the primary key of a security.
String type: Represents the type of the security.

Methods:

All of the methods of this class are trivial getters/setters.

BaseNotificationDecorator

Represents the base notification decorator for notifications.

Attributes:

BaseNotification baseNotification: Represents the wrapped base notification.

Methods:

decorateBorder(String alertType): Decorates the border of the notification based on the alert type.
setDescription(String description): Sets the description of the notification.
addTimeStamp(): Adds a timestamp to the notification.

DescriptionDecorator

Represents the description decorator for notifications.

Attributes:

Methods:

setDescription(String description): Sets the description of the notification.

TimeStampedDecorator

Represents the time stamped decorator for notifications.

Attributes:

Methods:

addTimeStamp(): Adds a timestamp to the notification.

HighAlertBorderDecorator

Represents the high alert border decorator for notifications.

Attributes:

Methods:

decorateBorder(String alertType): Decorates the border of the notification based on the alert type.

addHighAlertBorder(): Adds a high alert border to the notification.

WarningBorderDecorator

Represents the warning border decorator for notifications.

Attributes:

Methods:

decorateBorder(String alertType): Decorates the border of the notification based on the alert type.

addWarningBorder(): Adds a warning border to the notification.

SuccessBorderDecorator

Represents the success border decorator for notifications.

Attributes:

Methods:

decorateBorder(String alertType): Decorates the border of the notification based on the alert type.

addSuccessBorder(): Adds a success border to the notification.

HighAlertBorderDecorator

Represents the high alert border decorator for notifications.

Attributes:

Methods:

decorateBorder(String alertType): Decorates the border of the notification based on the alert type.

addHighAlertBorder(): Adds a high alert border to the notification.

BaseNotification

Represents the wrapped notification class.

Attributes:

Methods:

decorateBorder(String alertType): Decorates the border of the notification based on the alert type.

addTimeStamp(): Adds a timestamp to the notification.

setDescription(String description): Sets the description of the notification.

HealthCenterController

Attributes:

HealthCenterController has no attributes since it is designed to be singleton, one HealthCenterController controls multiple entities.

Methods:

addTestResult(testResult:TestResult): Method takes a TestResult object as a parameter and it returns a Boolean that indicates whether the operation was successful or not. It can only be done by a MedicalEmployee. Method automatically updates the risk status of the test result's owner, which is a User, regardless of the test result. If the risk status of the User's becomes positive from negative to positive, the method also updates the risk status of the User's neighbors. Risk status of the User's neighbors is also updated when the risk status of the User's becomes negative from positive.

uploadPhysExamReport(exReport : ExaminationReport): Method takes an ExaminationReport object as a parameter and it returns a Boolean that indicates whether the operation was successful or not. It can only be done by the MedicalEmployee actor.

updateRiskStatus(riskStatus : String): Method takes a String that indicates the new risk status of the User as a parameter and it returns a Boolean that indicates whether the operation was successful or not. It cannot be called directly, since there must be a change in a User's risk status.

NeighborController

Attributes:

NeighborController has no attributes since it is designed to be singleton, one NeighborController controls multiple entities.

Methods:

addNeighbor(userId: Integer, neighborId: Integer): Method takes an Integer that represents a user id as a parameter and another Integer that represents neighbor id. It returns a Boolean that indicates whether the operation was successful or not. Method adds a neighbor to the user whose id is given, it also marks the user as a neighbor of the newly added neighbor.

getNeighbors(userId: Integer): Method takes an Integer that represents a user id as a parameter and it returns an array of Student objects. Method retrieves the neighbors of the user whose id is given.

getNeighborInfo(neighborId: Integer): Method takes an Integer that represents a neighbor id as a parameter and it returns a CovidInformation object. Method retrieves the covid information of the neighbor whose id is given.

deleteNeighbor(userId: Integer, neighborId: Integer): Method takes an Integer that represents a user id as a parameter and another Integer that represents neighbor id. It returns a Boolean that indicates whether the operation was successful or not. Method deletes a particular neighbor of the user whose id is given, it also removes the user from the neighbor's neighbor list.

getRiskyNeighbors(userId: Integer): Method takes an Integer that represents a user id as a parameter and it returns an array of Student objects. Method retrieves the risky neighbors of the user whose id is given.

