Architecture Design

SE 606: Software Design and Architecture

Submitted to:

Kishan Kumar Ganguly
Lecturer
Institute of Information Technology,
University of Dhaka

Submitted by:

Group 1
Taslima Akbar Keya(BSSE0901)
Md. Alamgir Kabir(BSSE0907)
Noshin Tahsin Saaj(BSSE0914)
Md. Samsarabbi(BSSE0916)
Mridha Md. Nafis Fuad (BSSE0920)

Date: 6 November 2019

Chapter1: Introduction

1.1 Architectural design:

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Architectural design is the preliminary blueprint from which software is constructed. To develop the architectural design for E-Shongee app, we have followed the steps given below:

1. Representing the System in Context
2. Defining Archetypes
3. Refining the Architecture into Components

1.2 Component-level design

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established.

A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.
The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

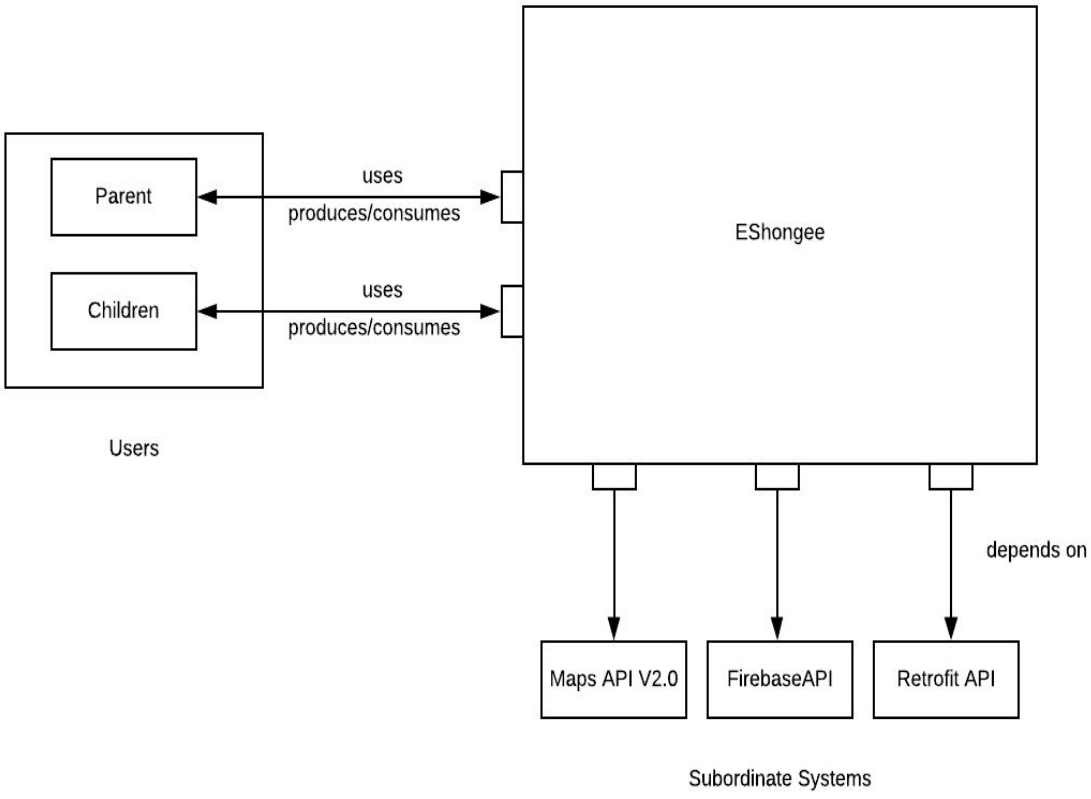
For conducting component level design, we have followed the following steps:

1. Transforming Analysis Class into Design Class
2. Specify message details when classes or components collaborate.

3. Describe processing flow within each operation in detail
4. Develop and elaborate behavioral representations for a class or component.
5. Elaborate deployment diagrams to provide additional implementation details.
6. Refactor every component-level design representation and always consider alternatives.

Chapter 2: Architectural Design

2.1 Context Diagram



We have developed a context diagram to model the manner in which the E-Shongee app interacts with entities external to its boundaries.

Systems that interoperate with the target system (the system for which architectural design is to be developed) are represented as

• **Superordinate systems**—those systems that use the target system as part of some higher-level processing scheme. There is no super-ordinate system for E-Shongee.

• **Subordinate systems**—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality. Maps API V2.0, Firebase API, and Retrofit API are used by the app and are shown as subordinate to it.

• **Peer-level systems**—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target System. There are no peer system for the app.

• **Actors**—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Parent and children are actors that are both producers and consumers of information used/produced by the E-Shongee app.

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

2.2 Archetypes Diagram

Defining Archetypes

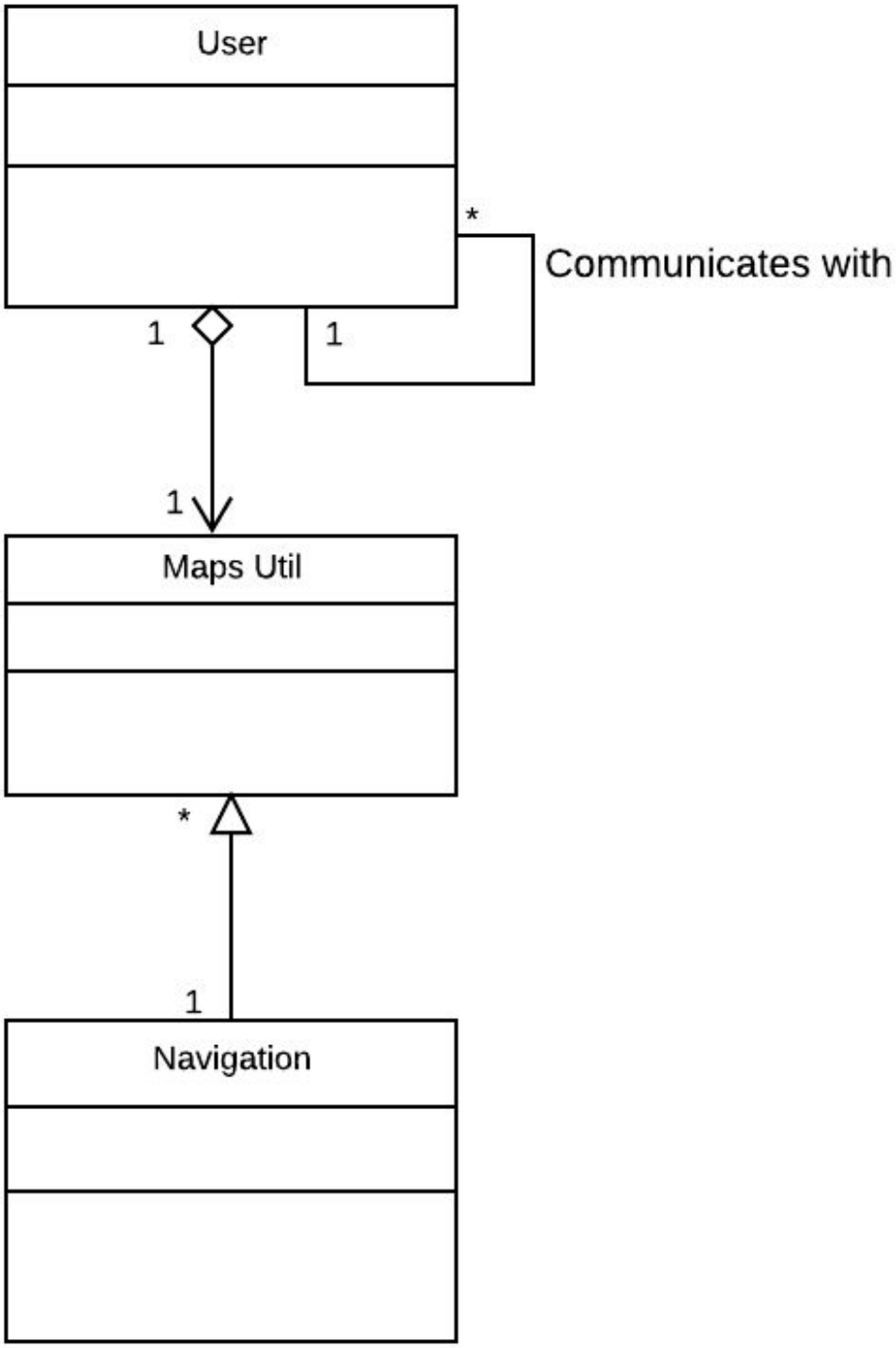
An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but maybe instantiated many different ways based on the behavior of the system.

We have derived the archetypes for the E-Shongee app by examining the analysis classes defined as part of the requirements model. The archetypes are defined as follows:

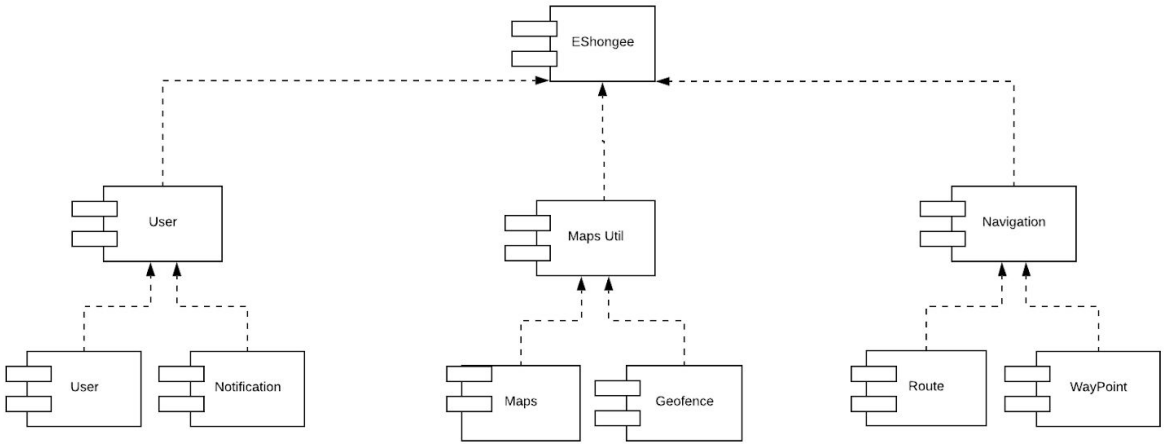
User: An abstraction that represents the parent and child generating routes, waypoints and creating geofence using the E-Shongee app.

MapsUtil: An abstraction that encompasses all mechanisms related to the map and geofence, that is, plotting points for creating a geofence.

Navigation: An abstraction that represents all mechanisms for indicating the route used, route generation, setting up waypoints, the movement of a child inside or outside the geofence.


