Bilkent University
Department of Computer Engineering

Senior Design Project

OverSeer

Low-Level Design Report

Talha Şen
Hakan Sivuk
Ahmet Berk Eren
Cevat Aykan Sevinç
Yusuf Nevzat Şengün

Supervisor: Ayşegül Dündar

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Contents

1 Introduction 3

1.1 Object Design Trade-Offs 3

1.2 Interface Documentation Guideline 4

1.3 Engineering Standards 5

1.4 Definitions, Acronyms, and Abbreviations 5

2 Packages 6

2.1 Internal Packages 6

2.1.1 Camera2 6

2.1.2 Android.Speech.TTS 7

2.2 External Packages 7

2.2.1 Pytorch 7

2.2.2 OpenCV 7

2.2.3 Mapbox.Directions 7

2.2.4 Mapbox.MapMatching 8

2.2.5 Mapbox.Geocoding 8

2.2.6 Mapbox.VectorTiles 8

3 Class Interfaces 9

3.1 Live Support Subsystem Service 9

3.1.1 Live Support Manager Subsystem Service 9

3.1.2 User Subsystem Service 11

3.1.3 Error Subsystem Service 12

3.1.4 Notification & Report Subsystem Service 13

3.2 Detection Support Subsystem Service 14

3.2.1 Camera Subsystem 15

3.2.2 Detection Subsystem 17

3.3 EventFlow Subsystem 20

3.4 Command Center Subsystem 22

3.5 Feedback Subsystem 24

3.6 UI Subsystem 25

3.7 Navigation Subsystem 28

4 References 32

Low-Level Design Report

OverSeer

1 Introduction

OverSeer is a mobile application that aims to remove barriers for visually impaired people. OverSeer navigates people for the places they want to go and warns them towards obstacles they face on their paths. Next, OverSeer provides live support to help them with any problem. They can ask for price information at a supermarket or they can just want a volunteer to show the correct location of an object with the help of live support.

In this report, having described the problem, we talk about our proposed system by describing the design trade-offs that affect the decisions, documentation, and engineering standards used for the project. Then, we give a detailed description of the low level design that includes packages and class interfaces along with the design patterns we use.

1.1 Object Design Trade-Offs

Our software architecture will be monolithic event-driven architecture. This is because monolithic architecture is simple to understand and implement while events enable developers to decouple subsystems. To control and distribute events, there will be an event center where each observable element reports itself. Any object can subscribe to any observable during its lifetime. As a result, when an event is raised, any subscriber can receive it instantly.

Event-driven designs are hard to debug and maintain. However, we believe a decoupled logic is cheaper to modify, thus, outweighing its disadvantages. We will use the component pattern to allow an object to subscribe to multiple events by controlling the full life cycles of concrete Observer objects. The object will have strict control over its components. These components will be encapsulated in a wrapper object, acting as an API for concrete objects to control their event flow.

One of the key design goals of our application is accessibility. Visually impaired users must be able to navigate the app on their own. Considering many accessible UI elements will be exposed to the user, their actions have to be controlled. To separate

accessibility logic, and button functionality, the command design pattern will be used.

Every button is an UIBase element that extends to IAccessibility. However, if accessible elements contain logic other than accessibility, the system would break SOLID principles. As a result, UI elements will signal to a command center whenever they are triggered. The command center will call the required command to execute the logic while the UI element is only responsible for providing an accessible interface to the user. As a result, it is cheaper to modify existing logic, and easier to add new accessible elements.

The application has a singleton service center. OverSeer has two main services: navigation and live stream. Moreover, there are support services such as object detection, voice recognition, accessibility. There should be only one service instance that must act as an API for the control objects. These services will be encapsulated in a singleton factory object, allowing ease of access to any service. Although this increases coupling, the services act as an API. Therefore, depreciated function calls could easily be refactored.

There will be a state center that employs the state machine pattern. Each state could call a relevant service upon state entry and exit. Furthermore, each state may act as a quality controller to ensure the system is not corrupted. Although the state pattern introduces additional complexity, it could be used to strictly control user input. This increases the reliability of the application.

1.2 Interface Documentation Guideline

The convention we follow for our interfaces can be described with the following regular expression: "I.*". anything that starts with an I is an interface. We use interfaces to dynamically distribute functionalities to distinct concrete objects. Following interfaces will be used in OverSeer:

- **IAccessible:** This interface ensures that a UI element is accessible. Upon touch, the UI element having this interface provides accessible support.
- **ICommand:** This interface provides polymorphism for multiple concrete objects. When an input or event trigger is received, the relevant ICommand is called. Allows the command pattern to be implemented.

- **IService:** There are many services in the application. IService provides a polymorphic collection for all the services for them to be encapsulated in a factory.
- **IState:** A concrete object implementing IState is necessarily strictly controlled by the state center service.
- **ISubscriber<T>:** Generic subscriber (observer). The concrete object implementing this can subscribe to any publisher (observable) of type T.
- **CustomEvent:** This is the base class for every custom event in OverSeer. Any ISubscriber can implement a concrete interface that extends to the custom event class to receive dispatched events.
- **SubscriberComponent<T>:** This is a generic base component class for allowing concrete objects to have multiple concrete objects implementing ISubscriber<T> so that they could receive multiple events. As a result, single responsibility and extensibility is introduced.

1.3 Engineering Standards

It is important for our application to comply with the standards of the software. Standards provide documentation to ensure reliability and quality [1]. Since OverSeer is developed for the visually impaired, it is critical to provide safety, security, and reliability.

Our goal is to comply with ISO standards. ISO ICS 35 provides standards over information technology. By the end of our development life-cycle, we aim to comply with software standards by following ISO ICS 35.080 documentation on quality assurance [2].

1.4 Definitions, Acronyms, and Abbreviations

- **Jitsi:** Jitsi is a Web-RTC application library. It also provides a video conferencing service that can be installed into the application main server. Jitsi connects users to each other with this conferencing server. The users can access this server via Jitsi Socket.

- **YOLOv5:** YOLO(You Only Look Once) is a famous object detection model to determine the location of certain objects and their classes. After it was first released in 2016, it is frequently used for real object detection tasks and the most current version is the v5. We finetuned the pre-trained YOLOv5 weights with our custom dataset.
- **Mapbox:** Mapbox is a mapping API that also provides native support for mobile. Mapbox provides several services as API, namely, ready to use map UIs with dynamic route drawings, navigation support, and search capabilities. These are the services that we use mainly to develop our application.
- **Firebase:** Firebase is a Backend-as-a-Service (Baas). It provides developers with a variety of tools and services to help them develop quality apps, grow their user base, and earn profit. It is built on Google's infrastructure. Firebase is categorized as a NoSQL database program, which stores data in JSON-like documents.