AppointmentController

Attributes:

appointmentSchedule : Appointment

AppointmentController holds an AppointmentSchedule object to handle appointment operations and provide the user with the available time slots to be chosen in the making an appointment process.

Methods:

makeAppointment() : Boolean
This creates an empty appointment object with a unique appointmentId, which will be filled with the information that the user will enter (the time slot he picks etc.)

cancelAppointment(appointmentId : Integer) : Boolean
This method takes an appointmentId integer inside and uses it as a primary key to look up the appointment to cancel and remove from the appointment schedule.

getAppointmentSchedule() : AppointmentSchedule
This method returns the current schedule for making an appointment, and in the making an appointment process to present the user with the available time slots to pick.

updateAppointment(updated:Appointment,old:Appointment) : Boolean
This method is called to reschedule an appointment, replacing the old appointment with the new one. Boolean is used to indicate that the operation is successful or not, as if the new appointment does not fit the schedule the method would return false.

setTime(time : String, appointment : Appointment) : void
This method is used to set the time in a currently created appointment, and the reason why this method exist is because a boundary object that the user interacts with can only communicate with a controller object, and this method of the controller will call the setTime method of the Appointment class.

NotificationController

Attributes:

NotificationController has no attributes since it is designed to be singleton, one NotificationController controls multiple entities.

Methods:

sendNotification(notification: Notification, users: User[]): Method takes a Notification to be sent and an array of users as receivers. It returns a Boolean that indicates whether the operation was successful or not. Method sends the given notification to the given Users. It can only be done by the Admin or the System.

deleteNotification(notification: Notification): Method takes a Notification to be deleted and it returns a Boolean that indicates whether the operation was successful or not. Method deletes the given notification.

getNotifications(user: User): Method takes a User as the owner and it returns an array of Notification objects. Method retrieves the notifications of a particular user.

SocialController

Attributes:

SocialController has no attributes since it is designed to be singleton, one SocialController controls multiple entities.

Methods:

sendFriendRequest(friendRequest: FriendRequest): Method takes a FriendRequest object to be sent and it returns a Boolean that indicates whether the operation was successful or not. Method sends the given friend request. It can only be done by the actor Student.

acceptFriendRequest(friendRequest: FriendRequest): Method takes a FriendRequest object to be accepted and it returns a Boolean that indicates whether the operation was successful or not. Method accepts the given friend request. The sender and the sendee of the request become friends. It can only be done by the actor Student.

declineFriendRequest(friendRequest: FriendRequest): Method takes a FriendRequest object to be declined and it returns a Boolean that indicates whether the operation was successful or not. Method declines the given friend request. It can only be done by the actor Student.

getFriendStatus(userID:Integer): Method takes an Integer that represents the id of the user as a parameter. It returns a String that indicates the risk status of the user. Method retrieves the risk status of the given user. It can only be done by the actor Student.

removeFriend(removerId: Integer, removedId: Integer): Method takes two Integers that represent the id of the users as parameters. It returns a Boolean that indicates whether the operation was successful or not. Method removes the risk status of the given user's friend. It can only be done by the actor Student.

isFriend(userID:Integer, userID2:Integer): Method takes two Integers that represent the id of the users as parameters and it returns a Boolean. Method checks whether the second user is a friend of the first user. It can only be done by the actor Student.

PersonalInfoController

Attributes:

PersonalInfoController has no attributes since it is designed to be singleton, one PersonalInfoController controls multiple entities.