User
*
Communicates with
1
1
1
Maps Util
*
1
Navigation

2.3 Refining Archetype into top-level Component



As the software architecture is refined into components, the structure of the system begins to emerge. We have chosen these components by beginning with the classes that were described as part of the requirements Model. These analysis classes represent entities within the application domain that must be addressed within the software architecture. The archetypes are User, MapsUtil and Navigation. The set of top-level components with their functions are given below:

User: This component is refined from the user archetype and involved with sending parental requests, checking requests, sending a route and sending geofence. Users (Parent and child) are involved with route and waypoint generation, geofence creation. The parent sends request to the child. The child accepts the request. Parents get notified when a child enters or leaves a geofence.

Notification: This component is refined from the user archetype. This component is involved with sending notification and initiating messaging service. Parents get notified when a child enters or leaves a geofence.

Maps: This component is refined from the Maps-Util archetype. This component is involved with loading maps, searching places, moving the camera, setting the route and setting geofence.

Geofence: This component is refined from the Maps-Util archetype and involved with manipulating and monitoring geofence.

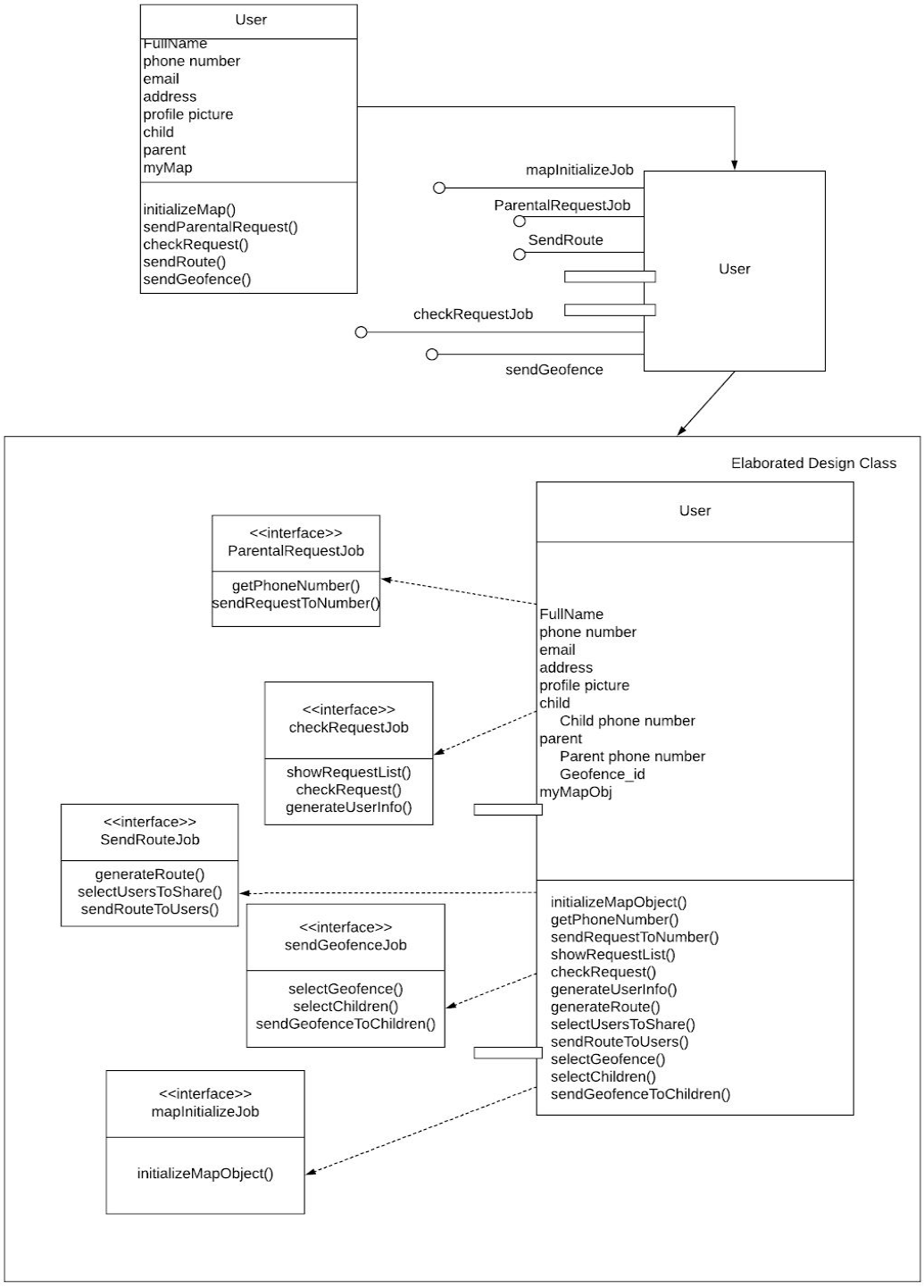
Route: This component is refined from the Navigation archetype and it is involved with loading waypoints and generating routes.

Waypoint: This component is refined from the Navigation archetype and it is involved with generating and plotting waypoints.