2 Packages

2.1 Internal Packages

2.1.1 Camera2

Camera2 is the internal hardware package Android provides after the Camera package got deprecated. From the official documentation: "This package models a camera device as a pipeline, which takes in input requests for capturing a single frame, captures the single image per the request, and then outputs one capture result metadata packet, plus a set of output image buffers for the request. The requests are processed in-order, and multiple requests can be in flight at once. Since the camera device is a pipeline with multiple stages, having multiple requests in flight is required to maintain full framerate on most Android devices" [3].

We use this package to obtain a manager and control the back camera of the phone (if it exists and if we get permission to it) to scan the environment, obtain frames and send it to the detection subsystem for prediction and further processes. It is an essential package to our detection support subsystem.

2.1.2 Android.Speech.TTS

Android system library for processing text input to output voice commands. This package is used by the feedback service to alert the user on various activities.

2.2 External Packages

2.2.1 Pytorch

Pytorch is a well-known machine learning library that allows developers to train and use several types of models for many tasks. In this project, Pytorch is used for training machine learning models (in Python). To load these trained model weights and perform detection tasks, Pytorch Android library is used. It is essential for us to use optimized algorithms of this library for a lower inference time and a more efficient system performance.

2.2.2 OpenCV

OpenCV is a well-known image processing library that allows developers to perform several operations on a visual object such as images, videos, etc. In this project, OpenCV is used to preprocess the frames that are captured through the camera of the phone and get them ready to be used by the detection models. For a lower inference time and a better performance, using optimized algorithms of this library is important for the project.

2.2.3 Mapbox.Directions

Mapbox.Directions API can provide point to point navigation routes with several alternatives. The given routes are combined of directions. The directions are separated by turns throughout the route. Additionally, the API provides a textual explanation of each direction along with the start and end coordinates.

2.2.4 Mapbox.MapMatching

Mapbox.MapMatching API provides the real-time information for the user. For example, it provides the direction of the user and matches between the direction and the route. Also, it can provide remaining duration information about the route.

2.2.5 Mapbox.Geocoding

Mapbox.Geocoding API provides basic search capabilities. Searching locations with a keyword its main capability. It provides several alternatives for the given keyword regarding user's current location, the popularity of places, and several other metrics. This API also provides special IDs (POI) to places. This way, a place can be represented by a unique ID (POI), and the location can be stored easily.

2.2.6 Mapbox.VectorTiles

Mapbox.VectorTiles API provides the required visual contents for the application. It includes the map itself mainly, but not limited to that. The API also provides real-time route and location drawing onto the map. This way, the user can follow the route visually, as well, which will be optional in our application.

3 Class Interfaces

3.1 Live Support Subsystem Service

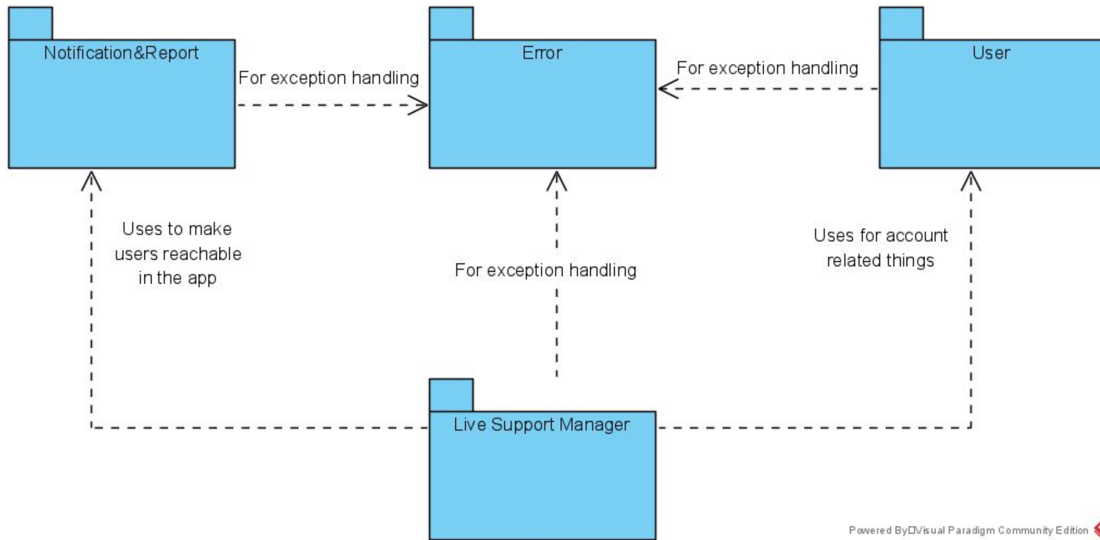

Fig 1. Live Support Subsystem Service

3.1.1 Live Support Manager Subsystem Service

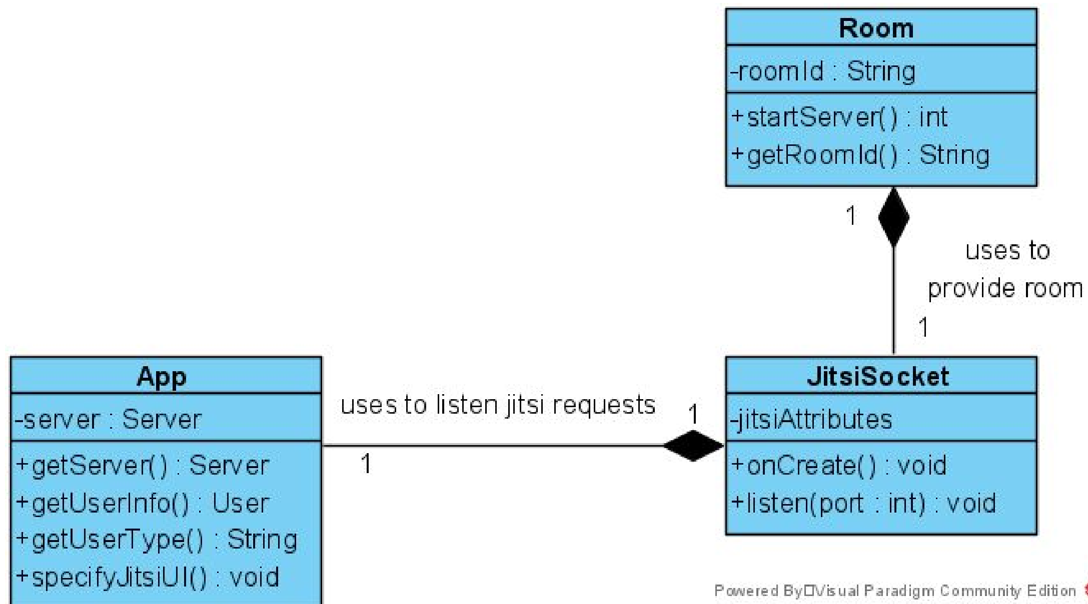

Fig 2. Live Support Manager Subsystem Service

LiveSupportManager subsystem provides a service to manage all the communication between the server and the client. It listens to requests done by the server or done by the app.

