CMPE 492 / SENG 492

Senior Design Project II

Low Level Design Report

Burak Güçlü

Gizem Yüksel

İrem Özyurt

Zeynep Sude Bal

Advisor

Emin Kuğu

09.03.2025

Table of Contents

1. Introduction

Natural disasters and unexpected emergencies are among the most important risks that threaten human life. Especially in countries located on active earthquake zones such as Turkey, such disasters occur frequently and can cause large-scale destruction. Disasters such as earthquakes, floods and fires occur suddenly and cause serious losses in a short time. In such cases, it is vital to quickly determine the safety status of disaster victims, direct emergency response teams and provide the necessary assistance as soon as possible. Traditional disaster management systems often cause delays and make post-disaster intervention difficult. At this point, creating a fast and effective disaster management process by taking advantage of the opportunities offered by today's technology is a great necessity in terms of minimizing loss of life.

This project aims to detect environmental risk factors through IoT-based sensors during and after a disaster, communicate with disaster victims through mobile applications and transmit the obtained safety information to emergency teams. The system continuously monitors environmental conditions with various IoT components such as earthquake wave sensors, temperature sensors and water level sensors, and automatically initiates a security verification process when it detects any danger. Individuals in the disaster area are sent a notification saying "Are you safe?" via the mobile application, and users are expected to respond with "I am safe" or "I am not safe". If the user does not respond within the specified time or selects the "I am not safe" option, the system automatically notifies the relevant emergency units and the user's relatives.

This system was developed to prevent communication disruptions during disasters, support rapid decision-making processes and make rescue operations more efficient. The mobile application has an infrastructure where users can determine and add emergency contacts before a disaster, update their profile information and receive instant security notifications when necessary. At the same time, the web-based management panel allows emergency teams to monitor user security status, view risk maps in disaster areas and respond quickly to incidents. Thus, beyond ensuring individual safety, the overall disaster management process is aimed at being carried out in a coordinated and effective manner.

As a result, this project aims to provide a fast, reliable and scalable solution against natural disasters, both to increase individual safety and to optimize disaster response processes. This system, strengthened by the integration of IoT technologies, mobile and web-based platforms, not only adapts to current disaster scenarios, but also has an expandable structure against

different types of disasters in the future. In this way, society will be better prepared for disasters and loss of life will be minimized.

1.1. Object Design Trade-offs

Object-oriented design is a process that requires critical decisions in software engineering. When creating software objects, developers must balance different design goals. In this process, trade-offs must be made between various criteria such as flexibility, maintainability, performance, and abstraction. While the right decisions increase the long-term maintainability and extensibility of the software, the wrong decisions can increase technical debt and maintenance costs. This section will cover the basic trade-offs that need to be considered in object design for post-disaster communication.

1.1.1 Abstraction vs. Complexity

Abstraction hides the internal details of the object or system, allowing users to focus only on the necessary information, manages this complexity and makes the system understandable. Good abstraction increases readability and modularity. However, excessive abstraction can complicate the system with unnecessary layers, making it difficult to understand and reduce performance. Complexity is the difficulty of understanding, managing and changing the system, high complexity can lead to errors and maintenance difficulties. While abstraction is used to reduce complexity, it can increase complexity when used incorrectly or excessively. The balance between abstraction and complexity is about determining the right level of abstraction; the ideal level is the one that simplifies the system sufficiently while not causing unnecessary complexity. To achieve this balance, unnecessary abstraction should be avoided, and abstraction should be used only at the points where it is needed. The right level of abstraction should be determined according to the target audience and the purpose of use, the purpose of each abstraction layer should be clear and documented. Simple solutions should be preferred, unnecessary complexity should be avoided, and abstraction should reduce dependency between modules and increase compatibility. Using abstraction only as much as necessary for our project will help us establish the right balance between complexity and abstraction to ensure readability and maintainability.

1.1.2 Flexibility vs. Simplicity

Flexibility in software systems is the ability to easily adapt to future changes and new features. Flexible systems offer the opportunity to adapt and expand to different scenarios with modular architecture, well-defined interfaces and abstraction layers. However, this flexibility can increase the complexity of the system; excessive abstraction and complex design patterns can make the system difficult. Simplicity, on the other hand, means that the system is understandable, easy to learn and maintain. Simple design prefers minimal solutions that are free from unnecessary complexity. Although simplicity provides rapid development at the beginning, it can create a rigid system that is resistant to future changes. In order to balance flexibility and simplicity, project requirements and life cycle should be taken into account. While simplicity may be a priority in short-lived projects, flexibility is important in long-term projects. In order to achieve this balance, sufficient abstraction should be used, and abstraction should be preferred only at necessary points. By adopting modular design, the system can be divided into independent modules. Design patterns should be used consciously, and only patterns that make the system flexible should be selected. Finally, both flexibility and simplicity can be carried out together by ensuring that the code is clean and understandable. In our project, it will be a balanced approach to avoid excessive abstraction in order to make the code understandable, and to provide flexibility with modular structures when necessary.

1.1.3 Readability vs. Efficiency

Readability refers to how easily code can be understood by humans; readable code is organized, well-structured, has meaningful names, and complies with standards. Readability speeds up the development process and facilitates maintenance. Efficiency refers to how effectively the code uses resources; efficient code runs quickly and consumes few resources. Efficiency optimizations can reduce the readability of the code; complex algorithms and low-level techniques can make it difficult to understand the code. The balance between readability and efficiency depends on project priorities. For most applications, readability should be a priority; code comprehensibility provides a more efficient process in the long run. However, efficiency may become more important in applications that require high performance. To achieve this balance, readability should be prioritized in normal situations and efficiency optimizations should be made only at critical performance points. Critical points should be determined with performance analyzes, and comment lines should be added to code sections whose readability decreases due to efficiency optimizations. The general structure of the code should be

documented and both readability and efficiency can be increased by conducting code reviews within the team. In the development phase of our project, prioritizing readability, except for critical performance points, and making it easier for the team to understand and maintain the code will provide a more efficient development process in the long run.

1.1.4 Coupling vs Cohesion

Coupling refers to the level of dependency between different objects or modules. High coupling means that changes in one module affect the others, while low coupling refers to the independence of the modules. High coupling makes the system difficult to maintain, test, and reuse. Cohesion, on the other hand, indicates how related and goal-oriented the elements within a module are. High cohesion ensures that the module focuses on a specific responsibility and its elements work to fulfill this responsibility. Low coupling and high cohesion are ideal; this is the basis of modular, maintainable, and flexible systems. Dependency injection can be used to achieve this balance, where dependencies between objects can be injected from outside, making modules more independent. Interfaces and abstract classes reduce dependency by defining interactions between modules. Design patterns can also be used to reduce coupling and increase cohesion. By adopting the single responsibility principle, each module has only one responsibility and fulfilling it increases cohesion. For our project, reducing the dependency between objects, ensuring that the modules work independently, and increasing cohesion by adopting a modular structure will make our system more sustainable.

1.1.5 Existing Components vs. New Components

Existing components are software components that have been previously developed and are in use; reusing them speeds up development time and increases reliability. However, existing components may not fully meet the specific requirements of the project or may not be optimal in terms of performance. New components, on the other hand, are components that are specifically developed for the project; they can be designed to fully meet the requirements of the project and optimized for performance. However, developing new components requires more time, cost, and risk. Choosing between existing and new components depends on the project requirements. In most cases, reusing existing components is preferable, but new components can be developed in cases where existing components are not suitable. To achieve this balance, existing components should be evaluated first and if there are reliable components

that can meet the requirements of the project, they should be reused. "Reinvention" should be avoided, suitable components should be selected, and the necessity of developing new components should be evaluated. A hybrid approach can also be an option, such as reusing existing components and developing new components for missing functions. Evaluating external libraries and existing technologies in our project, selecting the most suitable components, and avoiding unnecessary re-development will make the development process efficient and increase the reliability of the system.

1.1.6 Performance vs. Maintainability

Performance is the fast and efficient operation of the system under a certain workload; high performance means fast response, high throughput, and low resource consumption. Optimization techniques can increase performance, but excessive optimization can reduce the complexity and readability of the code. Maintainability is the ease of fixing bugs, adding new features, and ensuring long-term sustainability of the system. Systems with high maintainability are understandable, modular, and testable. However, focusing on maintainability can compromise performance optimizations. Balancing performance and maintainability depends on project requirements. While performance may be a priority in performance-critical applications, maintainability is more important in most business applications. To achieve this balance, performance analyses should be performed and optimizations should only be applied to bottlenecks. Code readability should be preserved, and high-level optimizations should be prioritized. Caching increases performance while affecting maintainability relatively little. In general, it is a good approach to prioritize maintainability and optimize only for critical performance requirements. In our project, it will be balanced to ensure long-term sustainability by prioritizing maintainability and making optimizations only where necessary.

1.1.7 Object Customization vs. Object Reuse

Object customization means optimizing an object for a specific purpose; in this specific case, it can provide high performance but restrict reuse. Customized objects only work effectively in the case they were designed for. Object reuse, on the other hand, is the ability of an object to be used repeatedly in different scenarios, which reduces development time and costs. Reusable objects are usually designed as general purpose and modular. However, general purpose objects may not be the best solution in a specific case and may be less efficient than customized objects

in terms of performance. The balance of object customization and reuse depends on the project requirements. In most cases, reuse should be the priority; reusable components are more efficient in the long run. However, customization may be required in critical performance or special functionality needs. Modular design can be used to achieve this balance, the system can be divided into modular components and both reusability and customization can be supported. Interfaces and abstract classes allow objects to be more general. Customization and reuse can be combined with configuration and parameterization. Design patterns can be used to make objects more flexible. Customization should be done only in critical components, other components should be designed for general purposes. Balancing both reusability and optimizations by designing critical components in our project modularly will ensure that the system is efficient and sustainable.

1.1.8 Inheritance vs. Composition

Inheritance allows new classes to derive properties and behaviors from existing classes, which increases code reuse and supports polymorphism. However, when used incorrectly, it can lead to problems such as rigid class hierarchies and high dependency, and deep inheritance hierarchies can make code difficult. Composition, on the other hand, offers a more flexible structure by composing objects from smaller, independent components; composition establishes an "owns" relationship. Composition provides lower dependency, higher reusability, and more flexible design. Choosing between composition and inheritance depends on design decisions, but composition is generally a more flexible and preferred approach. To achieve this balance, composition should be preferred first, because it creates more flexible and maintainable systems. Inheritance should be used with caution only when there is a truly "of-kind" relationship, inheritance can be supported with interfaces and abstract classes. Design patterns make it easier to create flexible structures using composition. Both inheritance and composition aim to increase code reusability, but composition offers more modular and flexible reuse. Prioritizing composition and ensuring that the code is modular and maintainable will help us create a more flexible system.