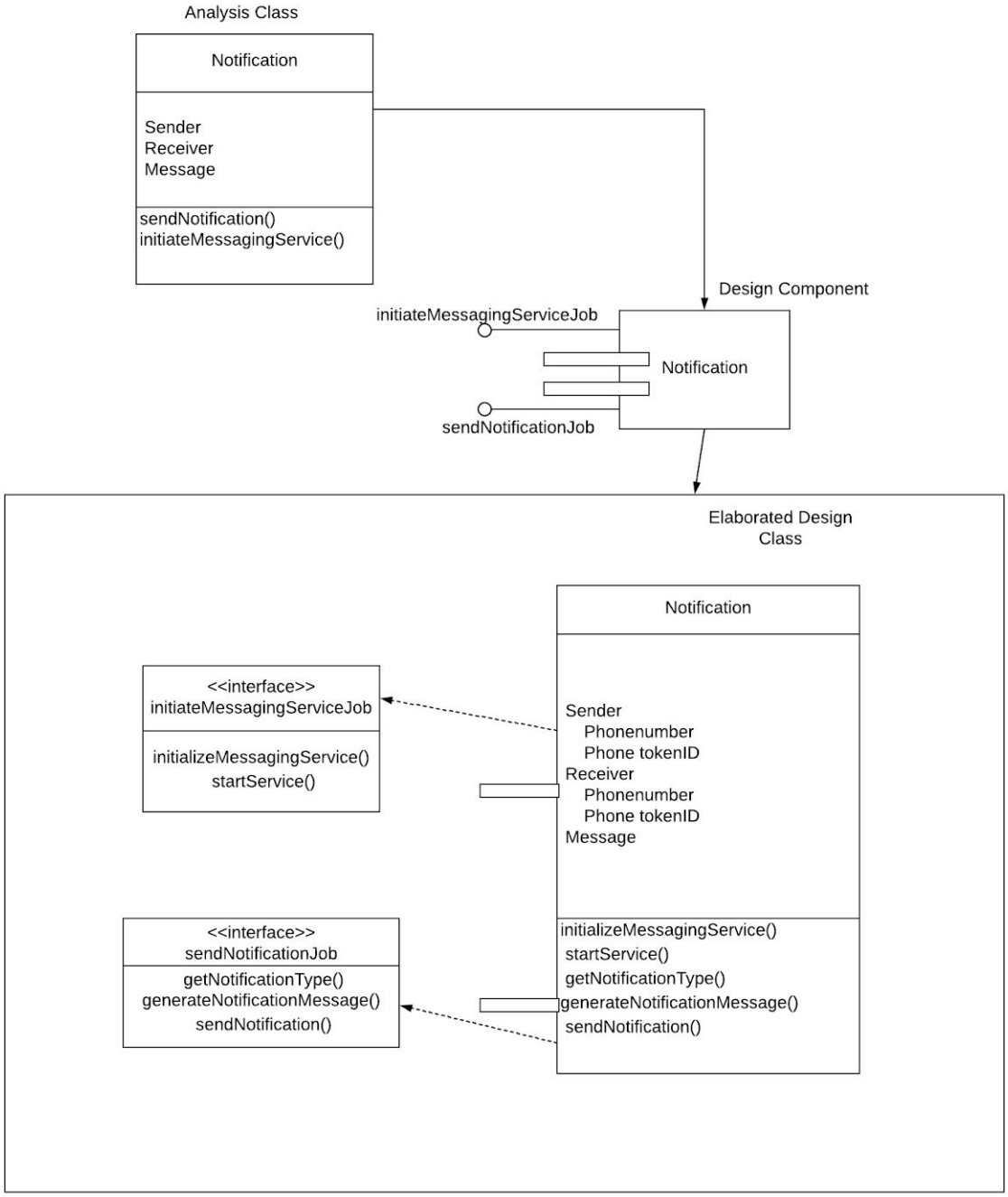
Chapter 3: Component-Level Design

3.1 Transforming Analysis Class into Design Class

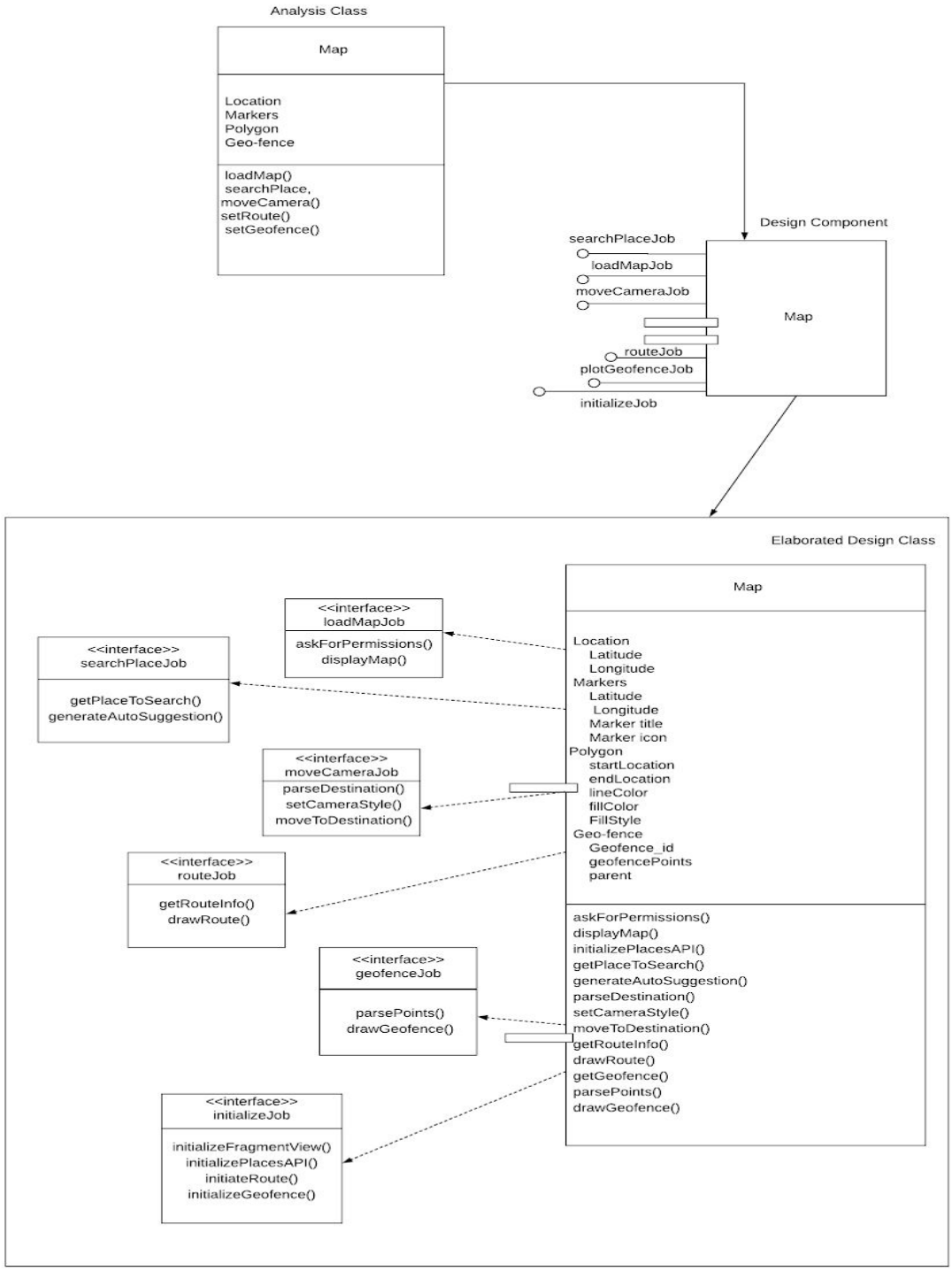
3.1.1 User


User
FullName
phone number
email
address
profile picture
child
parent
myMap
initializeMap()
sendParentalRequest()
checkRequest()
sendRoute()
sendGeofence()
mapInitializeJob
ParentalRequestJob
SendRoute
User
checkRequestJob
sendGeofence
Elaborated Design Class
User
<<interface>>
ParentalRequestJob
getPhoneNumber()
sendRequestToNumber()
FullName
phone number
email
address
profile picture
child
Child phone number
parent
Parent phone number
Geofence_id
myMapObj
<<interface>>
checkRequestJob
showRequestList()
checkRequest()
generateUserInfo()
<<interface>>
SendRouteJob
generateRoute()
selectUsersToShare()
sendRouteToUsers()
initializeMapObject()
getPhoneNumber()
sendRequestToNumber()
showRequestList()
checkRequest()
generateUserInfo()
generateRoute()
selectUsersToShare()
sendRouteToUsers()
selectGeofence()
selectChildren()
sendGeofenceToChildren()
<<interface>>
sendGeofenceJob
selectGeofence()
selectChildren()
sendGeofenceToChildren()
<<interface>>
mapInitializeJob
initializeMapObject()

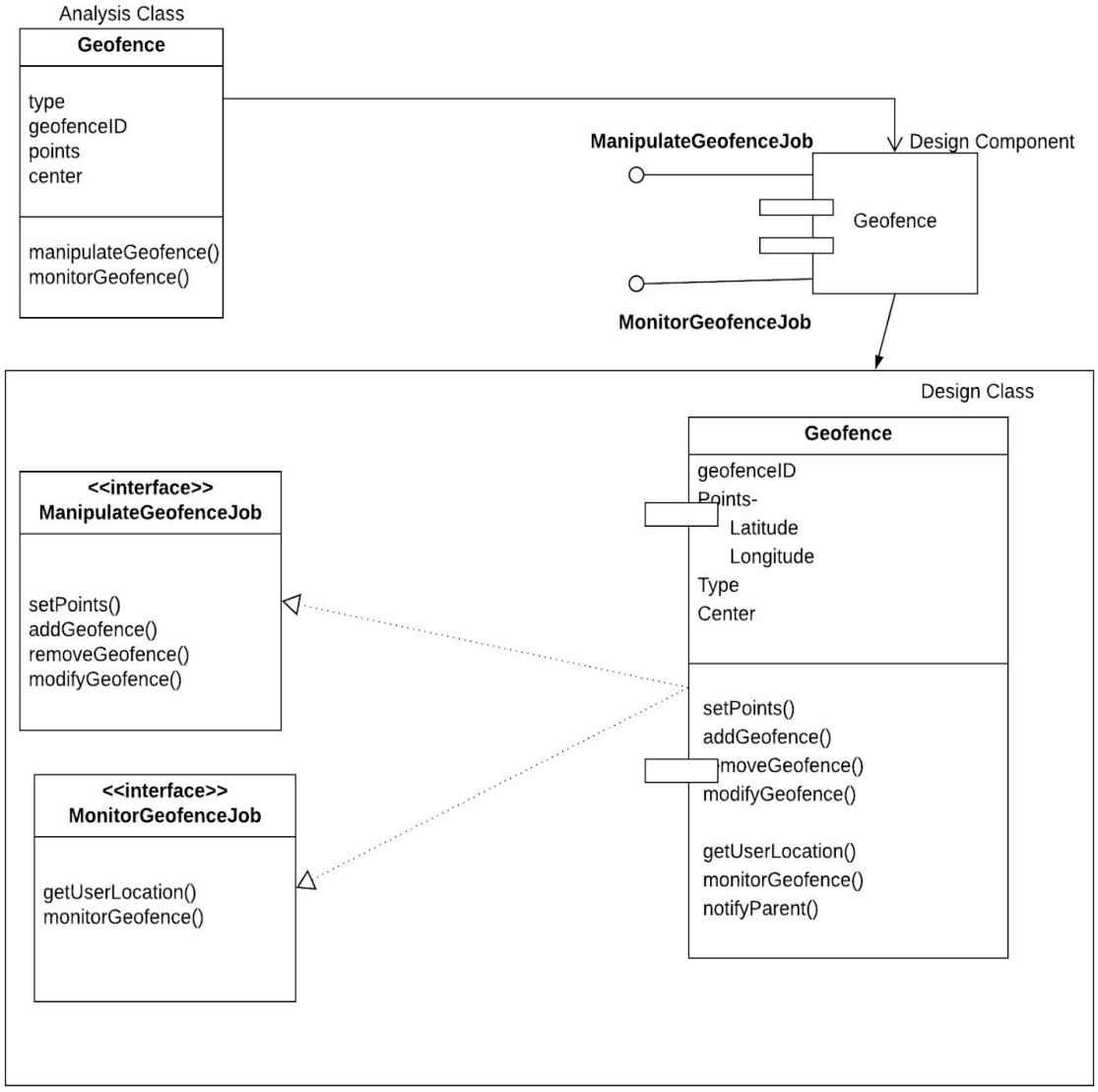
3.1.2 Notification



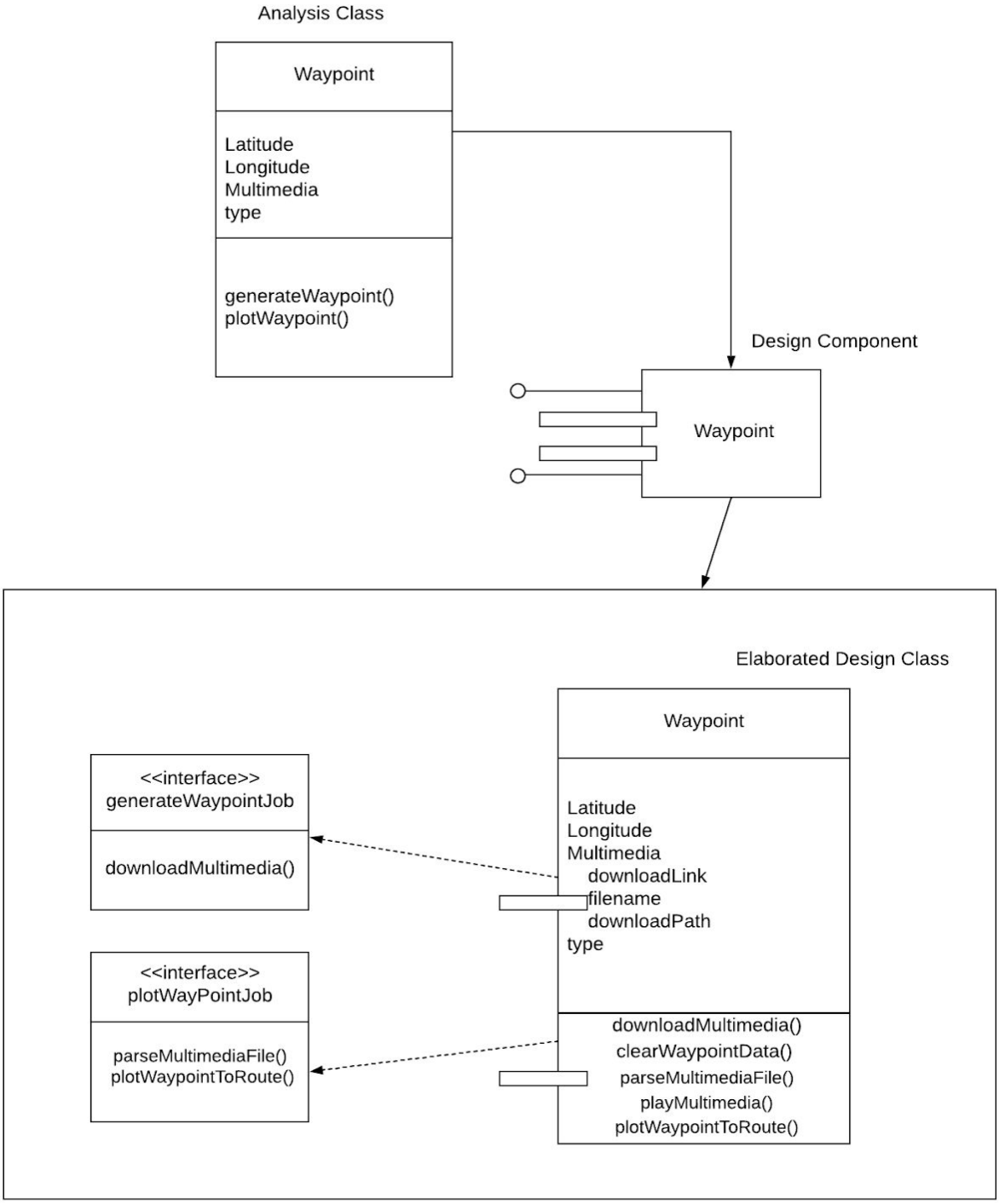
3.1.3 Map


Analysis Class
Map
Location
Markers
Polygon
Geo-fence
loadMap()
searchPlace,
moveCamera()
setRoute()
setGeofence()
Design Component
searchPlaceJob
loadMapJob
moveCameraJob
Map
routeJob
plotGeofenceJob
initializeJob
Elaborated Design Class
Map
<<interface>>
loadMapJob
askForPermissions()
displayMap()
<<interface>>
searchPlaceJob
getPlaceToSearch()
generateAutoSuggestion()
<<interface>>
moveCameraJob
parseDestination()
setCameraStyle()
moveToDestination()
<<interface>>
routeJob
getRouteInfo()
drawRoute()
<<interface>>
geofenceJob
parsePoints()
drawGeofence()
<<interface>>
initializeJob
initializeFragmentView()
initializePlacesAPI()
initiateRoute()
initializeGeofence()
Location
Latitude
Longitude
Markers
Latitude
Longitude
Marker title
Marker icon
Polygon
startLocation
endLocation
lineColor
fillColor
FillStyle
Geo-fence
Geofence_id
geofencePoints
parent
askForPermissions()
displayMap()
initializePlacesAPI()
getPlaceToSearch()
generateAutoSuggestion()
parseDestination()
setCameraStyle()
moveToDestination()
getRouteInfo()
drawRoute()
getGeofence()
parsePoints()
drawGeofence()