3.1.1.1 Room

Attributes:

- **String roomId:** The unique id of the room that connects the volunteer and the visually impaired user.

Methods:

- **private int startServer():** It starts the server and creates a room. It returns the id of the room.

3.1.1.2 JitsiSocket

Attributes:

- **jitsiAttributes:** This attribute includes the detailed information of the jitsi UI and jitsi server such as http information of the server.

Methods:

- **private void onCreate():** It starts the jitsi API.
- **public int listen(int port):** By using the server instance of the App, it makes socket listens on the specified events.

3.1.1.3 App

Attributes:

- **server:** It keeps the server details(Jitsi).

Methods:

- **public void specifyJitsiUI():** It receives the user’s information and orders the UI of the video meeting(Jitsi) according to them.

3.1.2 User Subsystem Service

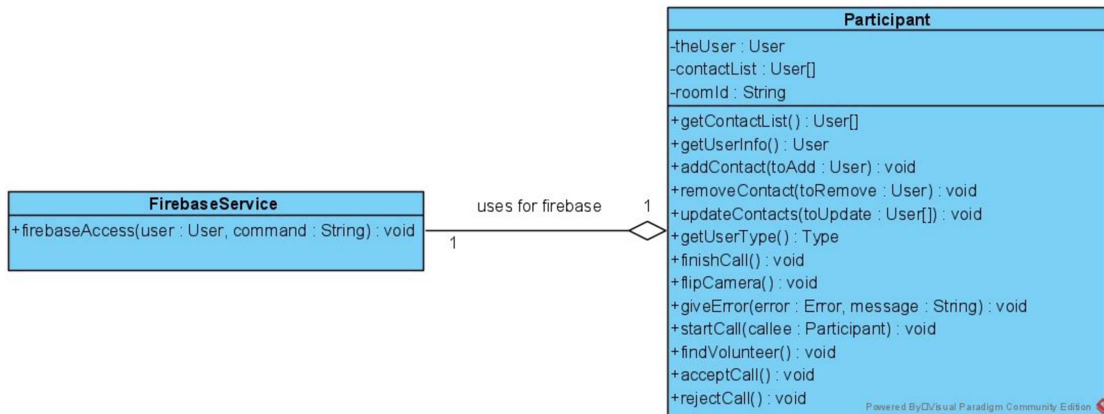

Fig 3. User Subsystem Service

The user subsystem provides a service to manage all the requests done by the user to the server and configure the UI.

3.1.2.1 FirebaseService

Methods:

- **public void firebaseAccess(User user, String command):** It sends commands to Firebase to change server databases and make call notifications.

3.1.2.2 Participant

Attributes:

- **theUser:** This attribute includes the detailed information of the user.
- **contactList:** It keeps saved contacts in order to call them.
- **roomId:** It keeps which room is assigned for the user.

Methods:

- **public void addContact(User toAdd):** It adds the "toAdd" user to the contact list.
- **public void removeContact(User toRemove):** It removes the "toRemove" user from the contact list.
- **public void updateContacts(User[] toUpdate):** It updates all of the "toUpdate" list both in UI and Firebase.

- **public void finishCall():** It finishes the current call and returns the main menu of the application.
- **public void flipCamera():** If the device has more than two cameras it changes the streaming camera.
- **public void giveError(ErrorModel error, String message):** If there is an error this function sends an error message.
- **public void startCall(Participant callee):** It starts the live support call according to callee.
- **public void findVolunteer():** It finds an available volunteer to the user.
- **public void acceptCall():** It accepts and starts the call if the volunteer presses the accept button.
- **public void rejectCall():** It rejects and closes the call if the volunteer presses the reject button.

3.1.3 Error Subsystem Service

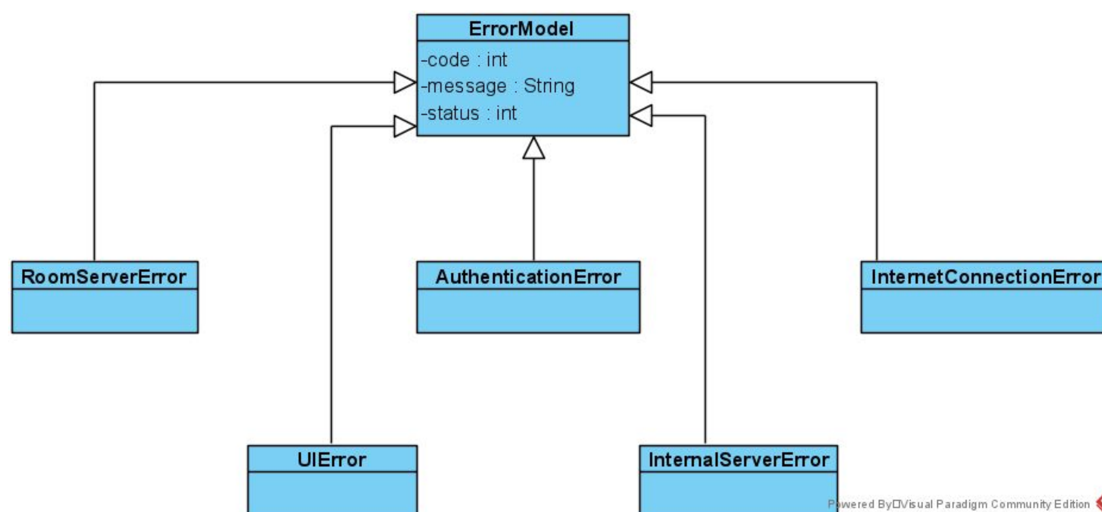

Fig 4. Error Subsystem Service

Error subsystem provides a service to handle all the exceptions or errors happening during the application. It provides a safe and robust environment in the online call.

3.1.3.1 ErrorModel

Attributes

- **int code:** Code specified for a particular error type.

- **String message:** Error message.
- **int status:** Status code for the error.

3.1.4 Notification & Report Subsystem Service

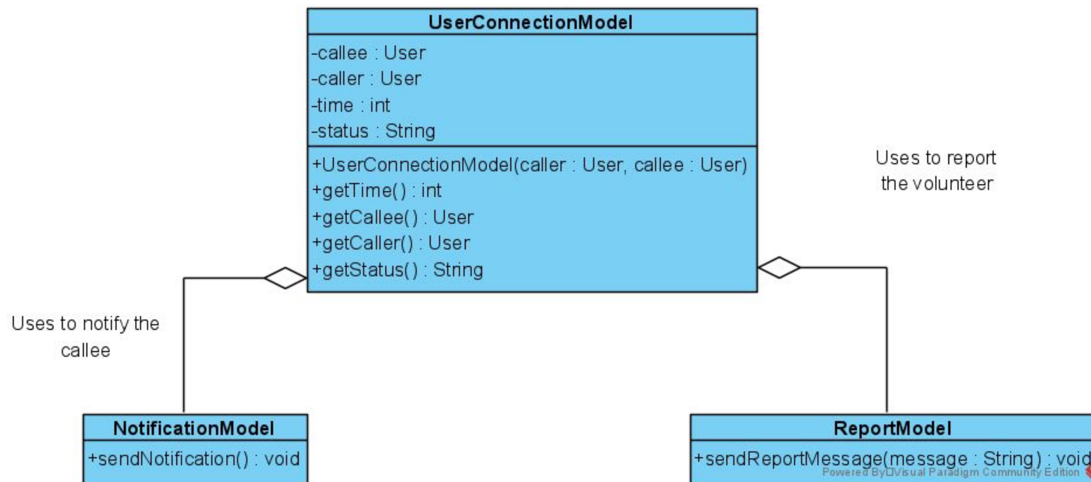

Fig 5. Notification & Report Subsystem Service

Notification & report subsystem provides a service to handle the notifications before the call and report the volunteer after the call.