1.1.9 Time vs. Memory

Time refers to the time it takes to complete a process; time efficiency means completing the process quickly. Fast response times are a critical factor for time. Optimization techniques can

be used to increase time efficiency, but time optimizations can increase memory consumption. Memory is the amount of memory required for the system to operate; memory efficiency means doing the same job using less memory. Memory efficiency can be critical with limited memory resources. Techniques such as data structure optimization can be used to reduce memory consumption, but memory optimization can increase processing time. The balance between time and memory depends on project requirements and resource constraints. If fast response is important, time efficiency may be the priority, while memory efficiency may be the priority if there is limited memory. In most applications, both time and memory efficiency are important, and a balance must be found. To achieve this balance, time and memory usage should be analyzed with performance analyzes, and algorithm and data structure optimizations should be performed. Caching and data compression techniques can be used to balance time and memory. Asynchronous operations and parallelization can reduce processing time while increasing resource consumption. Memory management optimization increases memory efficiency. Time and memory trade-offs should be managed according to project requirements. Optimizing time-critical operations in our project while avoiding unnecessary memory consumption will help us strike the right balance between time and memory.

1.2. Interface Documentation Guidelines

Interface documentation is critical to understanding how different components of any software system interact with external systems. Especially in complex and vital projects such as post-disaster communication systems, accurate and complete documentation of interfaces is essential for the success of the development, testing, integration, and maintenance processes. Well-prepared interface documentation provides developers, testers, system integrators, and even emergency teams with access to accurate and up-to-date information about the system. This prevents possible misunderstandings, speeds up the development process, reduces errors, and increases the overall reliability of the system. This section provides basic guidelines and best practices to consider when creating interface documentation for our project.

1.2.1 The Basic Purpose and Functionality of Interface Documentation

The documentation for each interface should clearly explain the purpose and basic functionality of that interface within the system. This should include which components the interface communicates with, what types of data are transferred, and the interface’s role in the overall

system architecture. In a disaster relief system, for example, the purpose of the interface between a mobile application and a back-end server is to allow users to report their security status and to communicate this information to emergency responders. Interface documentation should provide a basic starting point for understanding why the interface exists and what problem it solves. In this context, clarity and comprehensibility are essential for interface documentation. The documentation language should be clear, concise, and free of technical jargon, and should be easy for anyone reading the document to understand the interface.

1.2.2 Identifying Endpoints, Methods, and Functions

The interface documentation should list in detail all endpoints, methods, or functions provided by the interface. The following information should be provided for each endpoint: A unique and meaningful name for the endpoint or method, and a concise description explaining what it does, what action it performs, or what information it provides. For RESTful APIs, the HTTP method used (GET, POST, PUT, DELETE, etc.) should be specified. In addition, a list of input parameters expected by the endpoint or method (with name, type, mandatory status, and description) should be included in the documentation.

1.2.3 Explanation of Request and Response Formats

The interface documentation should describe in detail the data formats and structures of requests sent and responses received through the interface. Especially if formats such as JSON or XML are used for data exchange, schemas and examples of these formats should be included in the documentation. Due to the principle of comprehensiveness, the documentation should cover all important aspects of the interface. Explaining data structures with examples helps developers use interfaces correctly and perform data exchange smoothly. Interface documentation should always be consistent with the current state of the interface, in accordance with the principles of accuracy and timeliness, and any changes made to the interface should be reflected in the documentation in a timely manner.

1.2.4 Detailed Information About Parameters and Arguments

Detailed descriptions should be provided for each input parameter and output argument. These descriptions should include the purpose of the parameter or argument, its data type, possible

value ranges, mandatory status, and any special rules. Such details provide consistency and clarity for developers using the interface. All interface documentation throughout the project should be consistent in terms of format, terminology, and presentation, making it easy to navigate and understand different interfaces.

1.2.5 Documentation of Return Values and Responses

The possible return values or responses (including successful and error cases) of each endpoint or method should be fully documented. The expected data structure and format for successful responses, and the possible error codes, error messages, and their meanings for error cases should be explained. This allows applications consuming the interface to respond correctly to different scenarios and error cases. In a disaster relief system, it is critical to clearly specify error conditions and their meanings, especially in terms of system reliability and debugging processes.

1.2.6 Explaining the Error Management Strategy

The interface documentation should explain the system's error management strategy and how the interfaces should behave in error conditions. All possible error codes, their meanings, error messages, and actions to be taken in the event of an error (e.g., retry, display notification to the user, etc.) should be specified in detail. In critical systems such as disaster management, correct management and rapid resolution of error conditions are of vital importance.

1.2.7 Define Authentication and Authorization Mechanisms

Security is a top priority for applications that handle sensitive data, such as disaster relief systems. Interface documentation should fully describe the authentication and authorization mechanisms used to access interfaces (e.g., API keys, OAuth 2.0, JWT, etc.). It should clearly state which endpoints require authentication, which roles can access which resources, and the steps in the authentication process. This helps prevent unauthorized access and secure the system. Accessibility of interface documentation is also important; documentation should be easily accessible to all relevant stakeholders in the project, and the use of a centralized documentation management system or project wiki can increase accessibility.

1.2.8 Documentation Tools and Continuous Updates

Appropriate tools (Swagger/OpenAPI, Markdown, Online Documentation Platforms, UML Diagrams, etc.) and formats should be selected for interface documentation, and documentation should be continuously updated. Automated documentation tools and continuous integration processes can help keep documentation up to date.

1.3. Engineering Standards

Engineering standards are critical to ensuring quality, consistency, and interoperability in software development processes. Especially in complex and vital projects such as disaster relief systems, compliance with standards increases the reliability, sustainability, and efficiency of the system. This section will discuss the main engineering standards that should be considered within the scope of this project, especially UML (Unified Modeling Language) and IEEE (Institute of Electrical and Electronics Engineers) standards.

1.3.1. Importance of Engineering Standards

Engineering standards provide many benefits in the software development process:

- Standards facilitate communication by creating a common language and reference framework among project team members, stakeholders, and other interested parties.
- Standards provide structure and discipline to software development processes, reducing errors, improving testing processes, and ultimately resulting in higher quality and more reliable systems.
- Systems developed in accordance with standards are easier to maintain and update because they are better documented, modular, and understandable.
- Standards facilitate integration processes by ensuring that different system components and software work in harmony with each other.
- Standards include the best practices and methods accepted in the field, which supports the project to be more professional and successful.

1.3.2. UML (Unified Modeling Language)

UML (Unified Modeling Language) is a standard language used to visually model software systems. UML diagrams facilitate the understanding and communication of complex systems by representing different aspects of the system (structural, behavioral, interactional, etc.).

1.3.2.1. Use Case Diagrams

It shows the functional requirements of the system and how users interact with the system. For a post-disaster relief system, use cases such as "Report Security Situation", "Request Emergency Assistance", "Monitor Sensor Data" can be defined. These diagrams support requirements analysis and scoping.

1.3.2.2. Class Diagrams

It shows the classes in the system, their properties, methods and the relationships between them (association, inheritance, composition, etc.). Used to design the system architecture and data model. For example, classes such as User, SecurityStatus, Sensor, EmergencyTeam and the relationships between them can be modeled.

1.3.2.3. Sequence Diagrams

It shows the interactions between objects in time order. Used to understand the message exchange and workflow of objects for a specific use case or scenario. For example, in the "Security status verification" scenario, the interactions between the mobile application, web server, sensor data module and database can be modeled.

1.3.2.4. Activity Diagrams

It is used to model workflows, processes and algorithms. Useful for visualizing parallel and sequential processes within the system. For example, "Disaster notification sending process" or "Security status processing process" can be modeled with activity diagrams.

1.3.2.5. State Diagrams

It shows the different states and state transitions of an object. It is especially useful in event-driven systems or for modeling the life cycle of objects. For example, the states of the User object such as "Active", "Safe", "Waiting for Help", "Reached" and the transitions between these states can be modeled.

1.3.3. IEEE Standards

IEEE (Institute of Electrical and Electronics Engineers) is a major organization that develops and publishes standards in various engineering disciplines. IEEE standards in the field of software and systems engineering provide guiding principles and methods on critical issues such as reliability, quality, security and performance. Some relevant IEEE standards that can be considered for a disaster relief system project are as follows:

1.3.3.1. IEEE 830 - Software Requirements Specification (SRS)

It provides complete, consistent, understandable and testable documentation of software requirements. This standard helps to clearly define project requirements and establish a common understanding among stakeholders.

1.3.3.2. IEEE 1016 - Software Design Definition (SDD)

It provides detailed documentation of software design. Provides a framework for recording design decisions and justifications such as architectural design, interface design, data structure design, algorithm design.

1.3.3.3. IEEE 829 - Software Test Documentation

It sets standards for planning, executing and reporting software testing processes. It defines how to prepare test documents such as test plans, test scenarios, test reports and what to consider.

1.3.3.4. IEEE 1012 - Software Verification and Validation (V&V)

It defines how to manage and document V&V processes to verify that the software conforms to the requirements and meets expectations.

1.3.3.5. IEEE 1490 - Software Project Management Plan

It provides a standard framework for planning, managing, monitoring and controlling software projects. It describes how to plan and manage project management elements such as project scope, schedule, resource management, risk management and communication management.

1.3.3.6. IEEE 730 - Software Quality Assurance Plans

It defines how to plan and implement quality assurance processes to ensure software quality. It provides a guide for managing quality assurance activities such as quality standards, audits, tests and configuration management.

1.3.3.7. IEEE 802.11 Series - Wireless Local Area Network (WLAN)

It defines wireless communication protocols. It is important to select and implement appropriate standards (e.g. Wi-Fi standards) for wireless communication, especially between IoT sensors and mobile devices.