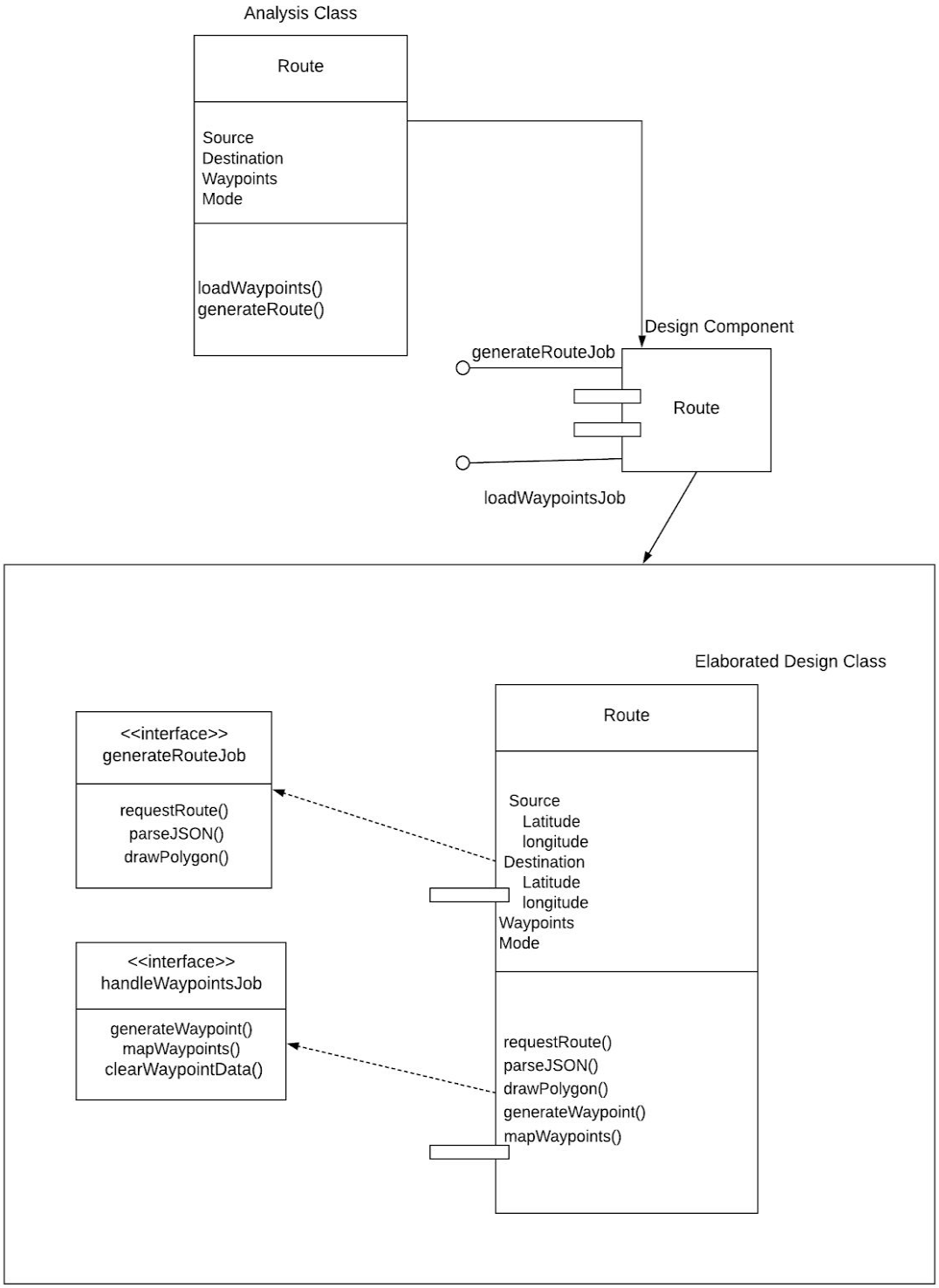
3.1.4 Geofence



3.1.5 Waypoint



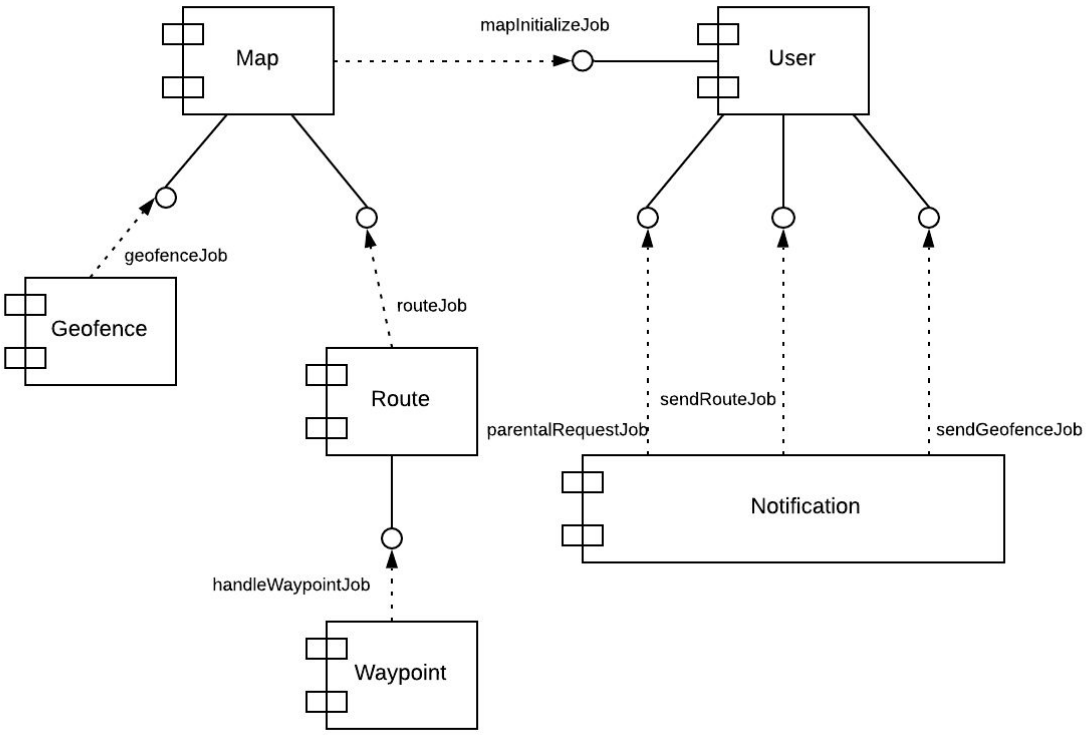
3.1.6 Route


Analysis Class
Route
Source
Destination
Waypoints
Mode
loadWaypoints()
generateRoute()
Design Component
generateRouteJob
Route
loadWaypointsJob
Elaborated Design Class
<<interface>>
generateRouteJob
requestRoute()
parseJSON()
drawPolygon()
<<interface>>
handleWaypointsJob
generateWaypoint()
mapWaypoints()
clearWaypointData()
Route
Source
Latitude
longitude
Destination
Latitude
longitude
Waypoints
Mode
requestRoute()
parseJSON()
drawPolygon()
generateWaypoint()
mapWaypoints()

3.2 Message Passing:

We used a collaboration diagram to show how analysis classes collaborate with one another. As our component-level design proceeded, it was useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. This design activity is used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

The figure below illustrates a simple collaboration diagram for the E-Shongee app.