3.1.4.1 UserConnectionModel

Attributes:

- **callee:** This attribute includes the detailed information of the callee user.
- **caller:** This attribute includes the detailed information of the caller user.
- **time:** It keeps the elapsed time of the call.
- **status:** It keeps the current status of the call.

Constructor:

- **public UserConnectionModel(Participant caller, Participant callee):** It creates the UserConnectionModel according to the caller and the callee.

3.1.4.2 NotificationModel

Methods:

- **public void sendNotification():** It sends a notification to the callee.

3.1.4.3 ReportModel

Methods:

- **public void sendReportMessage(String message):** If the user wants to report the other user this function sends a report message after the call.

3.2 Detection Support Subsystem Service

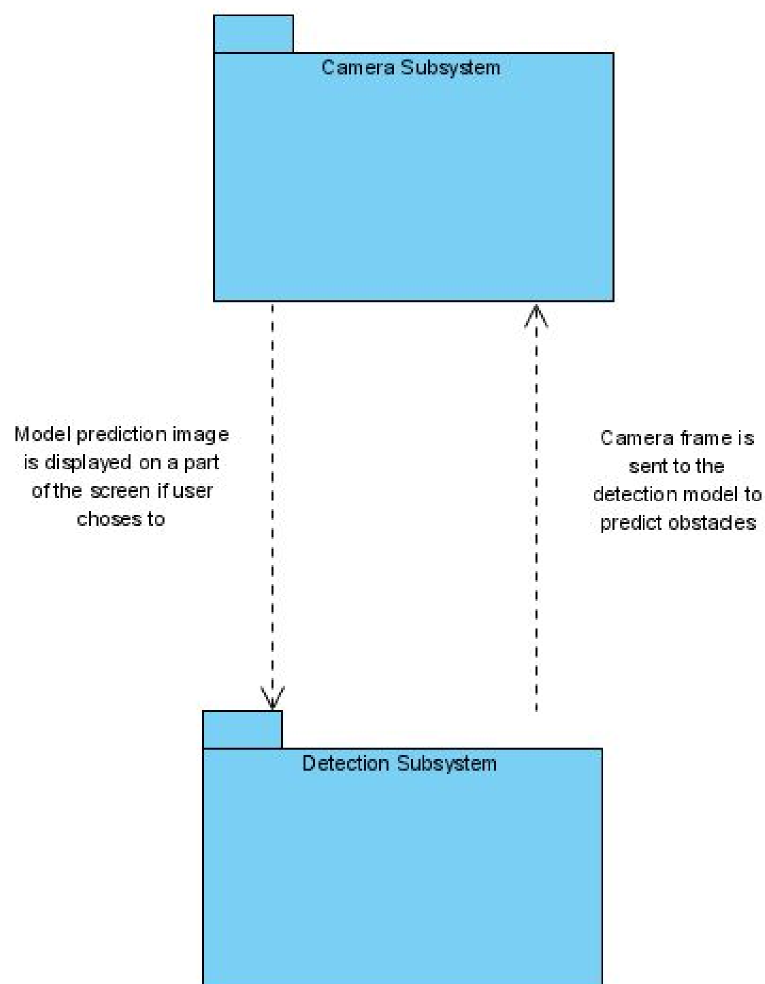

Fig 6. Detection Support Subsystem Service

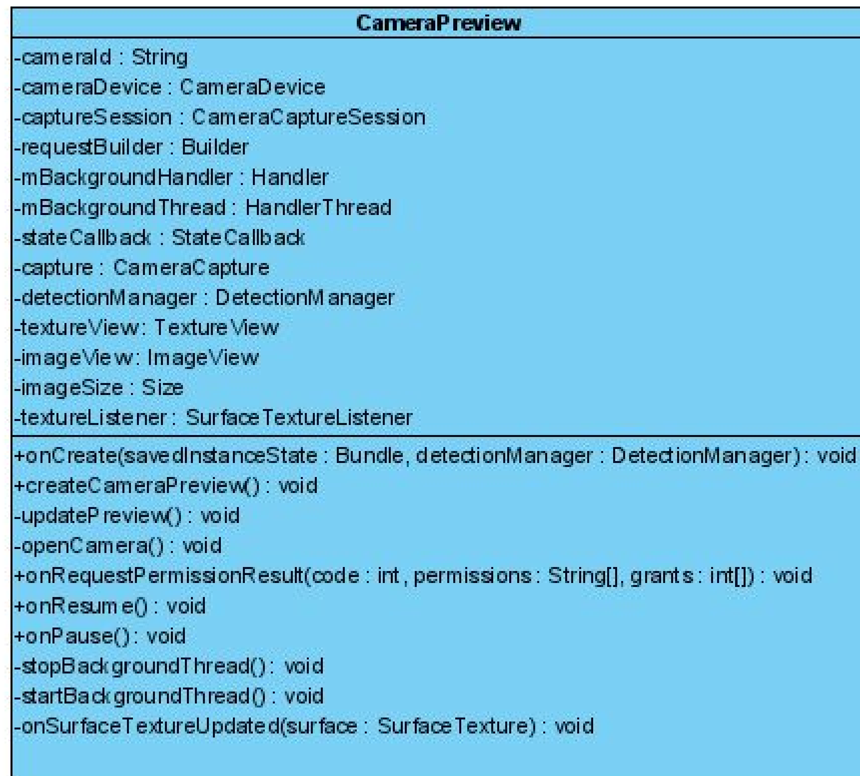

Fig 7. CameraPreview Class

3.2.1 Camera Subsystem

Camera subsystem has 2 main functionalities, providing a camera preview and capturing frames, and sending them to the model for prediction. Additionally, it can show the labeled prediction image on a part of the texture view (camera view). For this, the subsystem has a main class: CameraPreview. CameraPreview is responsible for providing a general camera functionality and it is responsible for regularly capturing camera frames and processing them together with the detection subsystem.