1.3.3.8. IEEE 27000 Series - Information Security Management Systems

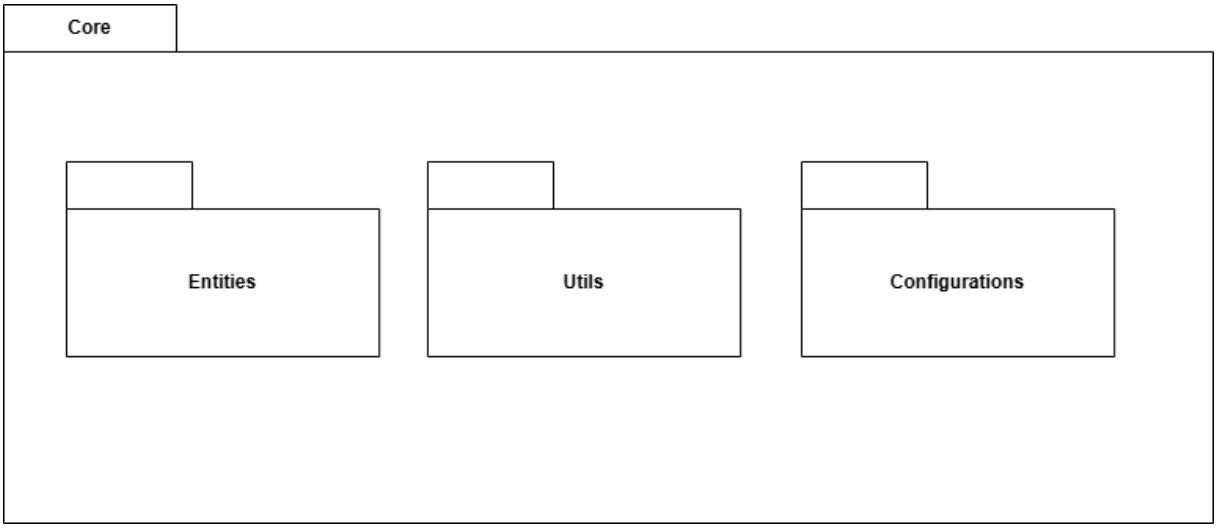
It provides standards for establishing, implementing, maintaining and continuously improving information security management systems. These standards should be taken into account to meet security requirements such as confidentiality, integrity and accessibility of user data.

Engineering standards are essential to the success of critical and complex software projects, such as a post-disaster communication project. By using accepted standards such as the UML modeling language and IEEE standards, the project team can communicate more effectively, develop higher quality software, and increase the reliability and maintainability of the system. Adherence to standards helps the project conform to industry best practices and is more successful in the long term.

2. Packages

To ensure modularity, scalability, and maintainability, the system is structured into different packages, each responsible for a specific functionality within the disaster management platform. These packages interact seamlessly to facilitate real-time disaster detection, alert management, and user response tracking. Below is a detailed breakdown of the packages, including their respective sub-packages and responsibilities.

2.1. Core Package



The Core package forms the foundation of the system by defining the primary data models, utility functions, and configuration settings that are shared across different modules. This package ensures that the system maintains a centralized and consistent data structure while providing essential helper functions for smoother operation.

2.1.1. Entities

This module defines key objects such as User, Sensor, Notification, and EmergencyContact. The User entity holds details such as name, contact information, and authentication credentials. The Sensor entity represents disaster detection devices that monitor various environmental factors such as seismic activity, temperature fluctuations, and rising water levels. The Notification entity handles alerts triggered by detected disasters, ensuring proper message delivery to users and emergency response teams. Finally, the EmergencyContact entity allows users to store information about close contacts who should be notified in case of an emergency.

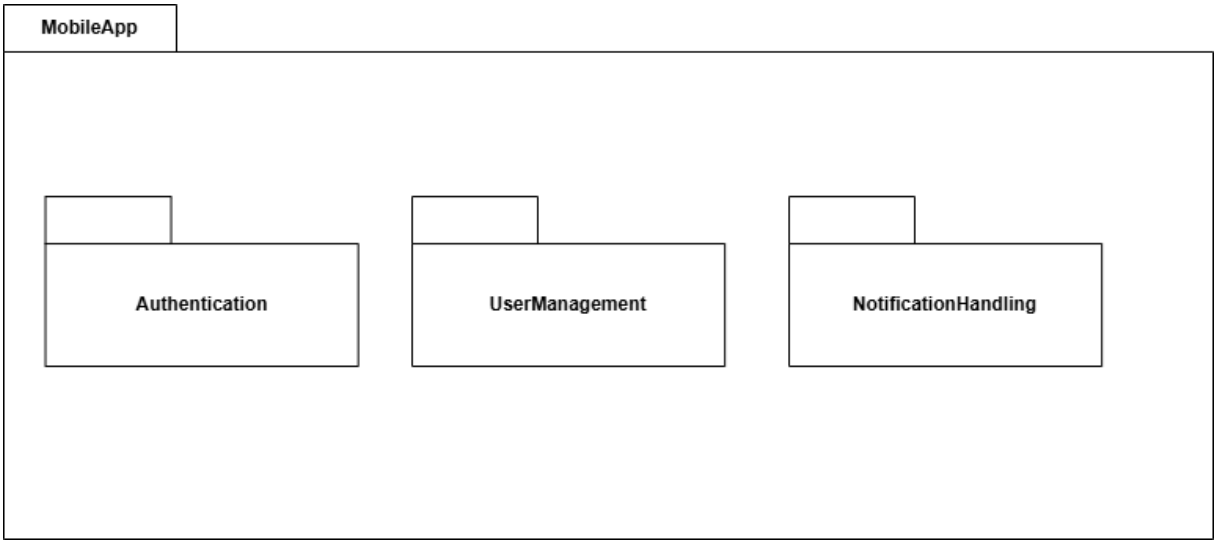
2.1.2. Utils

The Utils module provides helper functions to handle common tasks across the system. These include data conversion functions (e.g., JSON serialization and deserialization), error-handling mechanisms, logging functionalities, and data validation tools. These utilities simplify the implementation of key features and enhance the overall efficiency of the system.

2.1.3. Configurations

The Configurations module stores all system-wide settings and parameters, allowing dynamic adjustments to be made without modifying core logic. It includes settings for notification timeouts, API keys for external services, database connection parameters, and sensor data thresholds. This module ensures that the system remains adaptable to different environments and operational requirements.

2.2. MobileApp Package



The MobileApp package is dedicated to the mobile application used by end-users. It allows individuals to register, receive emergency alerts, respond to disaster notifications, and manage their personal safety settings.

2.2.1. Authentication

The Authentication module handles user login, registration, and authentication token management. It ensures that only authorized users can access the system, using secure encryption mechanisms for storing passwords and managing session tokens.

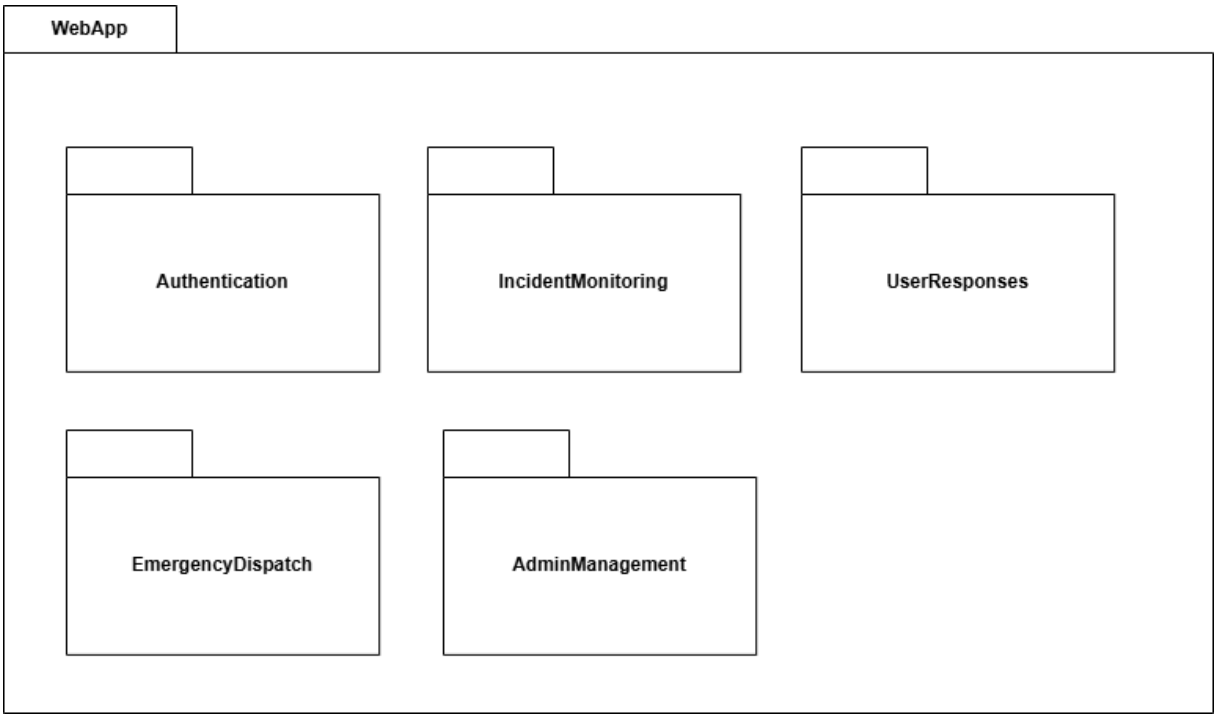
2.2.2. UserManagement

The UserManagement module enables users to update their profile details, including phone numbers, email addresses, and emergency contact lists. It also allows users to add and remove emergency contacts, ensuring that loved ones receive timely notifications during a disaster.

2.2.3. NotificationHandling

The NotificationHandling module is responsible for receiving, displaying, and managing disaster alerts. When a disaster occurs, the system sends notifications to users via push notifications and in-app alerts. Users can respond by marking themselves as safe or unsafe, and the system logs these responses for further action.

2.3. WebApp Package



The WebApp package serves as the control center for emergency response teams and administrators. It provides an interface for monitoring real-time disaster data, tracking user responses, and making critical decisions based on the collected information.

2.3.1. Authentication

The Authentication module ensures secure login and access control for emergency personnel and administrators. User roles and permissions are managed here, preventing unauthorized access to sensitive data.

2.3.2. IncidentMonitoring

The IncidentMonitoring module displays real-time sensor data on a dashboard, allowing administrators to assess disaster severity. It includes graphical representations, live maps, and trend analysis tools to support decision-making.

2.3.3. UserResponses

The UserResponses module tracks responses from users who received disaster alerts. It differentiates between users who have responded (marked themselves as safe or unsafe) and users who have not responded within the given timeframe. This helps emergency responders prioritize their actions based on the available information.