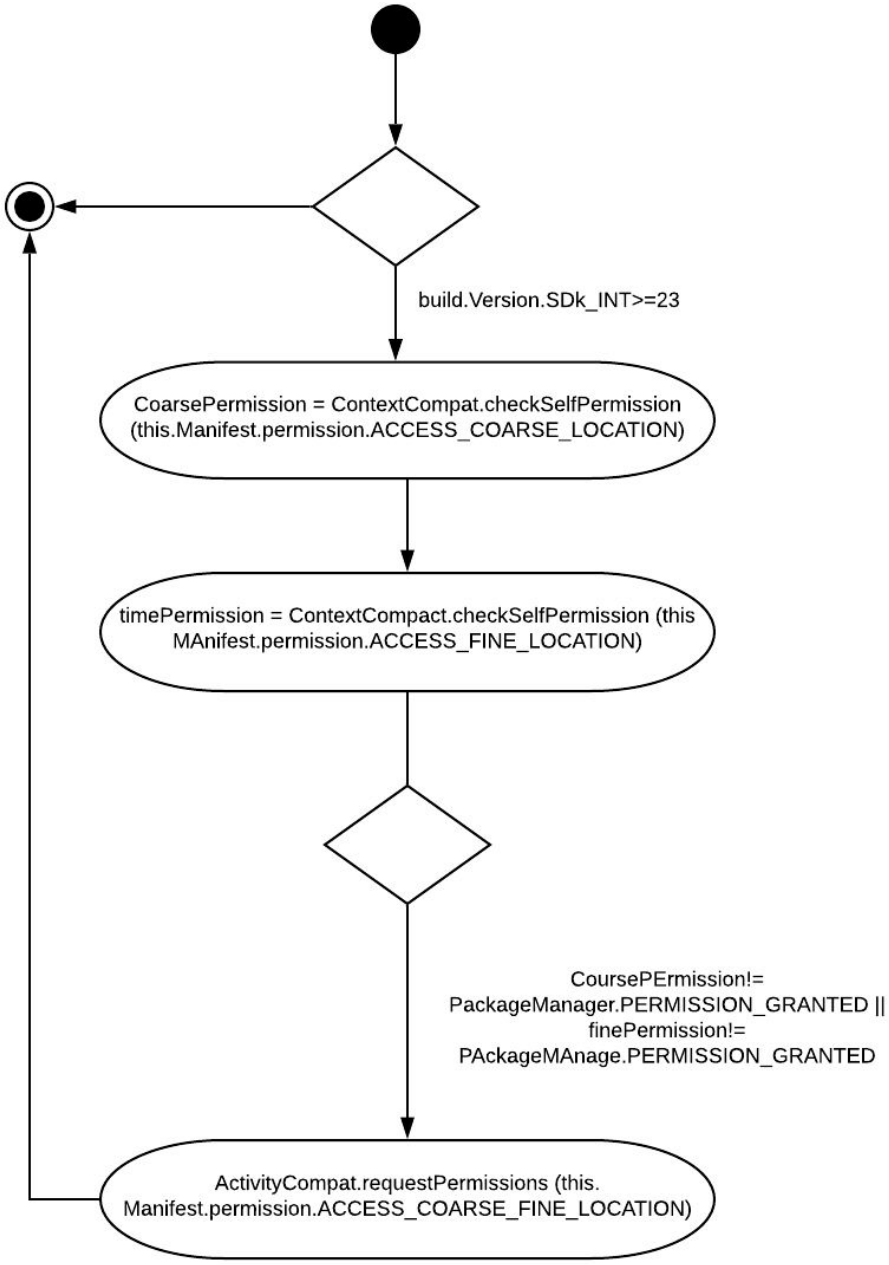


6 objects: Map, User, Notification, Geofence, Route, and Waypoint collaborate to run the functionalities of the E-Shongee app.
Messages are passed between objects as illustrated by the arrows in the figure. During requirements modeling the messages are specified as shown in the figure. However, as design proceeds, each message is elaborated by expanding its syntax in the following manner:
[guard condition] sequence expression (return value) :=
message name (argument list)

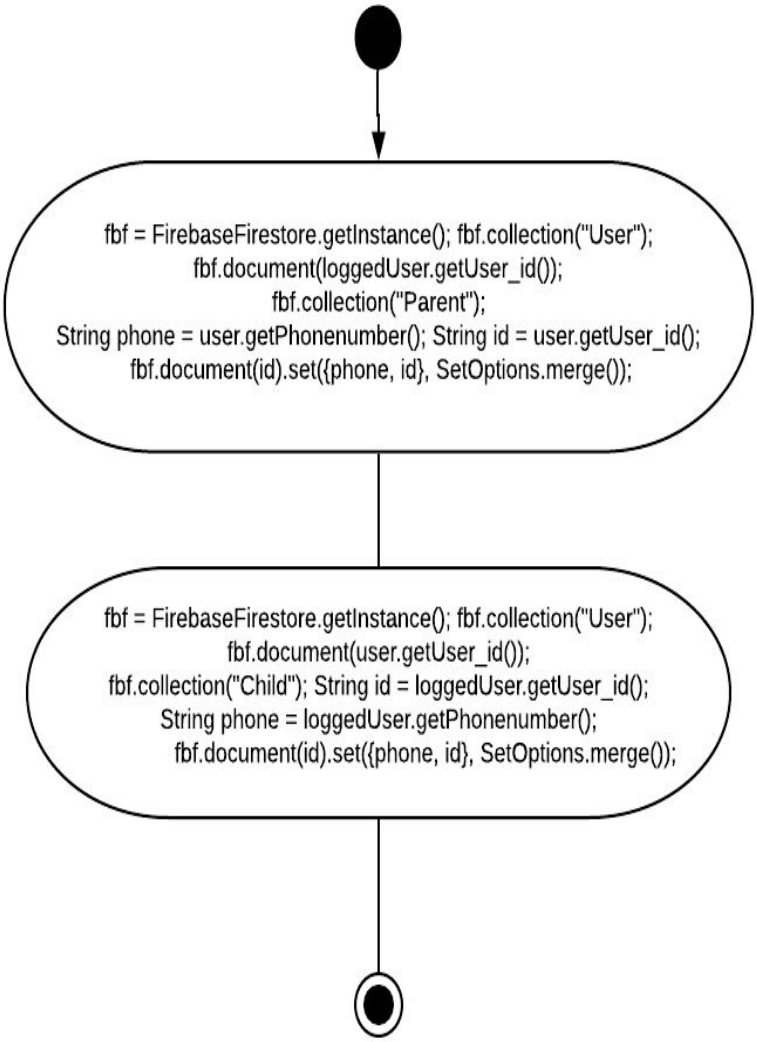
3.3 UML Activity Diagrams:

3.3.1 MAP CLASS

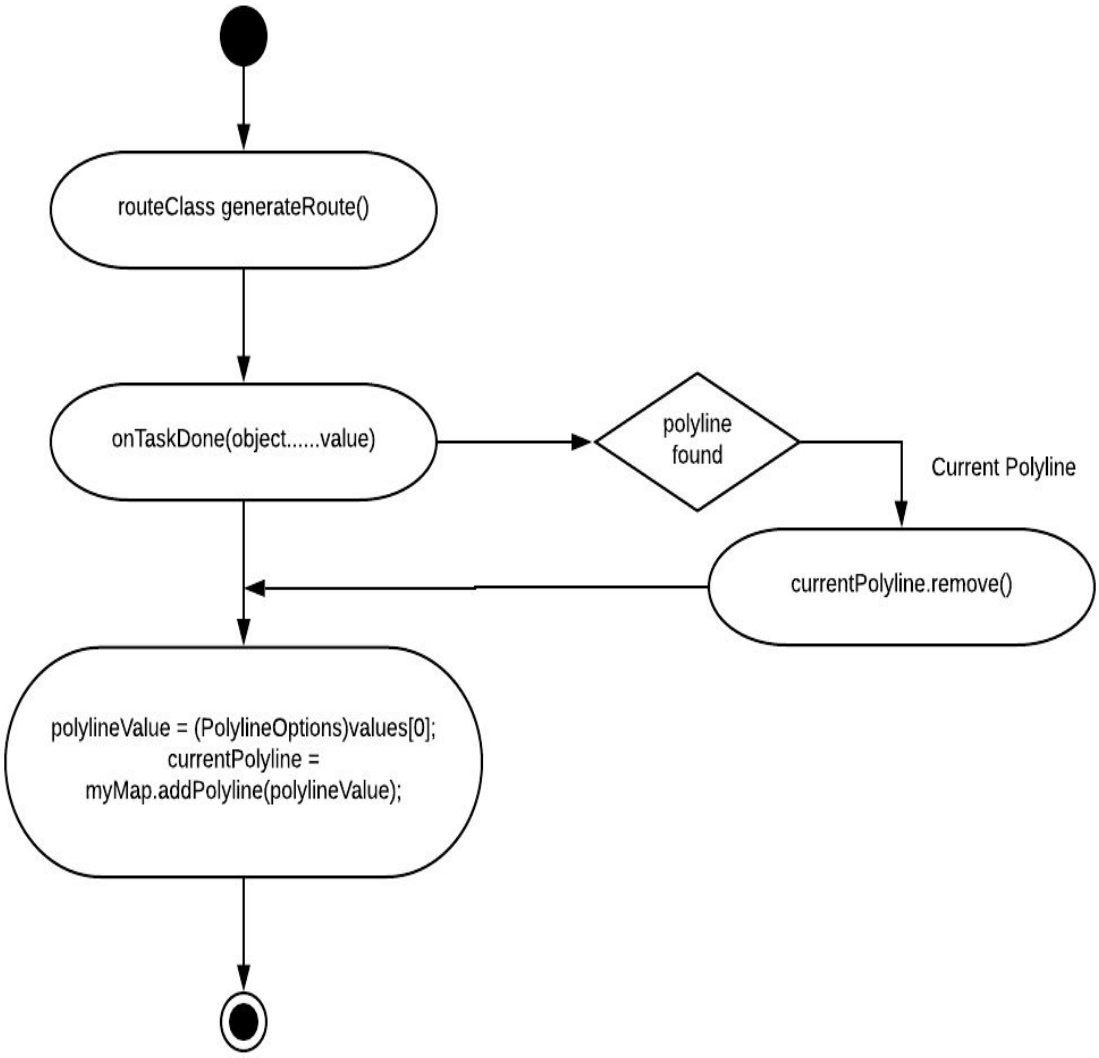
UML Activity Diagram for askForPermission() :=



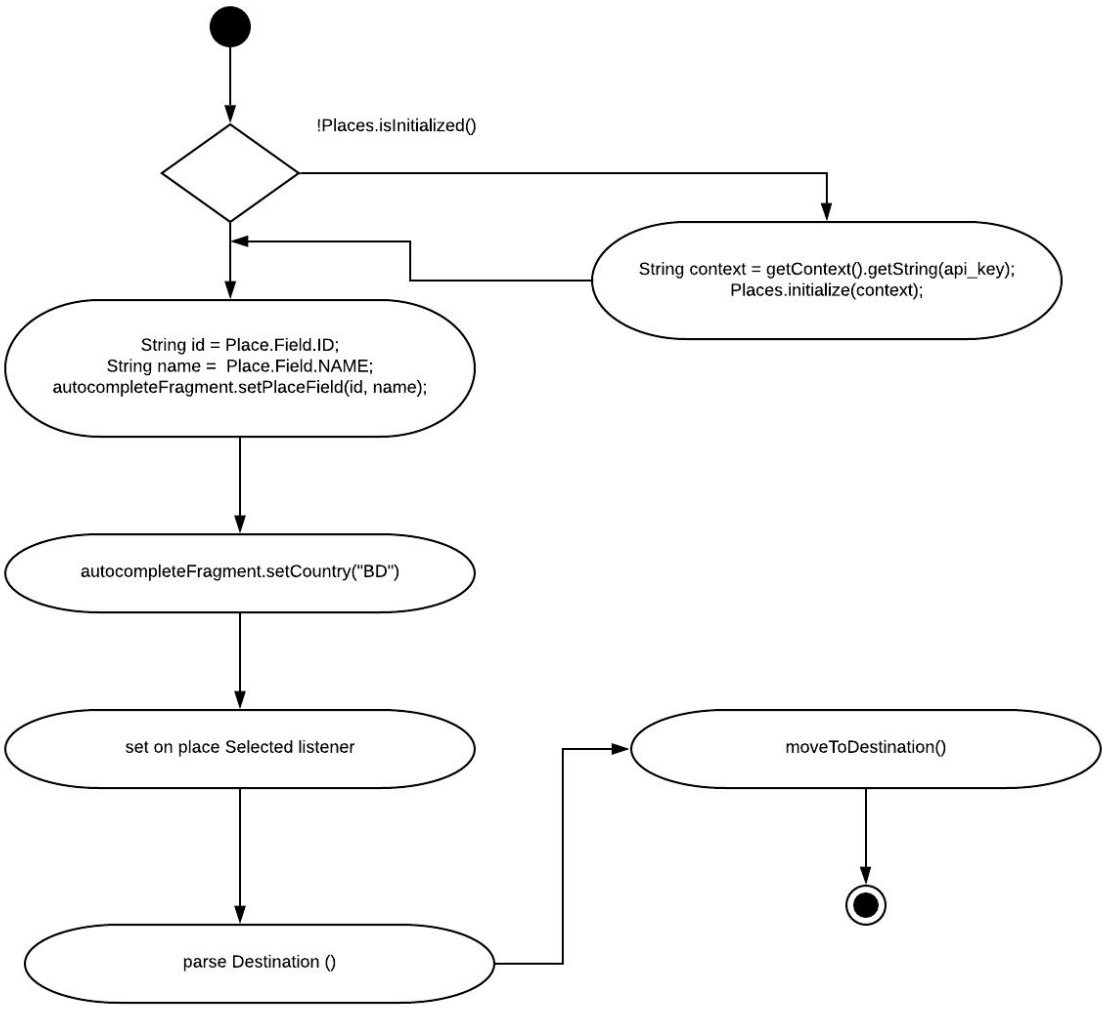
UML Activity Diagram for sendGeofenceToChild(child) :=