3.2.1.1 CameraPreview

Attributes:

- **String cameraId:** stores the id of the camera, which is used to identify the camera on the phone.
- **CameraDevice cameraDevice:** manager of the camera hardware on the phone.
- **CameraCaptureSession captureSession:** stores the camera capturing session at that given instance.

- **Builder requestBuilder:** builds the camera capture on a given surface.
- **Handler mBackgroundHandler:** handler for the camera preview process.
- **HandlerThread mBackgroundThread: a** thread for the camera preview process
- **StateCallback stateCallback:** handles the state changes of the camera hardware of the phone (open, disconnected, error etc.).
- **DetectionManager detectionManager:** The manager that controls all the detectors of the application. It uses the camera to gather frames for predictions and warnings event generations.
- **TextureView textureView:** holds the camera preview as a texture on the screen.
- **ImageView imageView:** holds the manipulated image (by detection subsystem).
- **Size imageSize:** holds the size of the image on the preview.
- **TextureListener textureListener:** listens to the events of the textureView, or in other words camera preview. Specifically, the frame changes.

Methods:

- **public void onCreate(Bundle savedInstanceState, DetectionManager detectionManager):** Initializes the camera preview activity (or in other words, the class itself as its an activity class) from the given detection manager class.
- **private void createCameraPreview():** Initializes the camera related attributes and creates a camera preview on the specified texture view.
- **private void updatePreview(Bundle savedInstanceState):** Updates the created camera preview with each camera configuration change.
- **private void openCamera(Bundle savedInstanceState):** Prepares and opens up the camera hardware of the phone if the camera access is granted.
- **public void onRequestPermissionsRequest(int code, String[] permissions, int[] grants):** Checks if a given permission access is for the camera permission.
- **protected void onResume():** If the activity is resumed (by user), the camera is opened again and the process thread is run again.
- **protected void onPause():** If the activity is paused(by user), camera is closed and the process thread is joined.
- **private void stopBackgroundThread():** Joins and stops the thread for the preview process.

- **private void startBackgroundThread():** Stars the preview process thread.
- **public void onSurfaceTextureUpdated(SurfaceTexture surface):** With each frame change, this method gets the frame from the surface, changes it to a bitmap and sends it to be processed. Also sets the imageView to the labeled prediction image if the user wants the option.

3.2.2 Detection Subsystem

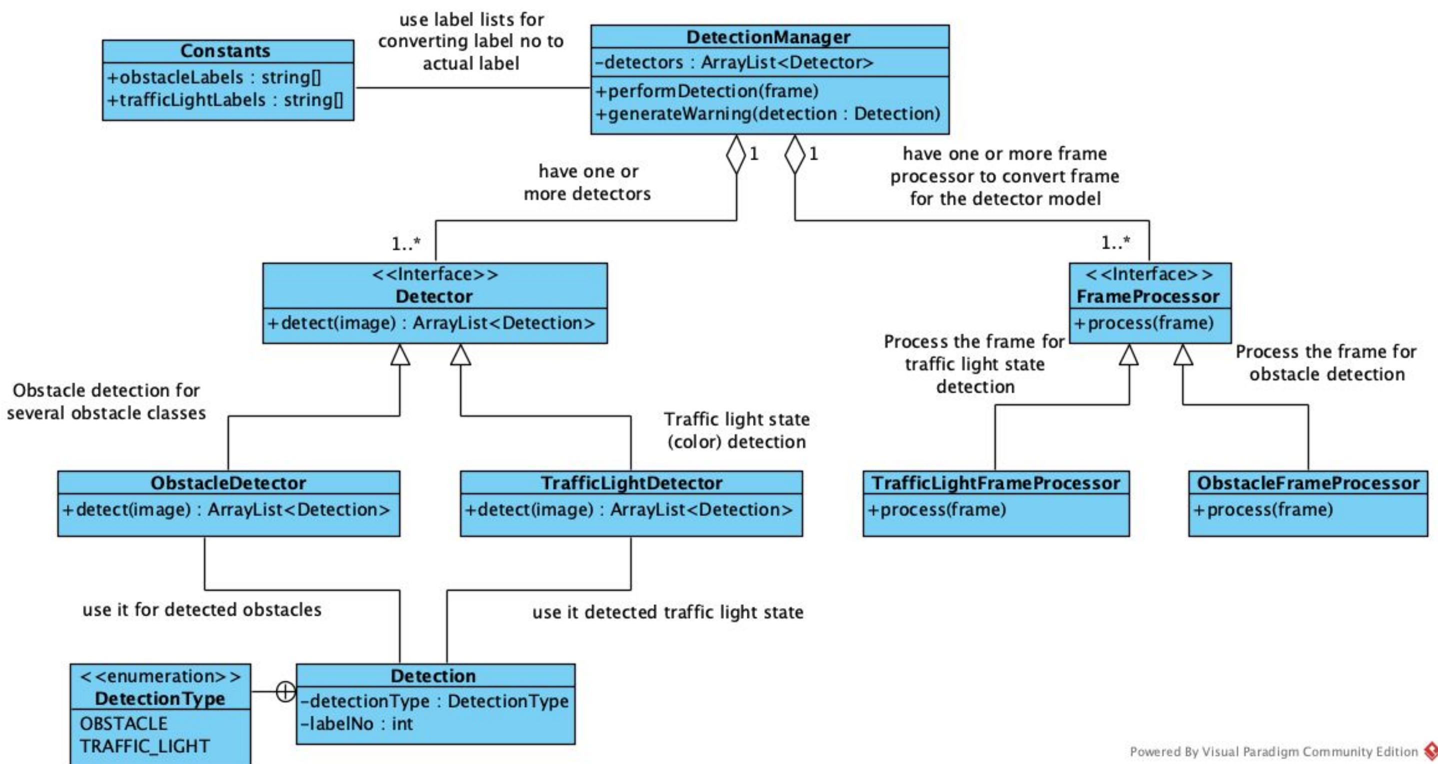

Fig 8. Detection Subsystem Service

3.2.2.1 DetectionManager

Attributes:

- **ArrayList<Detector> detectors:** List of detectors used for detection tasks such as traffic light detection and obstacle detection.

Methods:

- **public ArrayList< Detection > performDetection(frame):** It is called by CameraSubsystem to process the given frame and perform the detection tasks.

- **public void generateWarning(Detection detection):** It generates required warnings according to the given detection result. After it converts label no into the actual label text through Constant class. Then it sends a feedback event through the EventCenterAPI for generating sound output.

3.2.2.2 Detector

Methods:

- **public ArrayList<Detection> detect(image):** Abstract method to be overridden by the actual detector classes.

3.2.2.3 FrameProcessor

Methods:

- **public process(frame):** Abstract method to be overridden by the actual processor classes.

3.2.2.4 ObstacleDetector

Methods:

- **public ArrayList<Detection> detect(image):** It performs the obstacle detection task with the model and the given image. It returns a list of detection objects that are generated according to the output received from the obstacle detection model.