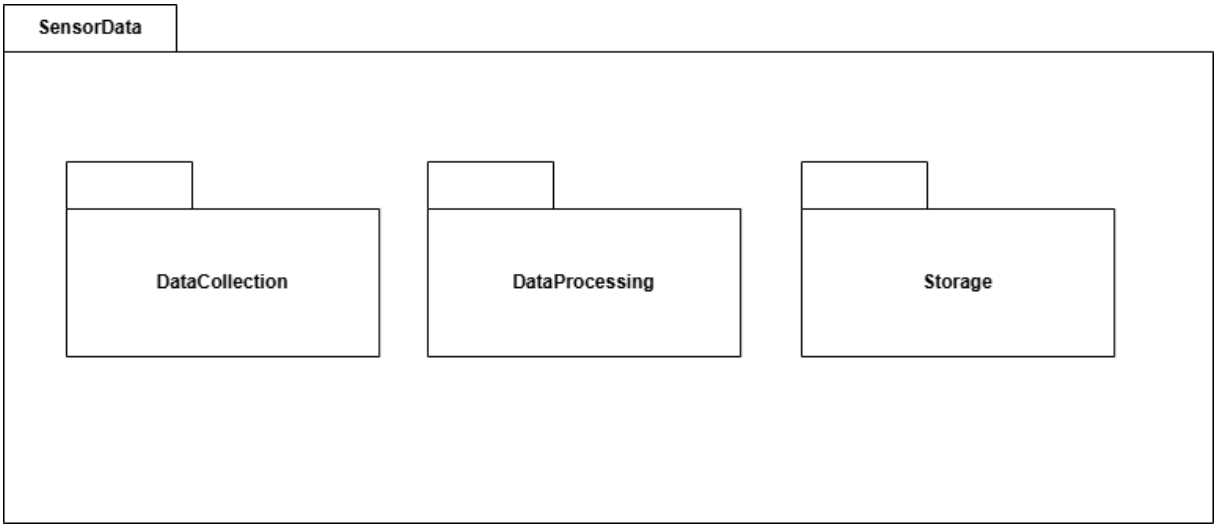
2.3.4. EmergencyDispatch

The EmergencyDispatch module is responsible for escalating alerts to emergency responders. If a user marks themselves as unsafe or fails to respond within a set timeframe, the system automatically notifies the appropriate emergency services.

2.3.5. AdminManagement

The AdminManagement module provides administrative controls, allowing authorized personnel to manage users, configure system settings, and access logs of past incidents. This module ensures smooth operation and governance of the disaster management platform.

2.4. SensorData Package



The SensorData package is responsible for collecting, processing, and storing real-time sensor data from various monitoring devices. It ensures that disaster alerts are triggered based on accurate environmental readings.

2.4.1. DataCollection

The DataCollection module is designed to continuously receive and log data from sensors that detect temperature, water levels, and seismic activity. This module ensures that sensor readings are accurately transmitted to the system in a structured format.

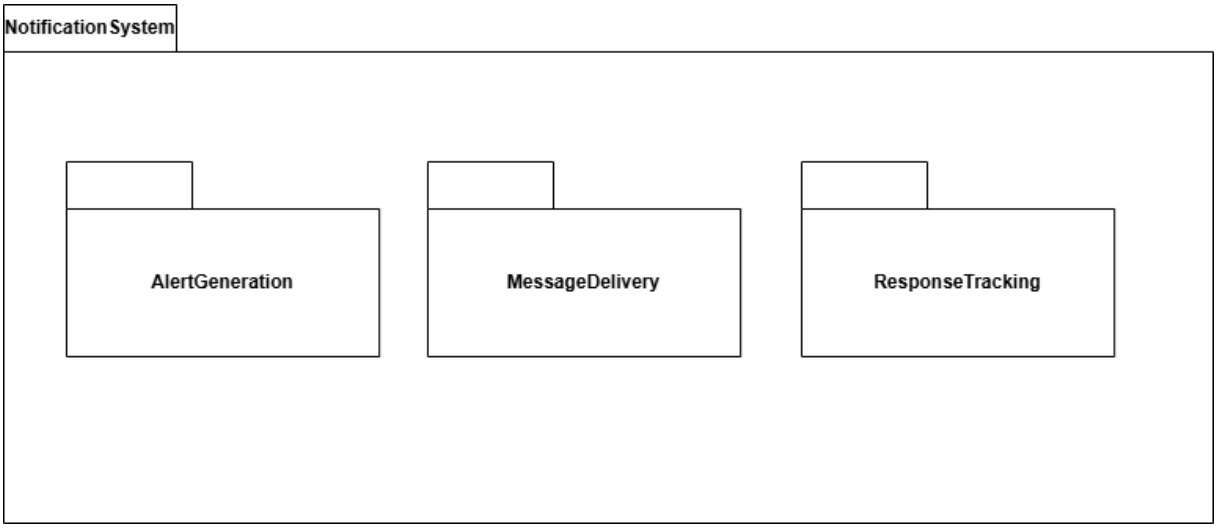
2.4.2. DataProcessing

The DataProcessing module is responsible for analyzing sensor data and determining whether the readings exceed predefined safety thresholds. If an anomaly is detected—such as a sudden spike in seismic activity or rising water levels—this module triggers an alert for further action.

2.4.3. Storage

The Storage module ensures that historical sensor data is saved in a structured database. This enables trend analysis, predictive modeling, and forensic investigation in the aftermath of a disaster.

2.5. NotificationSystem Package



The NotificationSystem package manages the creation, delivery, and tracking of disaster alerts. It plays a crucial role in ensuring that users receive timely and accurate information during emergencies.

2.5.1. AlertGeneration

The AlertGeneration module is responsible for creating notifications based on sensor triggers. When a disaster is detected, this module generates an alert containing relevant details such as location, severity, and type of hazard.

2.5.2. MessageDelivery

The MessageDelivery module ensures the efficient dissemination of notifications to users via multiple channels, including mobile push notifications, SMS messages, and email alerts. This module integrates with external communication APIs to ensure reliable message delivery.

2.5.3. ResponseTracking

The ResponseTracking module continuously monitors user responses to notifications. It maintains a log of acknowledged alerts and tracks users who have not responded within a specified timeframe. This information is crucial for emergency teams, helping them prioritize rescue efforts based on real-time user feedback.

2.6. Conclusion

This package structure ensures a modular and scalable design, where each package has a distinct responsibility. It enables efficient communication between components while maintaining clear separation of concerns. By following object-oriented principles, this design allows future enhancement and maintenance without affecting other system components.

3. Class Interfaces

In our project, the class interfaces play a crucial role in ensuring seamless communication between different components of the system. A class interface defines the public methods and attributes that external components can access, while abstracting away the internal implementation details. This approach promotes modularity, maintainability, and scalability within our system.

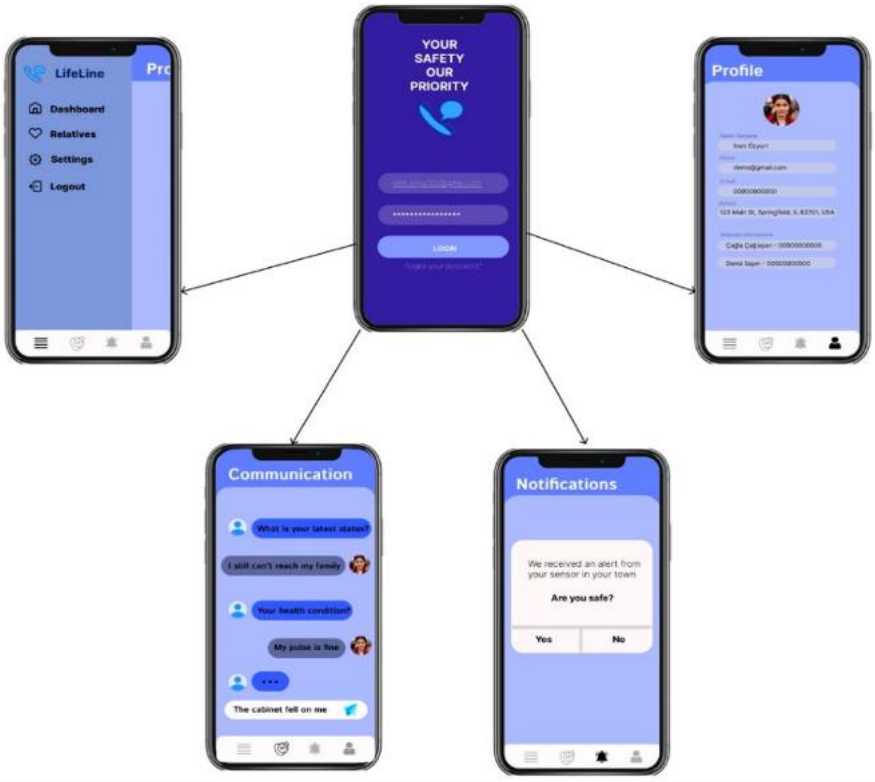
Given the complexity of our project, which involves multiple components such as notification systems, sensor data processing, and database management, a well-structured class interface is essential. Each class in our system is designed with a clear and well-documented interface, specifying the available methods, attributes, parameter types, return values, and potential exceptions. This ensures that different components—whether in the Flutter mobile application, the Node.js backend, or the React web application—can interact with each other efficiently without needing to understand the internal workings of every class.

By following object-oriented design principles, we have structured our class interfaces to be extensible and reusable. The interfaces allow data exchange between different modules while enforcing encapsulation and adhering to engineering standards such as UML modeling and IEEE software design guidelines. Additionally, we have implemented best practices such as error handling, access control mechanisms, and optimized data structures to enhance system performance and reliability.

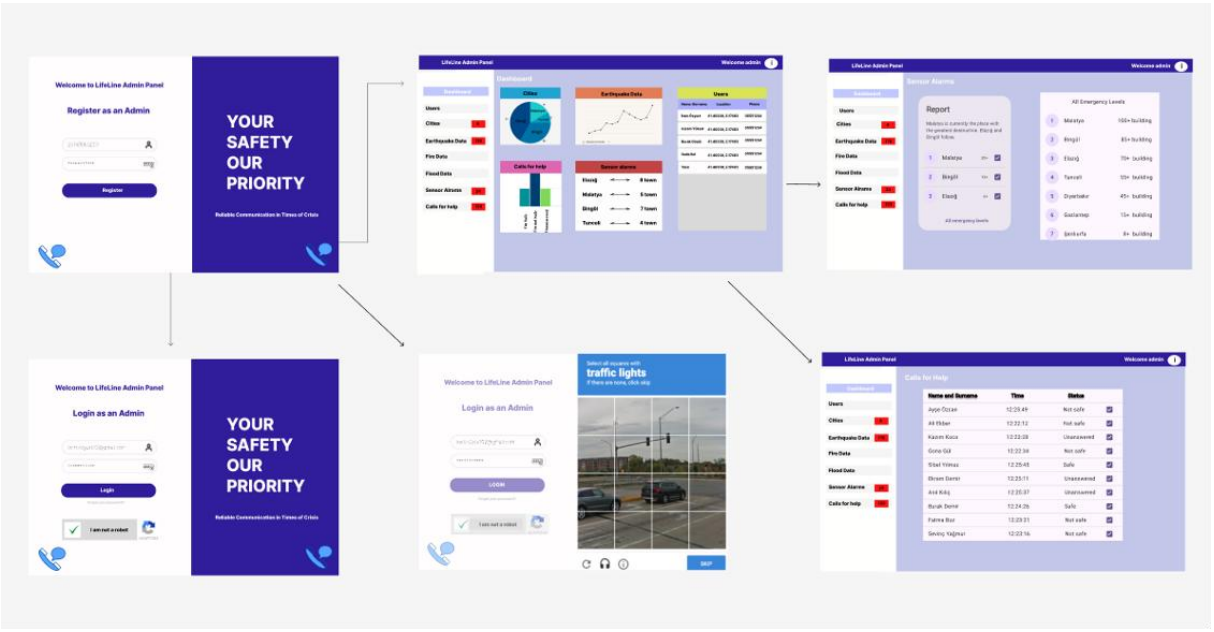
In the following sections, we will present the detailed class interfaces for key components of our system, including their attributes, methods, and interactions with other modules.