UML Activity Diagram for drawRoute(source, destination) :=

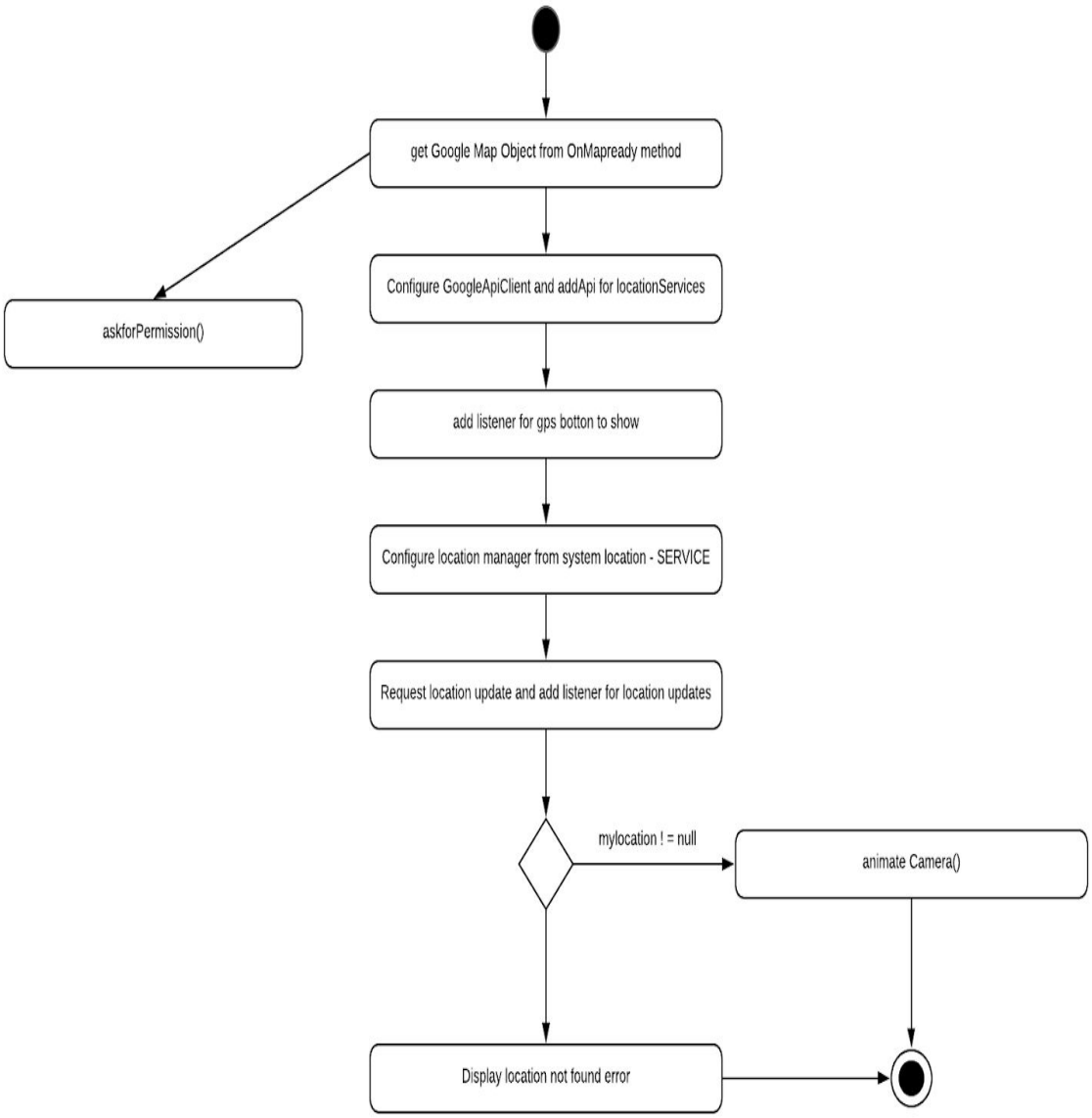


UML Activity Diagram for initializePlacesAPI() :=

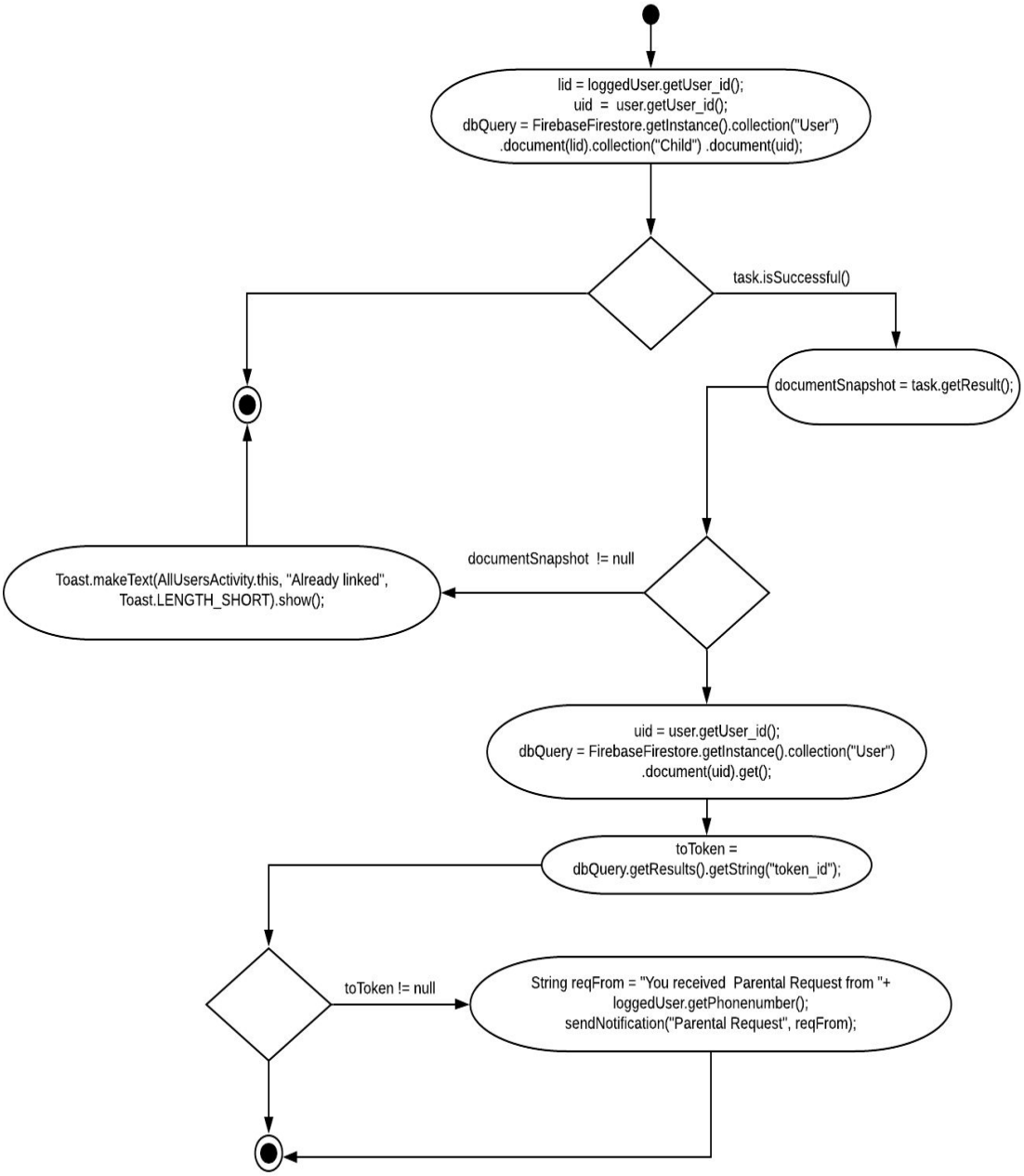


3.3.2 USER CLASS:

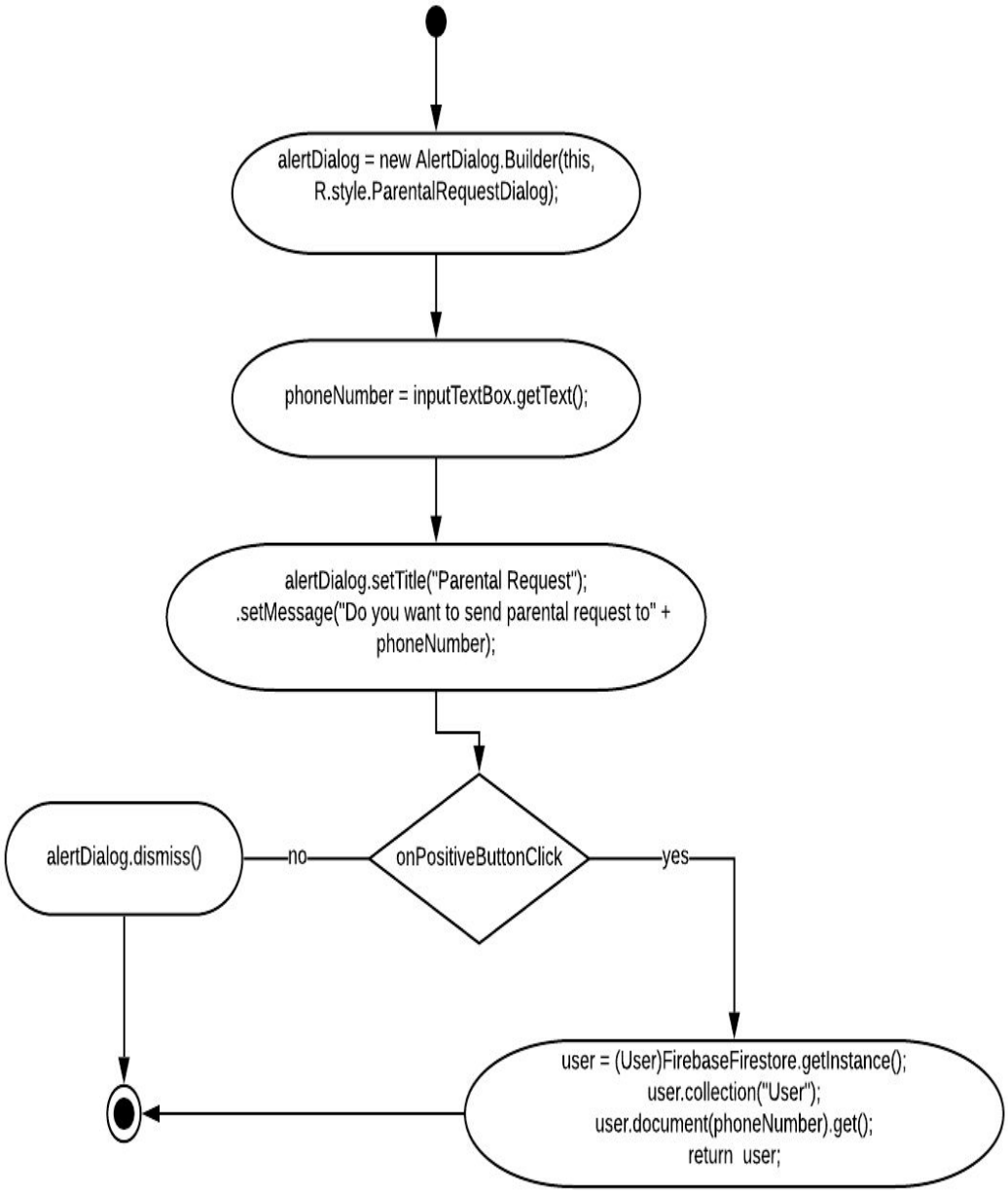
UML Activity Diagram for displayMap(mapObj) :=