3.2.2.5 TrafficLightDetector

Methods:

- **public ArrayList<Detection> detect(image):** It performs the traffic light state detection task with the model and the given image. It returns a list of detection objects that are generated according to the output received from the traffic light state detection model.

3.2.2.6 ObstacleFrameProcessor

Methods:

- **public process(frame):** It prepares the given image for the obstacle detection model. It performs operations such as resizing, scaling, etc.

3.2.2.7 TrafficLightFrameProcessor

Methods:

- **public process(frame):** It prepares the given image for the traffic light state detection model. It performs operations such as resizing, scaling, etc.

3.2.2.8 Constants

Attributes:

- **string[] obstacleLabels:** List of labels in text format for the obstacles (such as person, car, fire hydrant, etc).
- **string[] trafficLight:** List of labels in text format for the traffic light states (red, yellow, green).

3.2.2.9 Detection

Attributes:

- **DetectionType detectionType:** It specifies type of the detection. It is either OBSTACLE or TRAFFIC_LIGHT.
- **int labelNo:** It specifies the number of the detected class.

3.3 EventFlow Subsystem

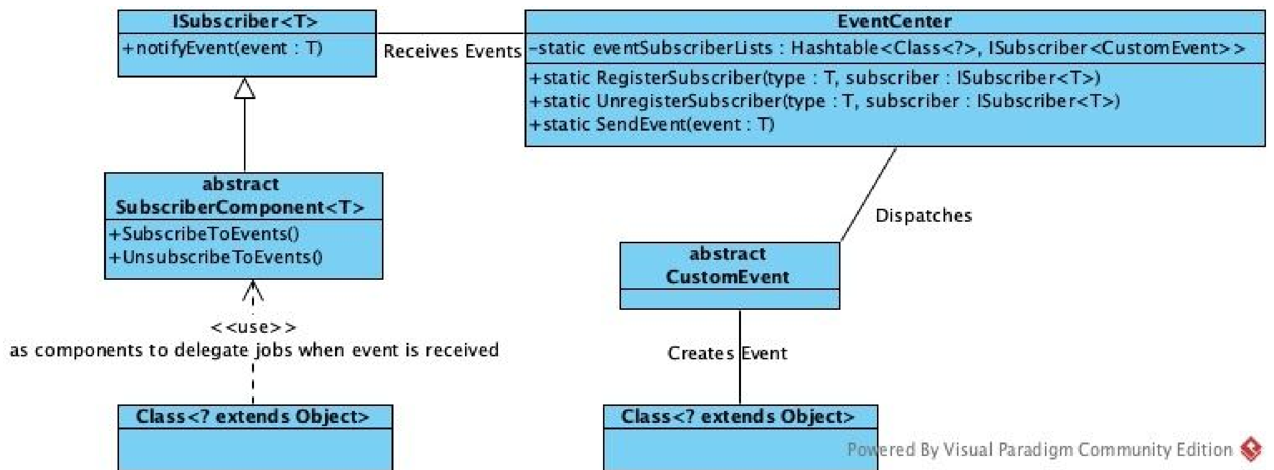

Fig 9. EventFlow Subsystem

EventFlow subsystem is responsible for managing and distributing events.

3.3.1 Abstract CustomEvent

Base for every custom defined event. Empty class.

3.3.2 EventCenter

Attributes:

- **static Hashtable< Class<? extends CustomEvent>, HashSet<ISubscriber<CustomEvent>> eventSubscriberLists:** Stores every observer of type custom event in the relevant set.

Methods:

- **public static <T extends CustomEvent> void RegisterSubscriber(Class<T> type, ISubscriber<T> subscriber):** Makes a subscriber (observer) unregister listening to the custom event T.
- **public static <T extends CustomEvent> void UnregisterSubscriber(Class<T> type, ISubscriber<T> subscriber):** Makes a subscriber (observer) unregister listening to the custom event T.

- **public static <T extends CustomEvent> void SendEvent(T event):** Sends event to the subscribers.

3.3.3 ISubscriber<T>

Methods:

- **public void notifyEvent(T event):** Contracts a concrete class to receive events of type T.

3.3.4 Abstract SubscriberComponent<T>

Methods:

- **public void SubscribeToEvents():** Subscribes to the events of type T.
- **public void UnsubscribeToEvents():** Unsubscribes to the events of type T.

3.3.5 Concrete Class<? extends Object>

Represents that any class having SubscriberComponents may receive events and any class can create any type of event.

3.4 Command Center Subsystem

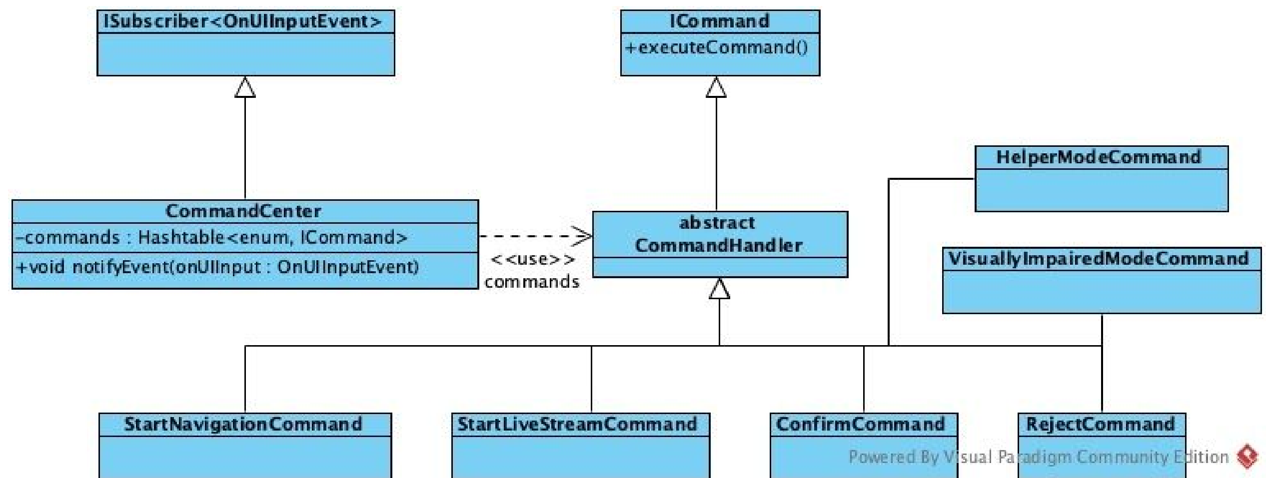

Fig 10. Command Center Subsystem

3.4.1 ICommand

Defines an interface for concrete classes to hide operations behind simple commands.

Methods:

- **public void executeCommand():** Contracts a concrete class to execute the received command.

3.4.2 Abstract CommandHandler

Base class for all concrete command objects. Command objects can receive commands and execute certain functionalities.