3.1. Project Interfaces

3.1.1. Mobile Application Interfaces



3.1.2. Web Application Interfaces



3.2. Project Components

Web Application Components	Mobile Application Components
Authentication Components LoginForm User login form RegisterForm New user registration form CaptchaComponent: CAPTCHA validation module for security	Authentication Components LoginScreen User login screen RegisterScreen New user registration screen
Navigation Components Sidebar: Access to tabs such as users, cities, earthquake data, fire and flood data TopBar User profile information and logout button	Navigation Components BottomTabNavigator: Main menu options (Dashboard, Relatives, Settings, Logout)
Dashboard Components StatsCard: Card component with general statistics and visuals PieChart: Pie chart showing city-based earthquake data LineChart: Graph showing data from earthquake sensors over time	User Profile Components ProfileCard: Card containing the user's profile information EditProfileForm: Form for the user to update their profile information
Emergency Alerts Components SensorAlertsTable: Table listing sensor alarms EmergencyLevelList: List showing city-based emergency levels	Communication Components ChatScreen Messaging screen between users ChatBubble Component containing user messages
Help Requests Components HelpRequestsTable: Table that lists help requests from users	Notification Components AlertPopup: Emergency notification (Ex: “Sensor alarm detected in your city. Are you safe?”) NotificationList: List of all notifications received by the user

3.2.1. Authentication Components Web Application

Login.js-Register.js	
Manages notifications. Uses Firebase Cloud Messaging to receive and handle notifications.	
Attributes	
_username: The username input field. password: The password input field. errorMessage: Message to display when login fails.	
Methods	
validateCredentials()	Checks if the username and password are correct.
_handleSubmit()	Submits the login form to authenticate the user.
verifyCaptcha()	Validates the CAPTCHA input by the user.

3.2.2. Authentication Components Mobile Application

Login.dart-Register.dart	
Manages notifications. Uses Firebase Cloud Messaging to receive and handle notifications.	
Attributes	
_username: The username input field. password: The password input field. errorMessage: Message to display when login fails.	
Methods	
validateCredentials()	Checks if the username and password are correct.
_handleSubmit()	Submits the login form to authenticate the user.
verifyCaptcha()	Validates the CAPTCHA input by the user.
navigateToRegister()	Redirects the user to the registration screen.

3.2.3. Navigating Component Web Applications

Sidebar.js-Topbar.js
Manages navigating process.
Attributes
menuItems: List of navigation menu items ("Cities", "Earthquake Data"). activeItem: Tracks the currently active menu item.
Methods
selectItem(): Changes the active menu item. toggleSidebar(): Toggles the visibility of the sidebar. handleLogout(): Logs out the user. showProfile(): Displays user profile options.

3.2.4. Navigating Component Mobile Application

BottomTabNavigator.dart
Manages navigating process.
Attributes
tabs: List of tabs (e.g., "Dashboard", "Relatives", "Settings"). activeTab: The currently active tab.
Methods
switchTab(): Switches to the selected tab. navigateTo(): Navigates to the corresponding page of the selected tab.

3.2.5. Dashboard Component Web Application

Dashboard.js
Overview of the app
Attributes
title: Title of the stats card ("Total Alerts"). stats: The data displayed (e.g., number of earthquakes). image: An image or icon related to the stats. data: The data used to render the pie chart (earthquake occurrences per city). title: Title of the pie chart ("Earthquake by City").
Methods
updateStats(): Updates the statistics on the card. displayCard(): Renders the stats card. renderChart(): Renders the pie chart based on the provided data. updateData(): Updates the pie chart with new data.

3.2.6. Profile Component Mobile Application

ProfileCard.dart
User can view and edit own personal informations
Attributes
userName: The user’s name. userEmail: The user’s email. userProfilePic: The user’s profile picture. name: Field to edit the user's name. email: Field to edit the user's email. profilePic: Field to upload a new profile picture.
Methods
displayProfile(): Displays the user's profile information. editProfile(): Allows the user to edit their profile information. validateForm(): Validates the edit form fields. submitChanges(): Submits the changes to the backend.

3.2.7. Chat Component Mobile Application

ChatScreen.dart
Userc can interact with the emergency in here.
Attributes
messages: List of messages between users. userStatus: The online/offline status of the user. sender: The sender of the message. message: The message content. time: The time when the message was sent.
Methods
sendMessage(): Sends a new message. loadMessages(): Loads previous messages for the chat. renderBubble(): Renders the message bubble. highlightMessage(): Highlights the message if new.

3.2.8. Notification Component Mobile Application

NotificationManager.dart
Manages notifications. Uses Firebase Cloud Messaging to receive and handle notifications.
Attributes
_firebaseMessaging: FirebaseMessaging instance for FCM communication. userToken: User's FCM token. notifications: List storing received notifications.
Methods
fetchUserToken() Retrieves the user's Firebase token. _configureFirebaseListeners() Listens for incoming notifications and adds them to the list. sendLocalNotification(title, body) Displays a local notification on the device. getNotificationHistory() Returns a list of past notifications.

3.2.9. Notification Component Web Application

NotificationController.js	
A backend controller class that handles web push notifications.	
Attributes	
_subscriptions: An array that stores users subscribed to notifications.	
Methods	
subscribeUser(req, res)	Adds a user to the notification subscription list.
sendNotification(req, res)	Sends a notification to a specific user.
broadcastNotification(req, res)	Sends a notification to all subscribed users.

3.2.10. Emergency Alerts Components Web Application

SensorAlertsTable.js
Admin people can monitor alerts can navigate in this screen.
Attributes
alerts: List of sensor-generated alerts. filter: Criteria to filter alerts (active, resolved). cityName: The name of the city. emergencyLevel: The level of emergency ("Critical", "Moderate").
Methods
fetchAlerts(): Fetches the list of alerts from the database. displayAlert(): Displays an individual alert in the table fetchLevels(): Fetches emergency levels for cities. updateLevel(): Updates the emergency level for a city.

3.2.11. Sensor Data Component Mobile Application

SensorDataModel.dart
A model class representing sensor data readings.
Attributes
sensorId: Unique identifier of the sensor. value: The recorded sensor reading.
Methods
Constructor and getter methods.

3.2.12. Sensor Data Processing Component Mobile Application

SensorDataManager.dart	
Handles sensor data processing in the mobile application. Collects, filters, and processes real-time sensor readings.	
Attributes	
sensorStream: A stream that continuously receives sensor data. processedData: A list storing processed sensor values. threshold: A predefined limit for alert generation.	
Methods	
startListening()	Starts listening to the sensor stream and processes incoming data.
filterData(rawData)	Filters out noise from raw sensor readings.
detectAnomalies(data)	Checks if sensor data exceeds the predefined threshold.
getProcessedData()	Returns the list of processed sensor data.

3.2.13. Sensor Data Component Web Application

SensorData.js
A React component that displays real-time sensor data in the web application.
Attributes
sensorData: An array storing processed sensor readings. alerts: An array of generated alerts.
Methods
useEffect() Listens for real-time updates via WebSocket or API calls. render() Displays the latest sensor readings and alerts in a dashboard.

3.2.14. Sensor Data Processing Component Web Application

SensorDataController.js
Backend controller for handling sensor data from IoT devices and processing alerts.
Attributes
sensorData: Stores incoming sensor data from devices. alerts: Stores alerts generated due to abnormal sensor readings.
Methods
receiveSensorData(req, res) Receives and stores real-time sensor data from IoT devices. analyzeSensorData() Processes sensor data and determines if an alert should be generated. getSensorData(req, res) Retrieves historical sensor data for analysis. sendAlert(alertData) Sends alerts if an abnormal sensor reading is detected.

3.2.15. Database Component Firebase

Value	Description
users	Stores user details such as name, email, and authentication info.
messages	Stores user messages exchanged in the application.
notifications	Stores notifications sent to users.
sensorData	Stores IoT sensor readings for further processing.
alerts	Stores alerts generated from abnormal sensor readings.

3.2.16. Firebase Firestore Database Structure

Description	**Manages all Firebase Firestore interactions in the Flutter mobile app. Handles user data, messages, and notifications.**	
Attributes	_firestore: Firestore instance for database operations. userCollection: Reference to the users collection. messageCollection: Reference to the messages collection. sensorDataCollection: Reference to the sensorData collection.	
Methods	addUser(userId,userData)	Adds a new user to Firestore.
	getUser(userId)	Retrieves user details from Firestore.
	sendMessage(senderId, receiverId, messageData)	Stores a chat message between users.
	getMessages(conversationId)	Fetches all messages in a conversation.
	storeSensorData(sensorId, data)	Saves IoT sensor readings in Firestore.
	getSensorData(sensorId)	Retrieves stored sensor data for analysis.

3.2.17. Node.js (Express) - FirebaseDatabaseController.js

Description	**Handles Firebase Firestore interactions on the backend, including API endpoints for user data, messages, and alerts.**	
Attributes	db: Firestore instance for database access. userRef: Reference to the users collection. messageRef: Reference to the messages collection. alertRef: Reference to the alerts collection.	
Methods	addUser(req, res)	Adds a user to Firestore from an API request.
	getUser(req, res)	Fetches a user's details from Firestore.
	sendMessage(req, res)	Stores a new message in Firestore.
	getMessages(req, res)	Retrieves all messages in a conversation.
	storeAlert(req, res)	Saves an alert triggered by an IoT event.
	getAlerts(req, res)	Fetches all stored alerts.

3.2.18. React - FirebaseService.js

Description	**A Firebase service file that handles Firestore interactions for the web app.**	
Attributes	db: Firestore instance. userCollection: Reference to users. messageCollection: Reference to messages.	
Methods	addUser(userData)	Adds a user to Firestore.
	getUser(userId)	Retrieves user details.
	sendMessage(senderId,receiverId, messageData)	Saves messages in Firestore.
	getMessages(conversationId)	Fetches messages fort he chat UI.