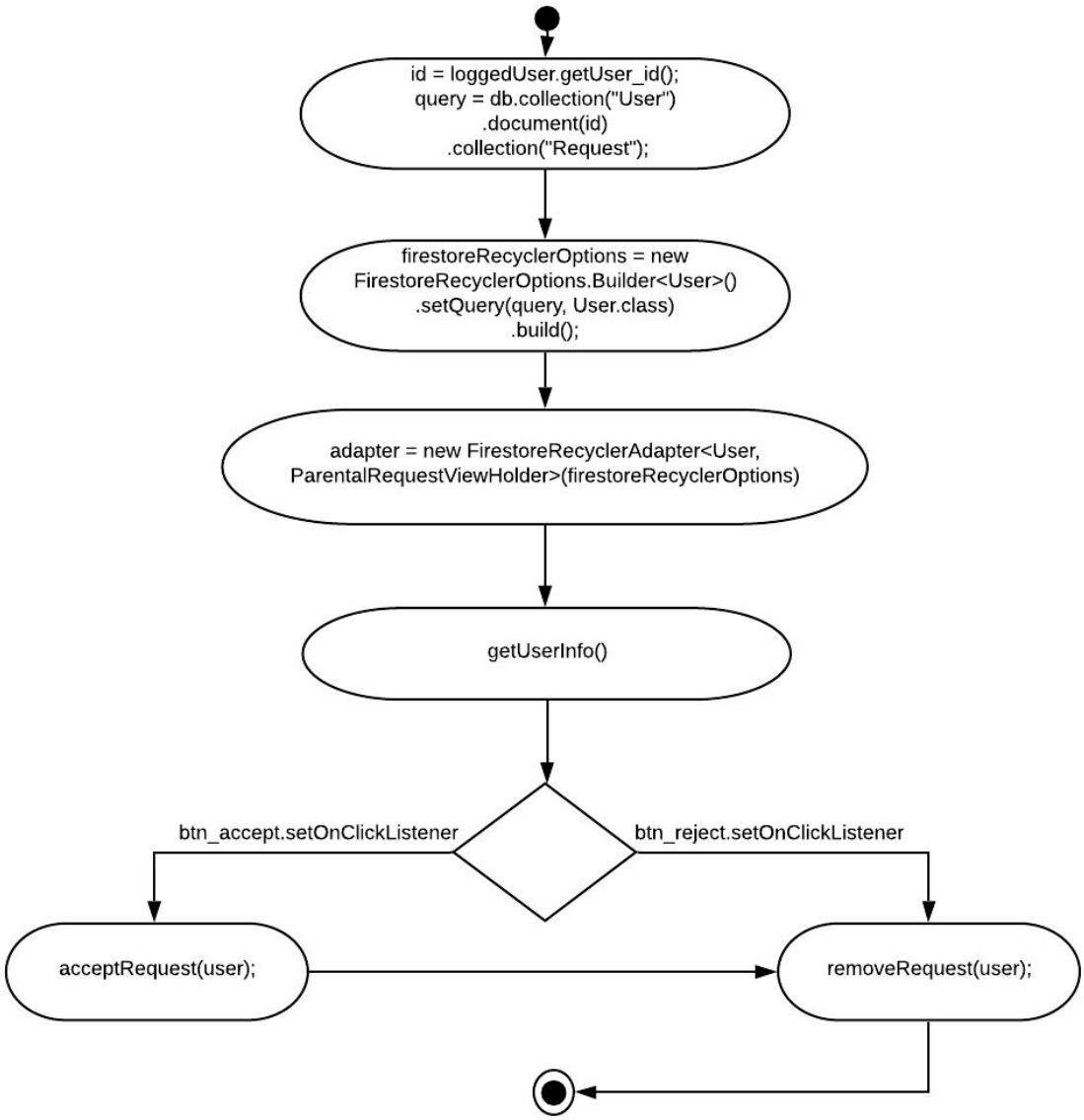
UML Activity Diagram for sendParentalRequest(parent) :=



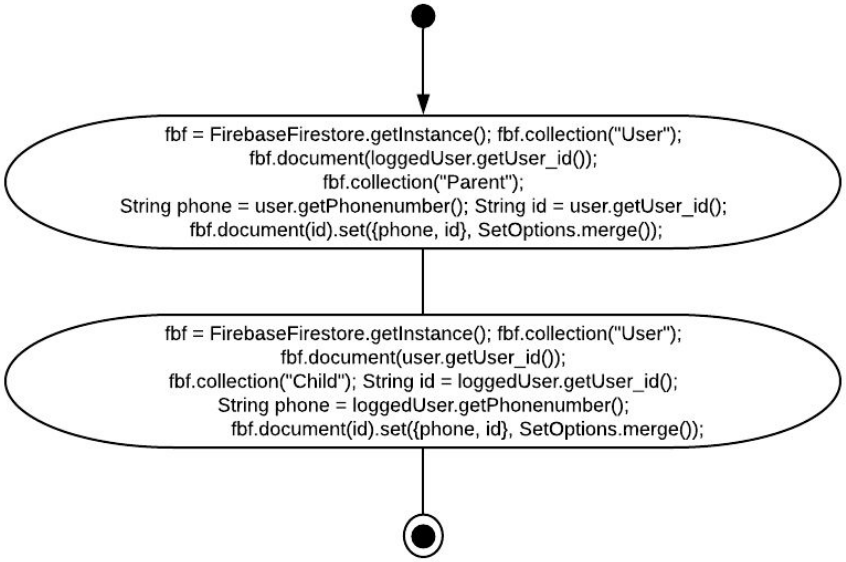
UML Activity Diagram for getUserFromNumber(phoneNumber) :=



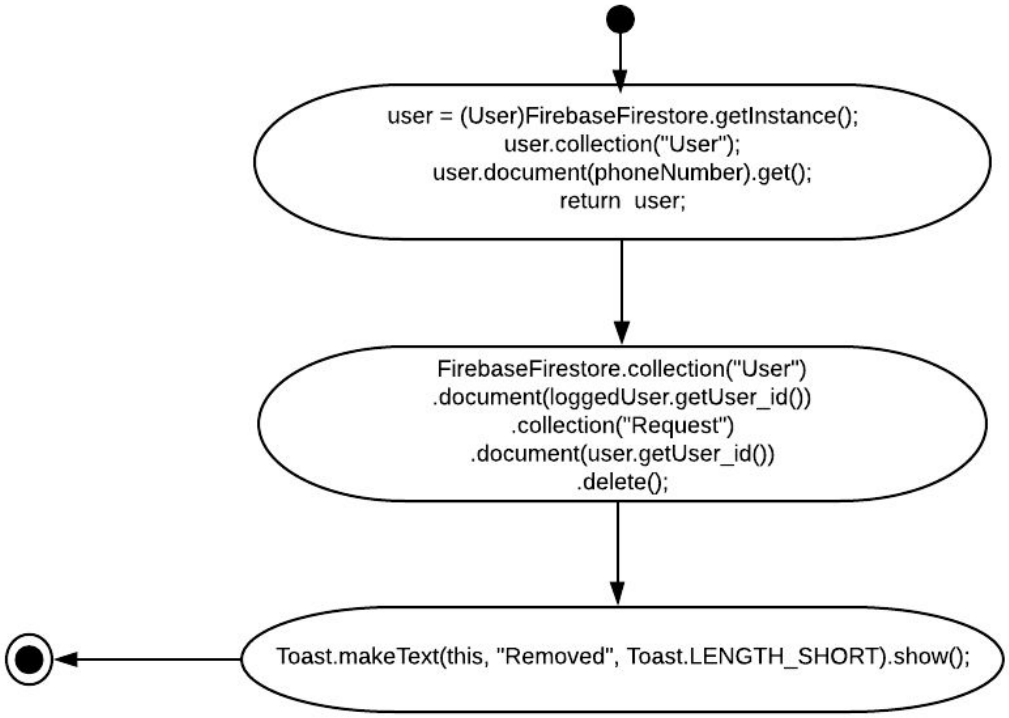
UML Activity Diagram for showRequestList() :=