3.4.3 CommandCenter

Encapsulates and controls each concrete command object. Acts as an interface between command requests and execution of the command.

Attributes:

- **Hashtable< enum CommandType, ICommand command > commands:** Stores every command object related to UI inputs. Allows cheap extension of new commands.

Methods:

- **public void notifyEvent(OnUIInputEvent onInputEvent):** Receives UI events and calls requested command.

3.4.4 StartNavigationCommand

Command object for starting navigation service.

3.4.5 StartLiveStreamCommand

Command object for starting live stream service.

3.4.6 ConfirmCommand

Command object for confirming previously requested command.

3.4.7 RejectCommand

Command object for rejecting previously requested command.

3.4.8 HelperModeCommand

Command object for setting up the application for the helper users.

3.4.9 VisuallyImpairedModeCommand

Command object for setting up the application for the visually impaired users.

3.5 Feedback Subsystem

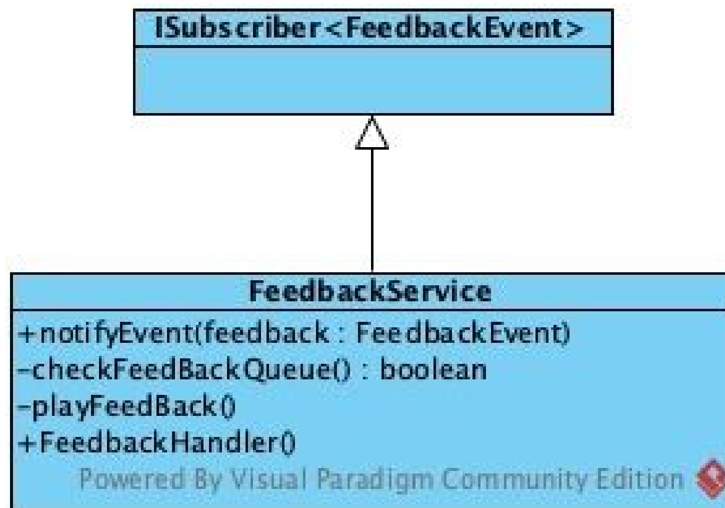

Fig 11. Command Center Subsystem

3.5.1 FeedBackService

Handles voice output requests by any concrete object. Receives requests as events.

Methods:

- **public void notifyEvent(FeedbackEvent feedback):** Receives feedback event. Checks if the system can play the feedback. Else, adds the feedback request to the queue or interrupts the current feedback according to the request.
- **private boolean checkFeedBackQueue():** Checks if the android system is available for playing sound.
- **private void playFeedBack():** Plays the requested feedback.

3.6 UI Subsystem

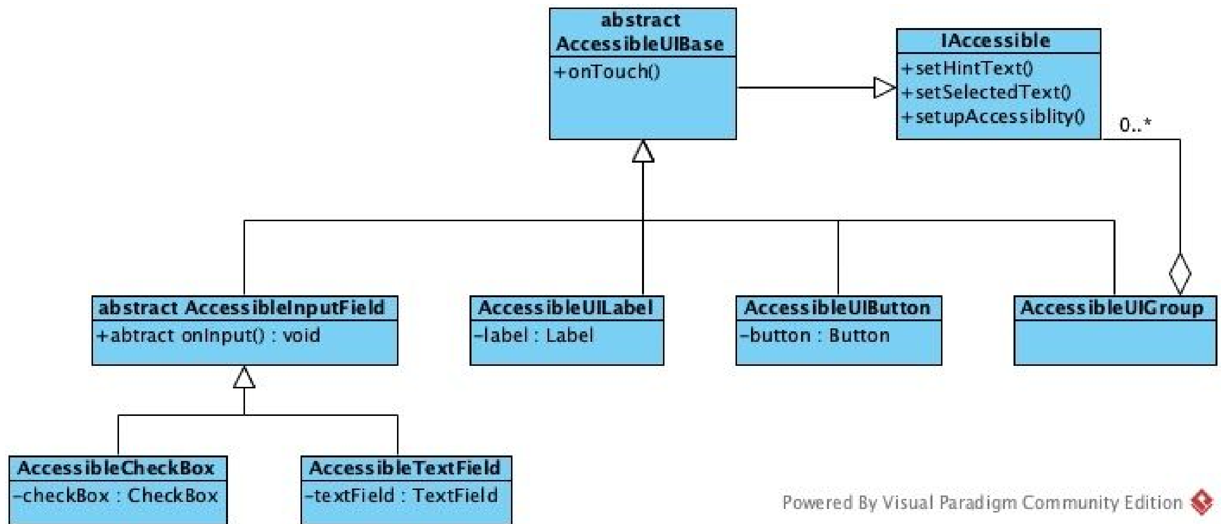

Fig 12. UI Subsystem Service

3.6.1 IAccessible

Methods:

- **public void setHintText():** Receives feedback event. Checks if the system can play the feedback. Else, adds the feedback request to the queue or interrupts the current feedback according to the request.
- **public void setSelectedText():** Checks if the android system is available for playing sound.
- **public void setupAccessibility():** Receives UI events and calls requested commands.

3.6.2 Abstract AccessibleUIBase

Methods:

- **public void onTouch():** Starts accessibility functionalities upon touch.

3.6.3 Abstract AccessibleInputField

Methods:

- **public void onInput():** Gets the input from the accessible ui element and makes the relevant input event for the command center.

3.6.4 AccessibleCheckBox

Base class for accessible generic checkbox. The engineer may use this class to create concrete checkboxes to get input on a variety of functionalities.

Attributes:

- **Checkbox checbox:** Checkbox ui component.

3.6.5 AccessibleTextField

Base class for accessible generic text field. The engineer may use this class to create concrete text fields to get input on a variety of functionalities.

Attributes:

- **TextField textField:** TextField ui component.

3.6.6 AccessibleUILabel

Base class for accessible generic label. The engineer may use this class to create concrete labels to navigate the user in-app.

Attributes:

- **Label label:** Label UI component.

3.6.7 AccessibleUIButton

Base class for the accessible generic button. The engineer may use this class to create concrete buttons to get input on a variety of functionalities.

Attributes:

- **Button button:** Button UI component.

3.6.8 AccessibleUIGroup

Groups multiple accessible elements to make them one accessible group component.