Firebase Security Rules:

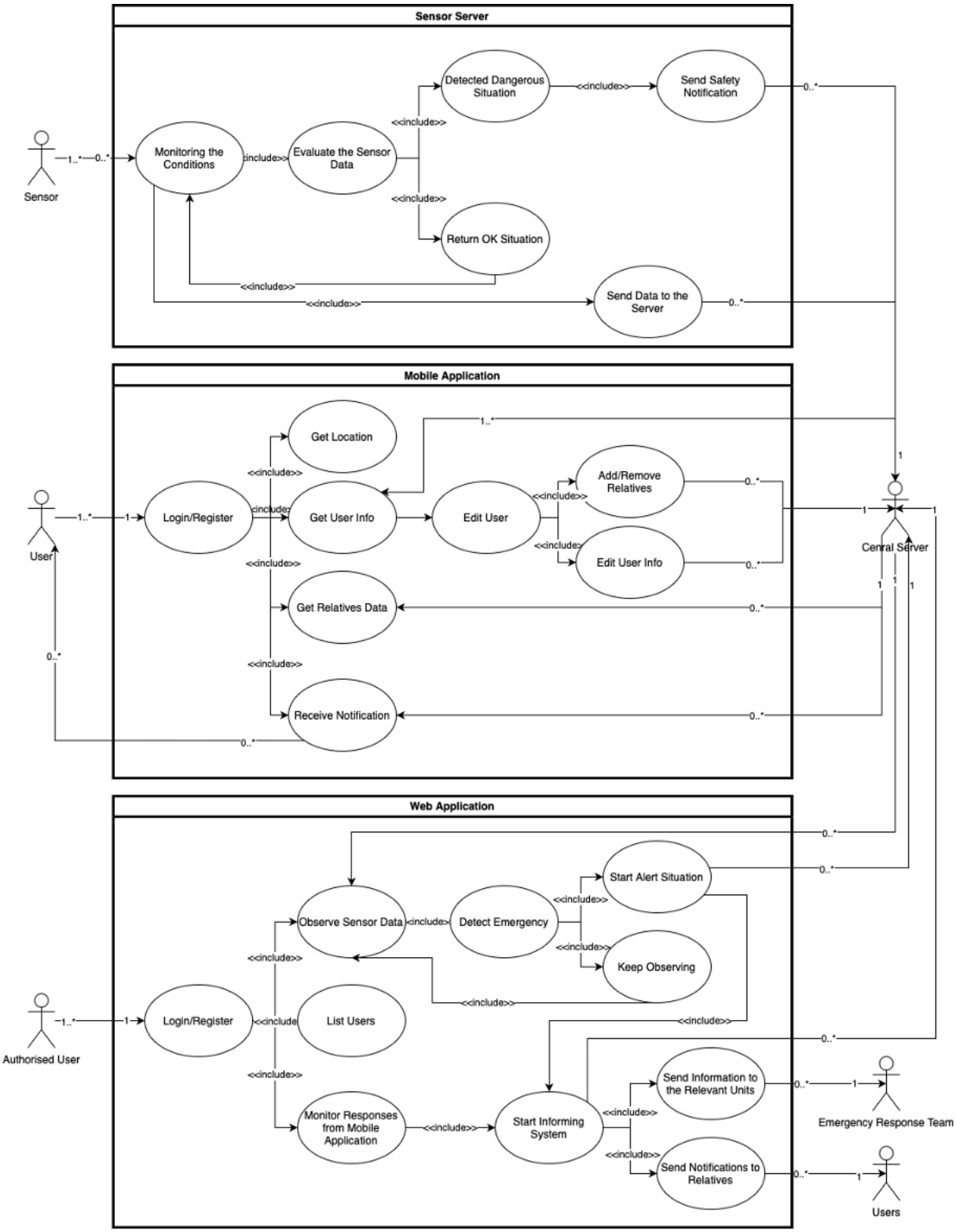
Users can read and write their own data.

Messages are accessible only when authenticated.

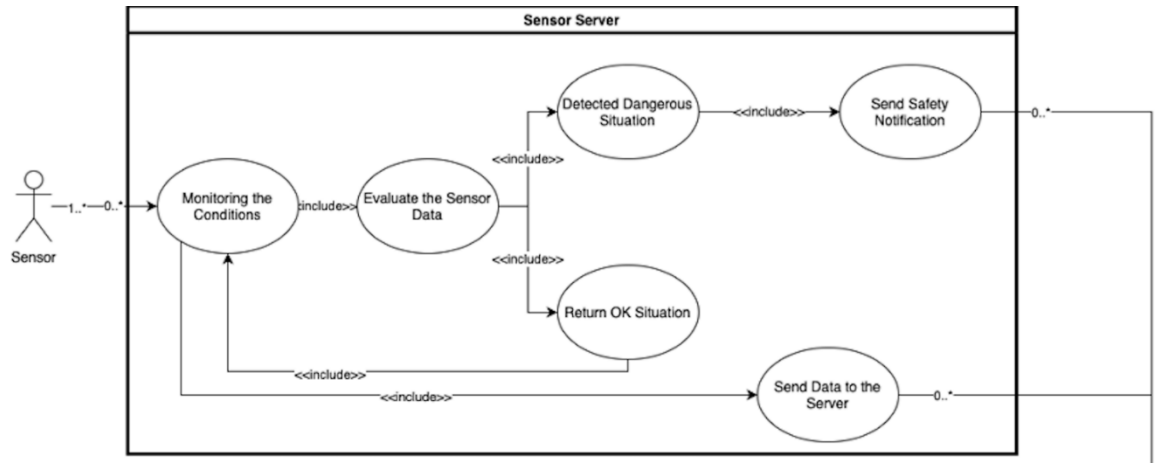
Sensor data can only be written by authorized users.

4. System Models

4.1. Use-Case Model



4.1.1. Sensor Server



Use Case	Monitoring the Conditions
Actor	Sensor
Description	The sensor is constantly monitoring environmental conditions such as temperature, water levels, seismic waves, or other safety factors. It captures real-time data and updates it at frequent intervals.
Flow of Events	The sensor starts collecting environmental data. The data is temporarily stored in the sensor memory. The sensor transmits the gathered data for further processing.

Use Case	Evaluate the Sensor Data
Actor	Sensor
Description	The sensor analyzes the collected data to determine whether conditions are normal or if an anomaly exists.
Flow of Events	The sensor receives new data readings. The system processes the data using pre-defined thresholds. The system identifies the condition to be either safe or dangerous.

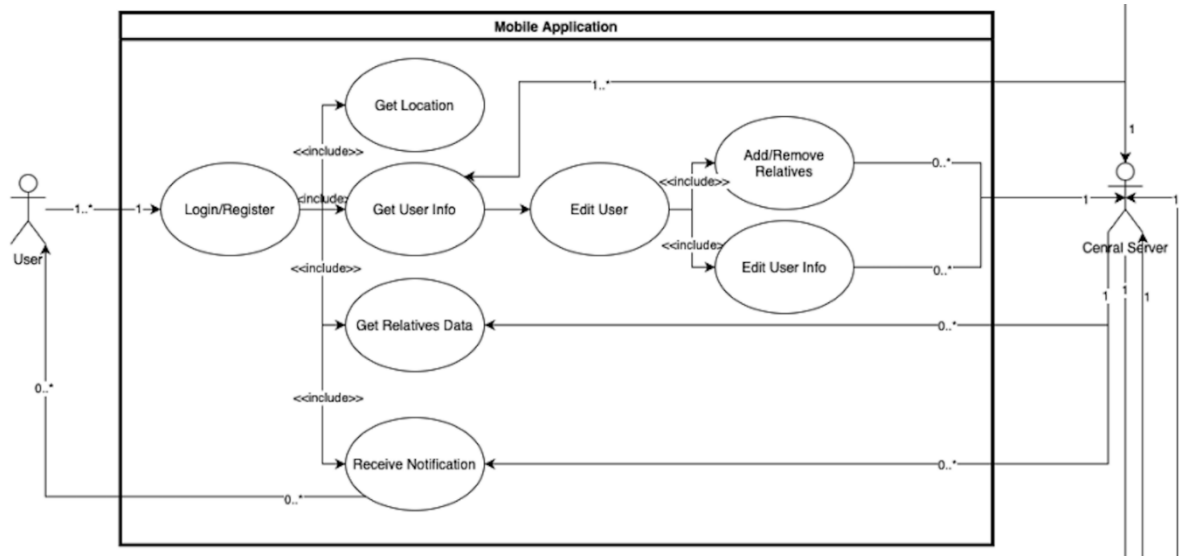
Use Case	Detected Dangerous Situation
Actor	Sensor
Description	If sensor data that has been measured goes beyond safety thresholds, the system identifies a dangerous condition.
Flow of Events	The system compares sensor readings with pre-defined safety thresholds. When an anomaly is detected, it is classified as an emergency. The system sends a notification to concerned users.

Use Case	Return OK Situation
Actor	Sensor
Description	If sensor data does not indicate a threat, the system returns an "OK" status.
Flow of Events	The system evaluates sensor data. The system determines that conditions are safe. The system returns an "OK" status.

Use Case	Send Safety Notification
Actor	Sensor
Description	When a dangerous situation is detected, the sensor sends an alert message to the Central Server.
Flow of Events	The system confirms an emergency. A notification message is generated. The system sends the notification to the Central Server for further action.

Use Case	Send Data to the Server
Actor	Sensor
Description	The sensor sends collected data to the Central Server for storage and further processing.
Flow of Events	The sensor collects the latest reading data. The data is forwarded to the Central Server. The server acknowledges the receipt and stores the data.

4.1.2. Mobile Application



Use Case	Login/Register
Actor	User
Description	Users can log in or register an account to access emergency features.
Flow of Events	The user creates a new account or enters their login information. Credentials are verified by the system. Access is either granted or denied by the system.

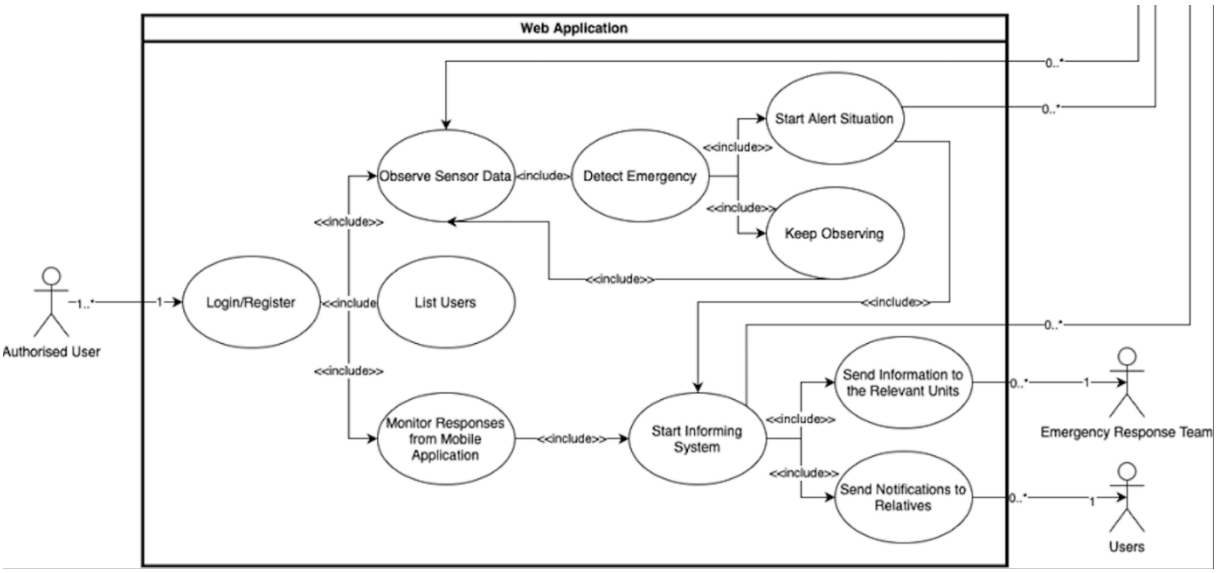
Use Case	Get Location
Actor	User
Description	The mobile app retrieves the user's real-time location.
Flow of Events	The app requests GPS permission. The system retrieves the user's coordinates. The system stores or displays the location.