Methods:

getVaccinationInfo(userId: Integer): Method takes an Integer that represents a user id as a parameter and it returns a VaccinationInformation object. Method retrieves the vaccination information of the user whose id is given.

getPersonalInfo(userId: Integer): Method takes an Integer that represents a user id as a parameter and it returns a PersonalInformation object. Method retrieves the personal information of the user whose id is given.

uploadVaccinationCard(card: VaccinationInformation, userId: Integer): Method takes an Integer that represents a user id as a parameter and a VaccinationInformation object. It returns a Boolean that indicates whether the operation was successful or not. Method uploads the vaccination information of the user whose id is given.

getTestResults(userId: Integer): Method takes an Integer that represents a user id as a parameter and it returns an array of TestResult objects. Method retrieves the test results of the user whose id is given.

getRiskStatus(userId: Integer): Method takes an Integer that represents a user id as a parameter and it returns a String. Method retrieves the risk status of the user whose id is given.

modifyPersonalInfo(info : PersonalInformation, userId : Integer): Method takes an Integer that represents a user id as a parameter and a *PersonalInformation* object to replace the old one. It returns a Boolean that indicates whether the operation was successful or not. Method modifies the personal information of the user whose id is given. It can only be done by the Admin actor.

getCovidInfo(userId : Integer): Method takes an Integer that represents a user id as a parameter and it returns a CovidInformation object. Method retrieves the complete covid related information of the user whose id is given.

GeneralInfoController

Methods:

getUniversityStats() : Fetches university-wide stats from the database

getGuidelines() : Fetches the guidelines that are created by admins to the general information tab on the site.

updateGuidelines(guidelineId : Integer, guideline : Guideline) : Can only be done by admins, used to update the guidelines that are sent to all of the users. Takes a guidelineId to replace it with the new guideline.

getWeeklyReport() : Fetches the weekly COVID status including test results, number of positive cases, number of patients healed etc.

getAnnouncements() : Fetches the announcements which are sent by the admins

updateAnnouncements(announcements : Announcement[]) : Can only be done by admins, used to update the announcements that are sent to all of the users.

deleteAnnouncement() : Can only be done by admins, used to delete the announcements that are sent to all of the users.

deleteGuideline(guidelineId : Integer) : Can only be done by admins, used to delete the guidelines that are sent to all of the users.

SeatingController

Attributes:

SeatingController has no attributes since it is designed to be singleton, one SeatingController controls multiple entities.

Methods:

updateSeating(classroomId: Integer, seating: ClassroomPlan): Method takes an Integer that represents a classroom id and a ClassroomPlan object as a parameter to replace the old one. It returns a Boolean that indicates whether the operation was successful or not. Method modifies the existing seating plan of the classroom whose id is given. It can only be done by the admins and staff members.

getSeating(classroomId: Integer): Method takes an Integer that represents a classroom id as a parameter and it returns a ClassroomPlan object. Method retrieves the seating plan and classroom information of the classroom whose id is given.

ReportController

Attributes:
ReportController has no attributes since it is designed to be singleton, one ReportController controls multiple entities.

Methods:
addReport(report : ExaminationReport): Method takes an ExaminationReport object as a parameter and it returns a Boolean that indicates whether the operation was successful or not. Method adds the given examination report to the system. It can only be done by the actor Medical Employee.

getReports(userID: Integer): Method takes an Integer that represents the examination report id as a parameter and it returns an array of ExaminationReport objects. Method retrieves the examination reports of the user whose id is given.

FormController

Attributes:
FormController has no attributes since it is designed to be singleton, one FormController controls multiple entities.

Methods:

addForm(form: Form): Method takes a Form object as a parameter and it returns a Boolean that indicates whether the operation was successful or not. Method adds the given form to the system so that it can be evaluated by the admins.

getForm(fromID: Integer): Method takes an Integer that represents the form id as a parameter and it returns a Form object. Method retrieves the form whose id is given.

getForms(): Method returns an array of Form objects. Method retrieves all of the forms in the system.

evaluateForm(fromID: Integer): Method takes an Integer that represents the form id as a parameter and it returns a Boolean that indicates whether the evaluation was positive or not. Method evaluates the form with given and it can only be done by the Admin.

StatisticsController

Attributes:

StatisticsController has no attributes since it is designed to be singleton, one StatisticsController controls multiple entities.