Attributes:

- **IAccessible accessibleComponents:** Accessible children components

3.7 Navigation Subsystem

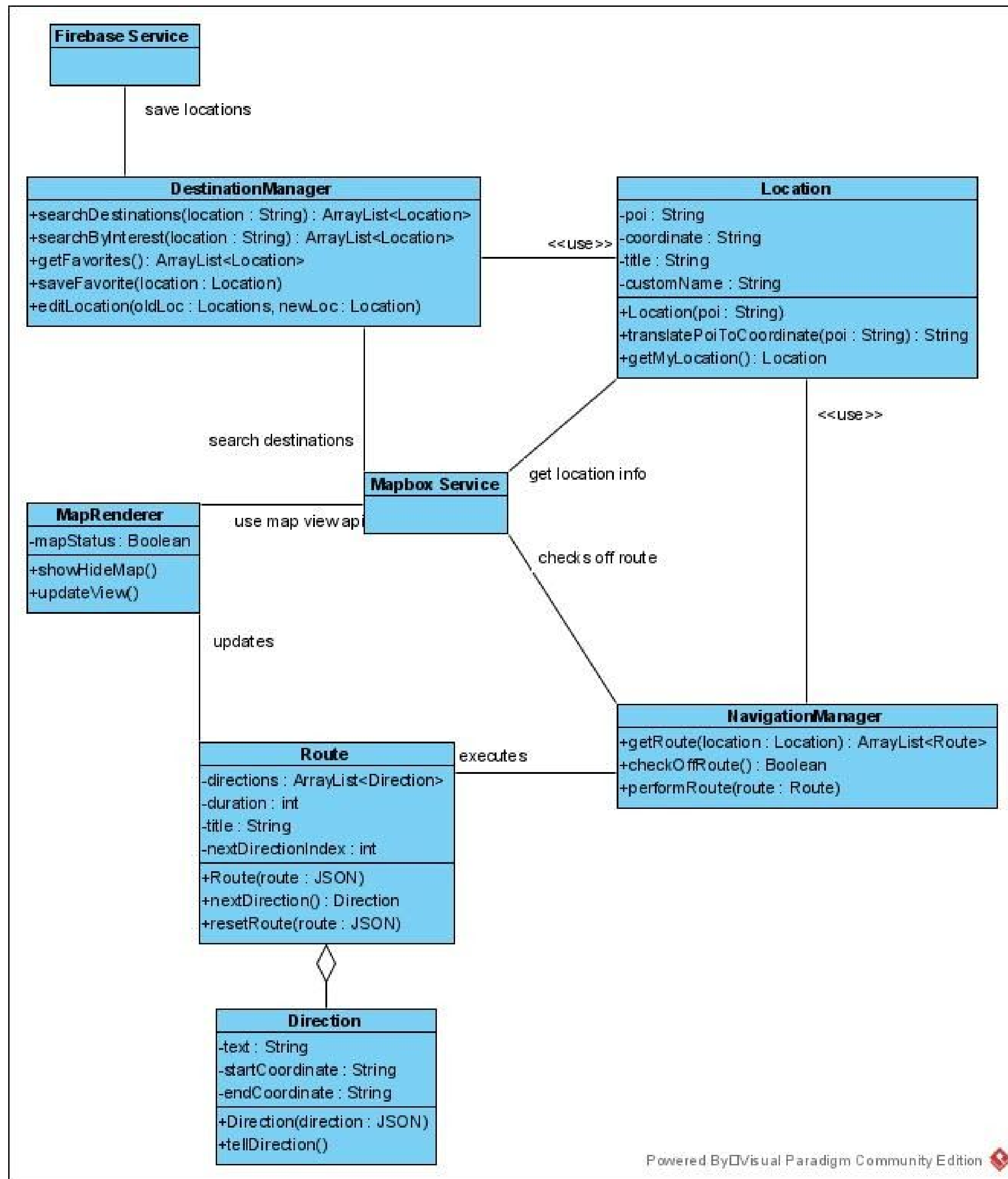

Fig 13. Navigation Subsystem Service

3.7.1 DestinationManager

Methods:

- **public Arraylist<Location> searchDestionations(String location):** Returns possible destinations according to the given string, so it tries to match destinations with the input by using Mapbox API.

- **public ArrayList<Location> searchByInterest(String location):** Returns possible destinations according to the given string. Used when a specified generic search is desired, namely, supermarket. This also communicates to Mapbox API.
- **public ArrayList<Location> getFavorites():** Returns the favorite locations of a user. Communicates to firebase to get stored data.
- **public void saveFavorite(Location location):** Saves a favorite location for the current user. Communicates to firebase to store data.
- **public void editFavorite(Location oldLoc, Location newLoc):** Edits a favorite location of the current user. Communicates to firebase.

3.7.2 NavigationManager

Methods:

- **public ArrayList<Route> getRoute(Location location):** Returns possible routes from my current direction to the specified location.
- **public Boolean checkOffRoute():** Checks if the user obeys the current route. If the user follows the current route, it returns true, otherwise false.
- **public void performRoute(Route route):** Performs the given route. In each step, it executes the next direction of the current route.

3.7.3 MapRenderer

Attributes:

- **private Boolean mapStatus:** Indicates if the map is visible or not.

Methods:

- **public void showHideMap():** Toggles the visibility of the map.
- **public void updateView():** Updates the current state of the map regarding the current route, direction, location and so on. It interacts with the Mapbox API.

3.7.4 Route

Attributes:

- **private ArrayList<Direction> directions:** Keeps the individual directions of the route.
- **private int duration:** Indicates the total duration of the current route.
- **private String title:** Indicates the title of the route.
- **private int nextDirectionIndex:** Indicates the last executed direction in the current route.

Methods:

- **public Route(JSON route):** The constructor. Parses the JSON and forms the directions in an appropriate form.
- **public Direction nextDirection():** Returns the next direction to be performed by the related class.
- **public void resetRoute(JSON route):** When the user goes off-route, the current route is resetted and updated with the new one.

3.7.5 Direction

Attributes:

- **private String text:** Indicates the textual representation of the direction.
- **private String startCoordinate:** Indicates the coordinate of the start location of this direction.
- **private String endCoordinate:** Indicates the coordinate of the end location of this direction.

Methods:

- **public Direction(JSON direction):** The constructor. Parses the given JSON into an appropriate form.
- **public void tellDirection():** Interacts with the related method of the related class to tell the direction by sound.

3.7.6 Location

Attributes:

- **private String poi:** Indicates the POI(Point Of Interest) representation of the location.
- **private String coordinate:** Indicates the longitude/latitude representation of the location.

- **private String title:** The formal title of the location.
- **private String customName:** The custom name of the location that may be given by the user.

Methods:

- **public Location(String poi):** The constructor. Parses the POI and sets the related attributes accordingly.
- **public String translatePoiToCoordinate(String poi):** translates a given POI to longitude/latitude.
- **static public Location getMyLocation():** Returns the current location of the user. Uses the GPS of the user's phone and interact with the mapbox API.

4 References

[1] "What are Standards," *library.rose,* Feb. 01, 2021. [Online]. Available: https://library.rose-hulman.edu/c.php?g=104543&p=888557. [Accessed: 08.02.2021].

[2] "ISO Standards," *iso,* [Online]. Available: https://www.iso.org/standards-catalogue/browse-by-ics.html. [Accessed: 08.02.2021].

[3] "Camera2," *developer.android,* [Online]. Available: https://developer.android.com/reference/android/hardware/camera2/package-summary. [Accessed: 08.02.2021].