Use Case	Edit User
Actor	User
Description	Users can edit their profile by modifying their necessary information. They can add, remove and edit the relative information while they are modifying personal data.
Flow of Events	The user selects the option to edit profile details. The user modifies personal and relative information. The system updates the profile and confirms the changes.

Use Case	Get User Info
Actor	User
Description	The user accesses their stored profile information.
Flow of Events	The user requests to view their information. The data is retrieved from the database by the system. The data is shown by the system.

Use Case	Receive Notification
Actor	User
Description	The user receives emergency alerts via the mobile app.
Flow of Events	The system detects an emergency. The system generates a notification. The user receives and views the alert.

4.1.3. Web Application



Use Case	Login/Register
Actor	Authorized User
Description	Authorized personnel log in to monitor sensor data and emergency alerts.
Flow of Events	The authorized user enters credentials. The system verifies credentials. The system grants access.

Use Case	List Users
Actor	Authorized User
Description	The system provides a list of registered users to the authorized user.
Flow of Events	A list of users is requested by the authorized user. User data is retrieved by the system. It shows the list of users.

Use Case	Observe Sensor Data
Actor	Authorized User
Description	The authorized user monitors real-time sensor data.
Flow of Events	The user signs in. Sensor data is retrieved by the system. Real-time readings are shown by the system.

Use Case	Detect Emergency
Actor	Authorized User
Description	The user reviews sensor data and confirms if an emergency exists.
Flow of Events	Sensor readings are processed by the system. The user examines trends in the data. The user marks the situation as an emergency or safe.

Use Case	Start Alert Situation
Actor	Authorized User
Description	The user triggers an emergency alert.
Flow of Events	The user reviews sensor alerts. The user confirms an emergency. The system triggers notifications.

Use Case	Keep Observing
Actor	Web Application (System)
Description	Once an emergency is identified, the system keeps checking to assess the ongoing situation.
Flow of Events	The system continues to monitor sensor data after identifying an emergency. If required, the system modifies answers and logs modifications.

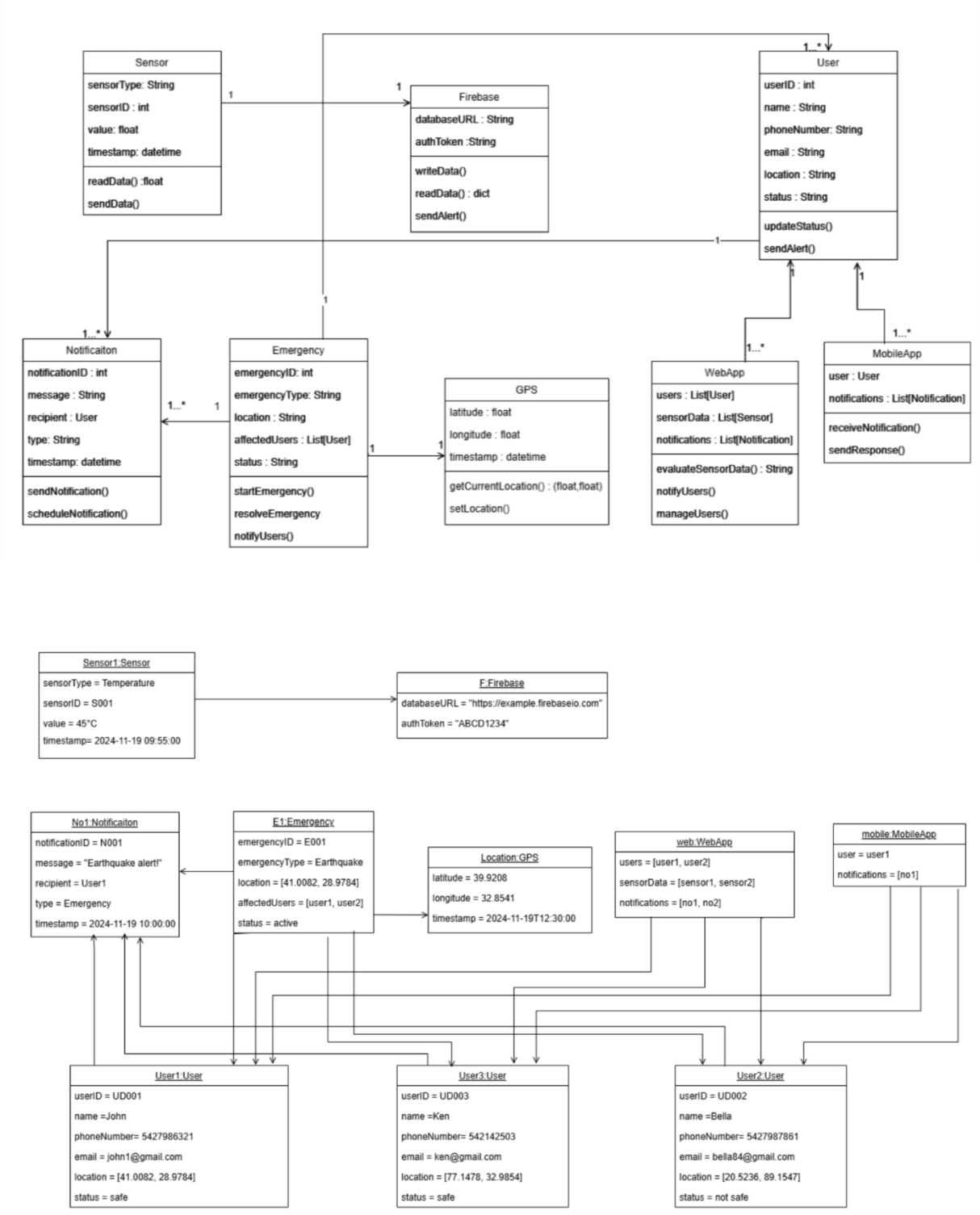
Use Case	Monitor Responses from Mobile Application
Actor	Authorized User
Description	The system allows the user to track responses from the mobile application.
Flow of Events	Response data from mobile users is collected by the system. The responses are examined by the authorized user.

Use Case	Start Informing System
Actor	Web Application (System)
Description	The system begins the process of notifying relevant individuals and authorities about an emergency.
Flow of Events	An emergency is detected by the system. The system begins notifying the appropriate parties.

Use Case	Send Information to the Relevant Units
Actor	Emergency Response Team
Description	The system notifies emergency response teams about the detected emergency.
Flow of Events	The system generates an emergency report. The report is sent to emergency response teams.

Use Case	Send Notifications to Relatives
Actor	Web Application (System), Users (Relatives)
Description	The system alerts the affected user's listed emergency contacts.
Flow of Events	Emergency contacts are retrieved by the system. Alert messages are sent by the system. The alerts are received and acknowledged by family members.

4.2. Object and Class Models



This class model represents a comprehensive system designed to integrate various components such as sensors, user management, notifications, emergency handling, GPS functionality, and

both web and mobile applications. The system is set up to gather, handle, and evaluate data while giving users real-time warnings and alerts, especially during emergencies.

The Sensor class, which represents data-collecting devices, is essential to the system. Each sensor contains characteristics that aid in precisely recognizing and capturing the data, such as sensorType, sensorID, value, and timestamp. The system may retrieve and send sensor data using the readData() and sendData() methods, guaranteeing that the data is always current and available.

The Firebase class serves as the backend for the system, providing essential services for data management. It guarantees safe and effective data processing using properties like databasesURL and authToken. The writeData(), readData(), and sendAlert() methods are essential for real-time data processing and notification because they allow the system to save, retrieve, and send alerts based on the data gathered.

The User class, which contains characteristics like userID, name, phoneNumber, email, location, and status, is in charge of managing users. The system can monitor each user's current status thanks to this class. To keep users informed and their statuses up to date, user information is managed and alerts are sent using the updateStatus() and sendAlert() methods.

The Notification class is designed to handle communication with users. It has features that aid in creating and maintaining notifications, such as notificationID, message, receiver, type, and timestamp. The system can deliver planned or instant notifications using the sendNotification() and scheduledNotification() methods, guaranteeing prompt communication.

The Emergency class is used in emergency situations. EmergencyID, emergencyType, location, affectedUsers, and status are among its properties. The system can efficiently handle emergency situations thanks to the functions startEmergency(), resolveEmergency(), and notifyUsers(), which guarantee that all impacted users are informed and that the matter is resolved quickly.

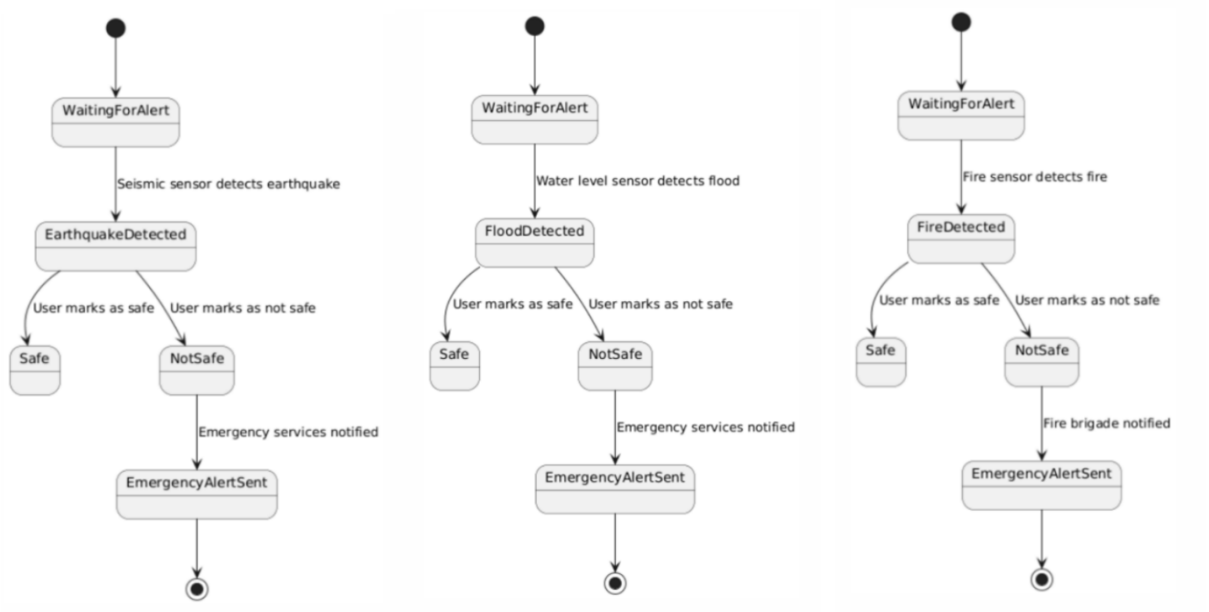
Latitude, longitude, and timestamp are among the location-based services offered by the GPS class. The system can retrieve and set location data using the getCurrentLocation() and setLocation() methods, which is essential for location-based alerts and emergency responses.