Methods:

getPositiveCount(): Fetches the positive count from the database.

getPositivePercentage() : Fetches positive percentage from the database.

getVaccinatedCount() : Fetches the positive count from the database.

getVaccinatedPercentage() : Fetches the vaccinated percentage from the database

getTestCouınt() : Fetches the total number of tests done from the database

getTotalStatics() : Fetches the positive count from the database.

BaseController <<interface>>

Attributes: No attributes since it is an interface

Methods:

view(name : String, data : ArrayList<Object>) : void

model(name : String, data : ArrayList<Object>) : Model

authenticate(userId : UUID, role : String) : Boolean

This method is to be implemented by any implementing controller class, in this case, AuthenticationController will override this method as necessary.

BaseControllerDecorator

Attributes:

controller : BaseController

This class delegates an object of BaseController, the interface which it implements. Since we are using the Decorator design pattern, we are "decorating" the base controller; which is an interface implemented by all controller objects. The way we are "decorating" is authenticating in this case, which will be explained in the Design Patterns section.

Methods:

view(name : String, data : ArrayList<Object>) : void

model(name : String, data : ArrayList<Object>) : Model

authenticate(userId : UUID, role : String) : Boolean

This method will be "decorated" by the children of BaseControllerDecorator class, in this case, it is the AuthenticationController, which includes various extra methods and code to authenticate a certain action taken by other controllers.

AuthenticationController

Attributes:

controller : BaseController

This class delegates an object of BaseController, the interface which the parent class (BaseControllerDecorator) implements. Since we are using the Decorator design pattern, we are "decorating" the base controller; which is an interface implemented by all controller objects.

The way we are “decorating” is authenticating in this case, which will be explained in the Design Patterns section.

Methods:

login(userId : Integer, password : String) : Boolean
This method is used for logging in on the login screen. We need to verify whether userId and password matches during login. It takes an Integer userID and a String password to be matched.

signUp(user : User) : Boolean
This method is used for sign-up on the sign-up screen. It takes a User object and reaches the SRS system (an external actor) to check whether the info given by the user matches with the data on SRS or not. This is explained in detail with the activity diagram on the dynamic models section.

isAdmin(userId : Integer) : Boolean
This method is used when we need to check whether a user is an admin, and used to check whether that user has the privileges to perform a certain action (in other words, call a certain method). This method is called by the authorize method as a helper function.

isStudent(userId : Integer) : Boolean
This method is used when we need to check whether a user is a student, and used to check whether that user has the privileges to perform a certain action (in other words, call a certain method). This method is called by the authorize method as a helper function.

isAcademicPersonnel(userId : Integer) : Boolean
This method is used when we need to check whether a user is an academic personnel, and used to check whether that user has the privileges to perform a certain action (in other words, call a certain method). This method is called by the authorize method as a helper function.

isHealthCenterEmployee(userId : Integer) : Boolean
This method is used when we need to check whether a user is a health center employee, and used to check whether that user has the privileges to perform a certain action (in other words, call a certain method). This method is called by the authorize method as a helper function.

isSecurity(userId : Integer) : Boolean
This method is used when we need to check whether a user is an academic personnel, and used to check whether that user has the privileges to perform a certain action (in other words, call a certain method). This method is called by the authorize method as a helper function. Security, in particular, does not have any additional functionalities but the forms that are sent by the security are prioritized.

isStaff(userId : Integer) : Boolean
This method is used when we need to check whether a user is an academic personnel, and used to check whether that user has the privileges to perform a certain action (in other words, call a certain method). This method is called by the authorize method as a helper function.

view(name : String, data : ArrayList<Object>) : void

This method imports a view with the parameters that will be used in that view.

model(name : String, data : ArrayList<Object>) : Object

This method returns an object with the given name with its parameters.

authenticate(userId : UUID, role : String) : Boolean

This method overrides the method in the parent class, BaseControllerDecorator, where it adds many additional functionalities in determining whether a certain user is able to perform certain actions, in other words call certain functions from all controllers (where they all implement BaseController), as shown in the diagram. The

method takes the userId, and a role to be checked. Instead of using methods like isAdmin() etc. each time, we are using a single function for authorization and passing a parameter to specify which actor is to be checked. For example, we will be calling the isAdmin() function if the role input is “admin”, checking whether the user who tries to call the function has admin privileges to perform that certain action.

Database Layer Repository Class Diagram

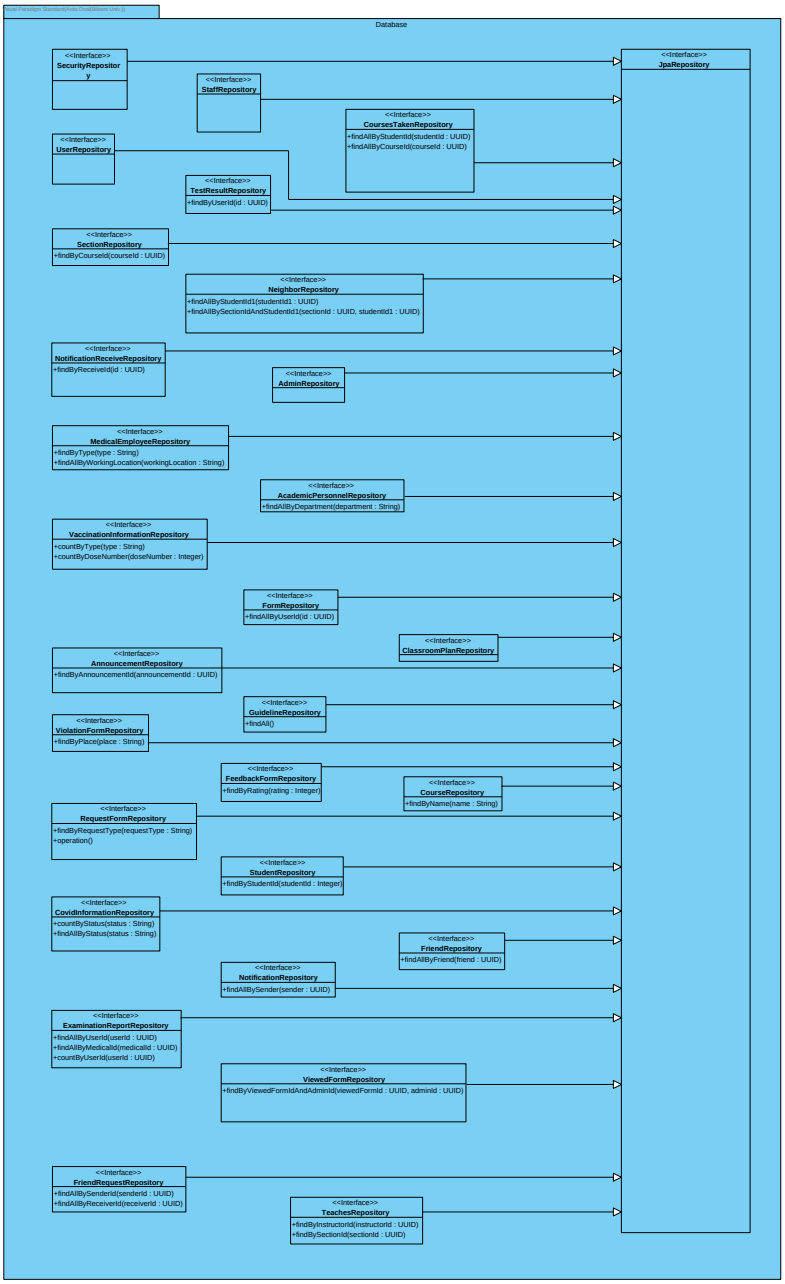

Fig. 9: Database Layer Repository Class Diagram

Design Patterns

Create Form Strategy Pattern:

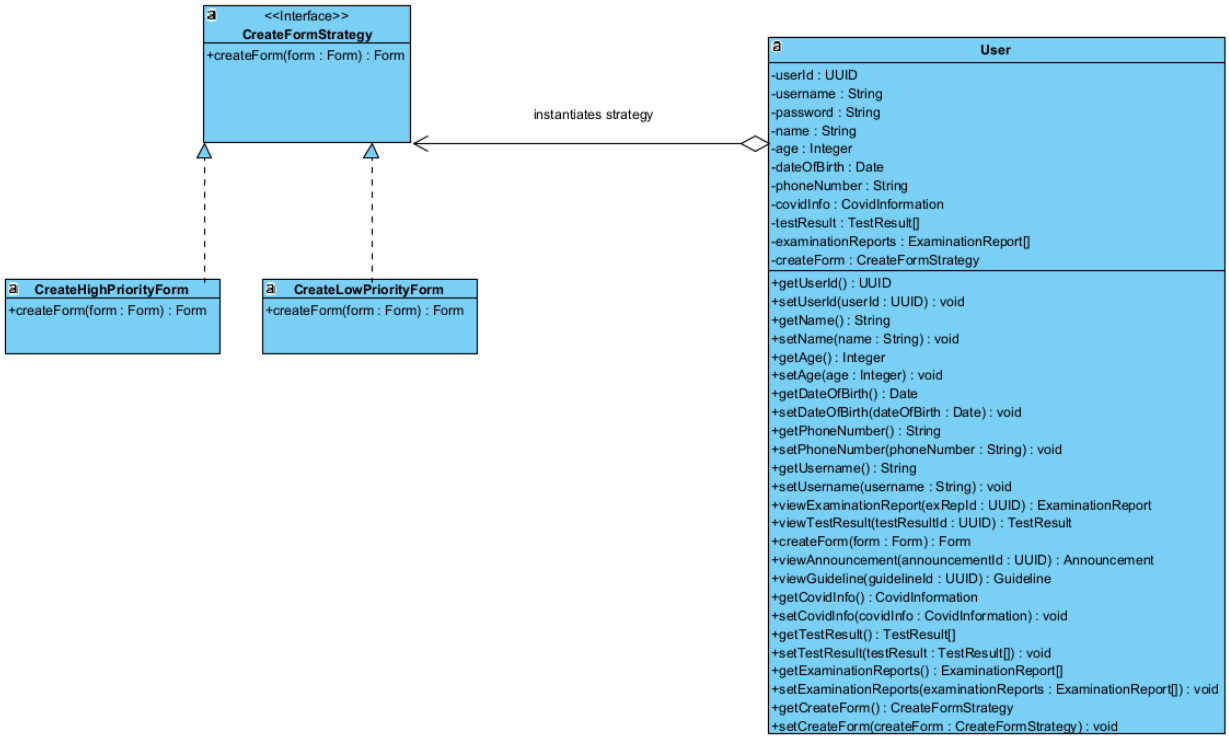

Fig. 6: Create Form Strategy Design Pattern

Strategy pattern is used to enable the selection of the create form algorithm at runtime, based on the user role. All users have the ability to create forms (violation report, request form, or feedback form), however, the algorithm for creating forms changes depending on the priority. User, which is the superclass of all role classes, has an abstract method for creating forms that should be overridden by all subclasses. Also, it holds a reference to the CreateFormStrategy interface, which is the strategy interface for the create form behavior. Two classes that implement this interface are CreateHighPriorityForm and CreateLowPriorityForm, and each has a different algorithm for creating forms depending on the priority. Role classes that extend User have their createForm methods calling a strategy, through the CreateFormStrategy interface reference in the User class. This way, students can send forms using the low priority strategy, whereas other users can select dynamically between the low and high priority strategies. All in all, strategy pattern makes the create form algorithms reusable in all role

classes and also enables some users to choose between strategies with different priorities at runtime.