The system's web application component is represented by the WebApp class. It has methods like evaluateSensorData(), notifyUsers(), and manageUsers() as well as characteristics like users, sensorData, and notifications. This class is in charge of handling user information, managing and evaluating sensor data, and alerting users.

Lastly, the user interface of a mobile application is represented by the MobileApp class. It contains methods like sendResponse() and receiveNotification() as well as characteristics like user and notifications. An interactive and responsive user experience is provided by this class, which guarantees that users can receive notifications and reply to them.

4.3. Dynamic Models

4.3.1. State Diagrams



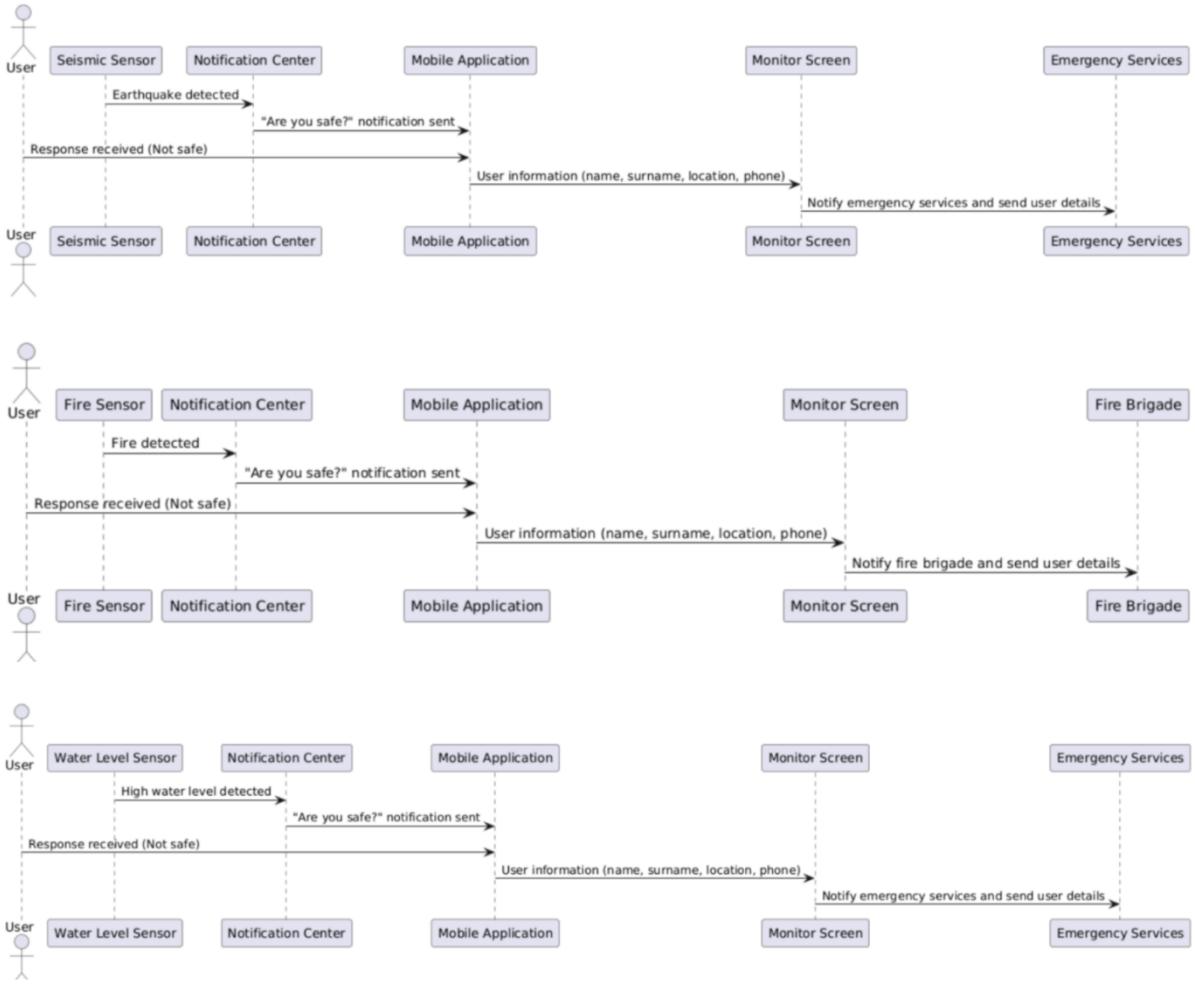
The process for handling various emergencies—such as earthquakes, floods, and fires—that are identified by sensors is shown in these state diagrams. Every diagram starts with the system in the WaitingForAlert state, which is inactive and keeping an eye out for any risks. Depending on the type of emergency, the system changes to a particular detection state—such as EarthquakeDetected, FloodDetected, or FireDetected—when a sensor detects one.

Once an emergency is detected, the system prompts users to indicate their safety status. Users have the option to designate themselves as Safe or NotSafe. The system keeps an eye on the issue without taking any further action if a user classifies themselves as safe. On the other hand, the system escalates the problem by alerting the relevant emergency agencies if a user tags themselves as NotSafe. Emergency services are contacted for floods and earthquakes, and the fire department is especially notified for fires. This step guarantees that those in need receive assistance as soon as possible.

After the emergency services are notified, the system transitions to the EmergencyAlertSent state, confirming that the alert has been successfully sent. This methodical approach guarantees

that the system can effectively manage a range of emergencies by offering seamless transitions between detection, user reaction, and emergency notification. By adhering to this process, the system can efficiently handle urgent circumstances, guaranteeing prompt reactions and improving user security.

4.3.2. Sequence Diagrams



These sequence diagrams show how different components interact and flow in reaction to various emergency scenarios that are sensed by sensors, such as floods, fires, and earthquakes. Every diagram highlights the progression of events from detection to response and has a similar format.

The process starts in Sequence Diagram I when an earthquake is detected by a seismic sensor. The user receives a notification asking, "Are you safe?" from the Notification Center and Mobile Application right away. In the event that the user answers "Not safe," the Monitor Screen and Emergency Services receive their information, which includes their name, last name,

location, and phone number. This guarantees that emergency personnel have the information they need to help the user right away.

Sequence Diagram II focuses on a fire emergency. When the Fire Sensor detects a fire, a safety alert is sent to the user via the Notification Center and Mobile Application. When a person selects "Not safe," their data is collected, sent to the Fire Department, and shown on the monitor screen. This enables the fire department to efficiently respond to the situation and swiftly get user information.

Sequence Diagram III deals with a flood scenario. When high water levels are detected by the water level sensor, the user receives a safety query from the Notification Center and mobile application. When a "Not safe" response is received, the system gathers the user's data and forwards it to Emergency Services and the Monitor Screen. This guarantees that, given the user's location and other information, emergency services can plan a response.

5. Glossary

Term	Definition
API (Application Programming Interface)	An interface that enables software components or systems to communicate with each other.
DOI (Digital Object Identifier)	A system that assigns persistent and unique identifiers to digital objects (e.g., articles, standards, datasets). It ensures that sources can be easily found and cited.
HTML (Hyper Text Markup Language)	A markup language used to define the structure and content of web pages. It is one of the fundamental technologies in the development of web-based interfaces.
UI (User Interface)	The visual interface that allows users to interact with a system (e.g., mobile application, web application). It is designed to make the system easy for users to operate.
IEEE SA (IEEE Standards Association)	The unit of IEEE that conducts and manages standardization activities. It is responsible for publishing and updating IEEE standards.
IoT (Internet of Things)	A technology in which physical objects (sensors, devices, vehicles, etc.) can exchange data with each other and systems over the internet. In this project, IoT sensors are used to collect environmental data.
V&V (Verification and Validation)	Processes for checking whether the software is developed correctly (verification) and whether it meets the user's needs (validation). It is defined in the IEEE 1012 standard.
JSON (JavaScript Object Notation)	A human-readable and machine-parsable data format used to represent structured data. It is widely used in API communication and data storage.
SDD (Software Design Description)	A document describing the details of the software design (architecture, interfaces, data structures, algorithms, etc.). It is defined in the IEEE 1016 standard.
SRS (Software Requirements Specification)	A document that comprehensively defines the functional and non-functional requirements of a software system. It is defined in the IEEE 830 standard.

6. References

[1] Berndtsson, M. P., Hansson, J., Olsson, E., & Lundberg, L. (2005). Software engineering research methods: trends and solutions. Springer.

[2] Cao, Y., & Yu, K. (2019). Research on the application of IoT technology in emergency management. International Journal of Disaster Risk Reduction, 37, 101173.

[3] Fenton, N., & Neil, M. (2014). Risk assessment and decision making in business and industry. Chapman and Hall/CRC.

[4] Jacobson, I., Booch, G., & Rumbaugh, J. (1999). The unified software development process. Addison-Wesley.

[5] Sommerville, I. (2016). Software engineering (10th ed.). Pearson.

[6] OMG (Object Management Group). (2017). OMG UML: Unified modeling language (OMG UML), Superstructure, V2.5.1. https://www.omg.org/spec/UML/2.5.1/

[7] Fowler, M. (2003). UML distilled: A brief guide to the standard object modeling language (3rd edition). Addison-Wesley Professional.

[8] Booch, G., Rumbaugh, J., & Jacobson, I. (2005). The unified modeling language user guide (2nd edition). Addison-Wesley Professional.

[9] IEEE Standards Association. (n.d.). IEEE Standards Association. [Website]. Retrieved from https://standards.ieee.org/

[10] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-oriented software architecture, volume 1: A system of patterns. John Wiley & Sons.

[11] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional.

[12] Newman, N. W., & Christozoglou, A. (2011). Database design and implementation. John Wiley & Sons.

[13] Powers, S., & Lau, M. C. (2018). Learning React: Functional web development with React and Redux. O'Reilly Media.