Notification Decorator Design Pattern:

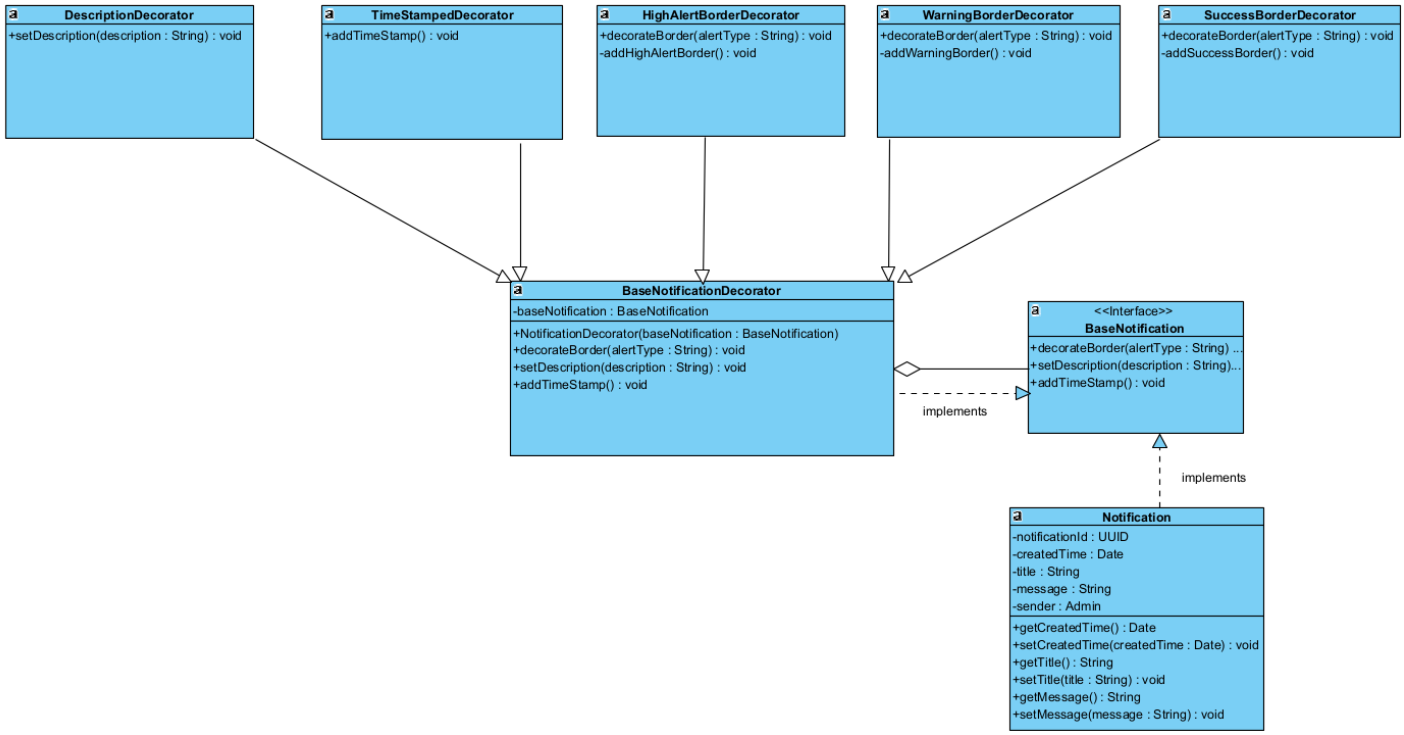

Fig. 7: Notification Decorator Design Pattern

The decorator pattern is used so that the notifications can be decorated according to their types. The DescriptionDecorator adds descriptions to the notifications, the TimeStampedDecorator adds a timestamp to the notification, the HighAlertBorderDecorator makes the border of the notification red, the WarningBorderDecorator makes the border of the notification yellow and the SuccessBorderDecorator makes the border of the notifications green. These decorators can be used together which is the purpose of the decorator design pattern. For example, when a neighbor of a student becomes positive, a notification which is decorated with DescriptionDecorator and TimeStampedDecorator, HighAlertBorderDecorator will be sent to the student which will have a description such as “One of your neighbors have tested positive”, which will include the test time of the neighbor and the notification will have a red border. A new announcement will be decorated with DescriptionDecorator and TimeStampedDecorator,

WarningBorderDecorator. Other types of notifications will be decorated accordingly using this design pattern.

Authentication Decorator Pattern:

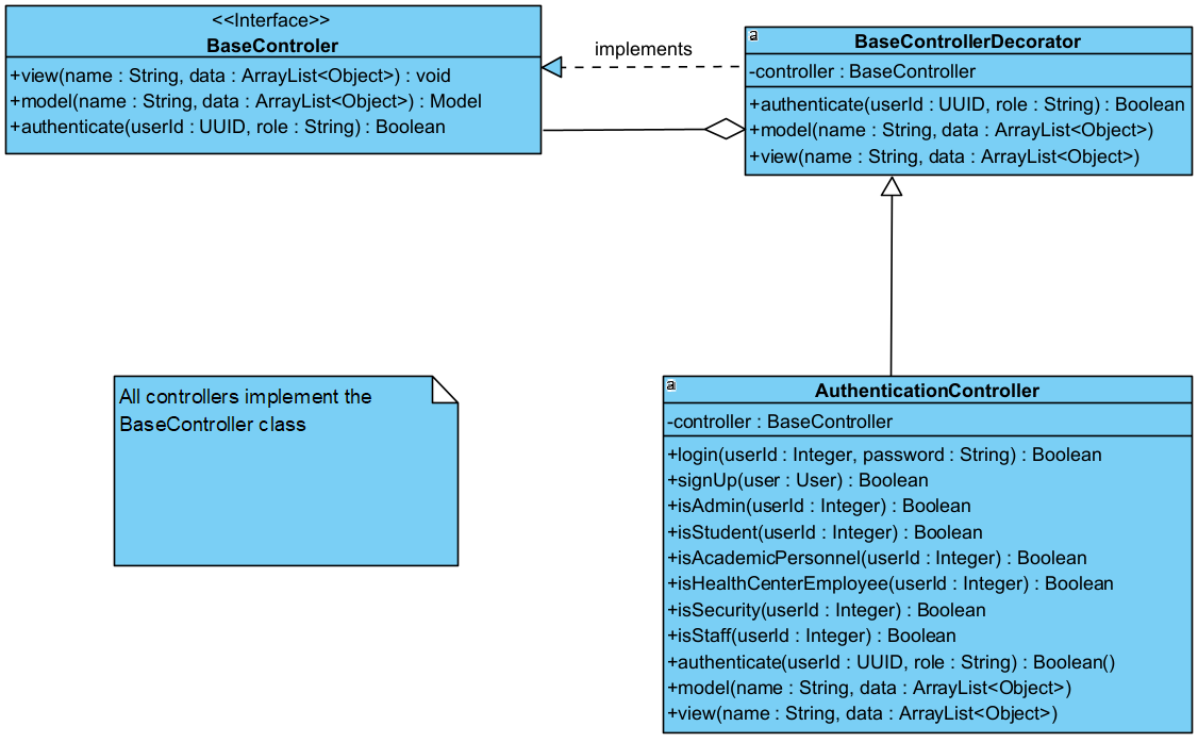

Fig. 8: Authentication Decorator Design Pattern

We used the decorator design pattern for authentication functionality. Every controller implements the BaseController interface, which has primitive controller methods along with the authenticate method. BaseControllerDecorator class implements the BaseController class and delegates an instance of the BaseController class to "decorate" it. The way it "decorates" all of the other controllers in this case is authentication, that AuthenticationController authenticates the actions all other controllers may take and checks if the user trying to perform this action has the privileges to do so. We are extending the authenticate() method with extra functionalities with decorator design pattern, and the authenticate method in AuthenticationController takes in a userId and a role to be checked. For example, if the String passed to the authenticate() method is "admin", isAdmin() method is going to be called and the controller will check whether the user of interest has admin privileges or not. This provides us with a

simpler design, that one does not have to call separate functions, just a single authenticate method to authenticate the actions taken by all other controllers that implement BaseController.

Packages and Frameworks

1. jQuery
2. Bootstrap
3. Angular Framework
4. Java Spring
    a. springframework.data.jpa
    b. springframework.security
    c. springframework.postgresql
    d. springframework.jdbc