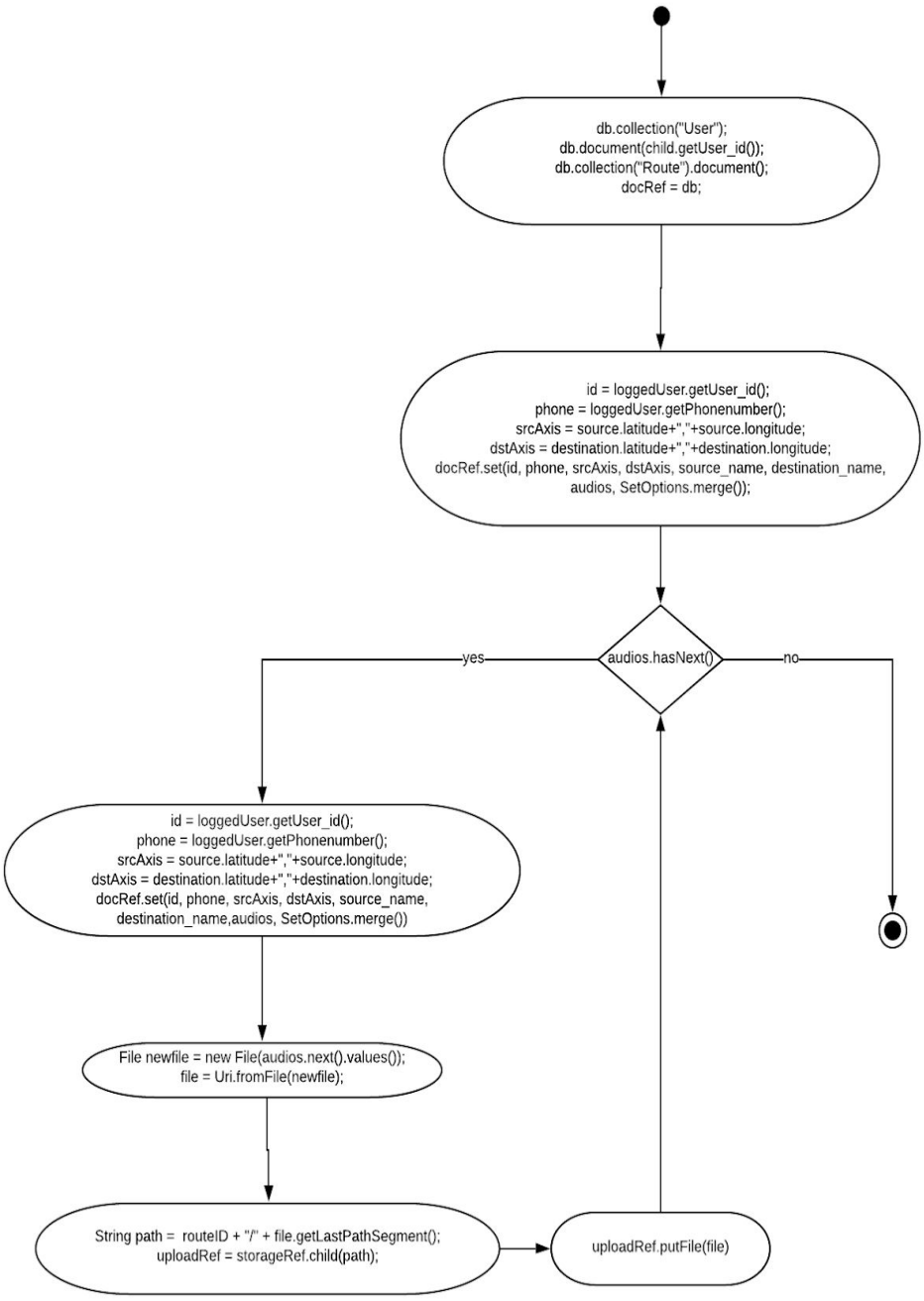
UML Activity Diagram for acceptRequest(userID) :=



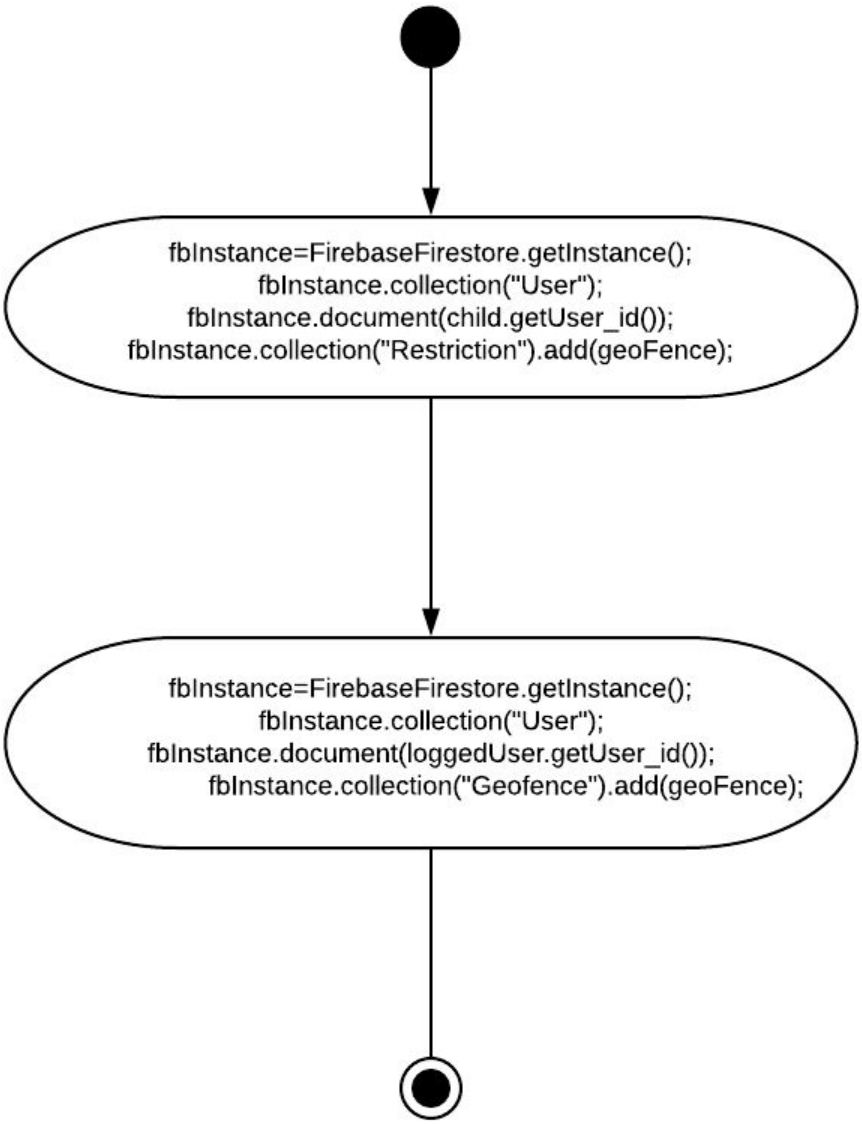
UML Activity Diagram for removeRequest(userID) :=



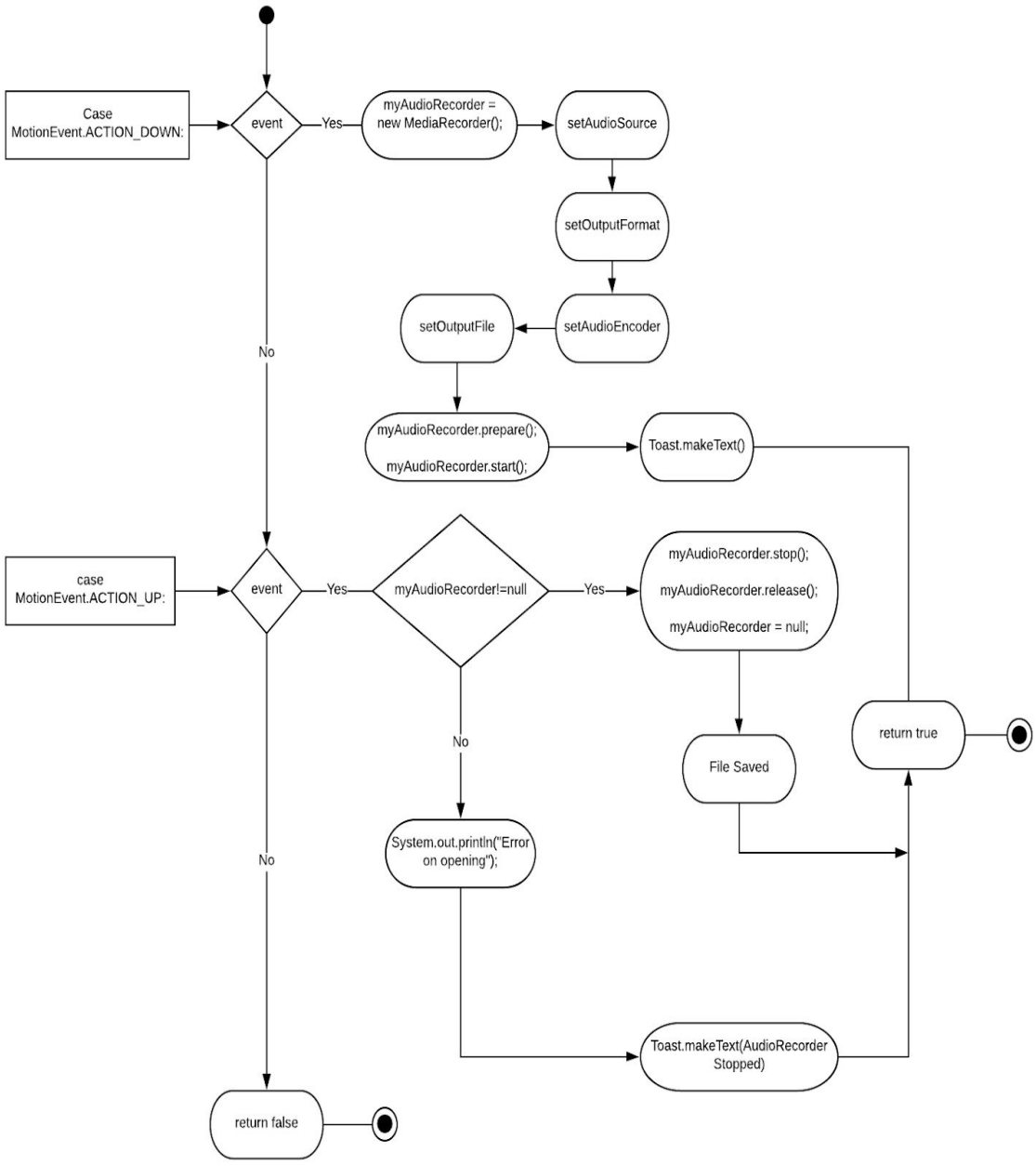
UML Activity Diagram for sendRoute(user) :=



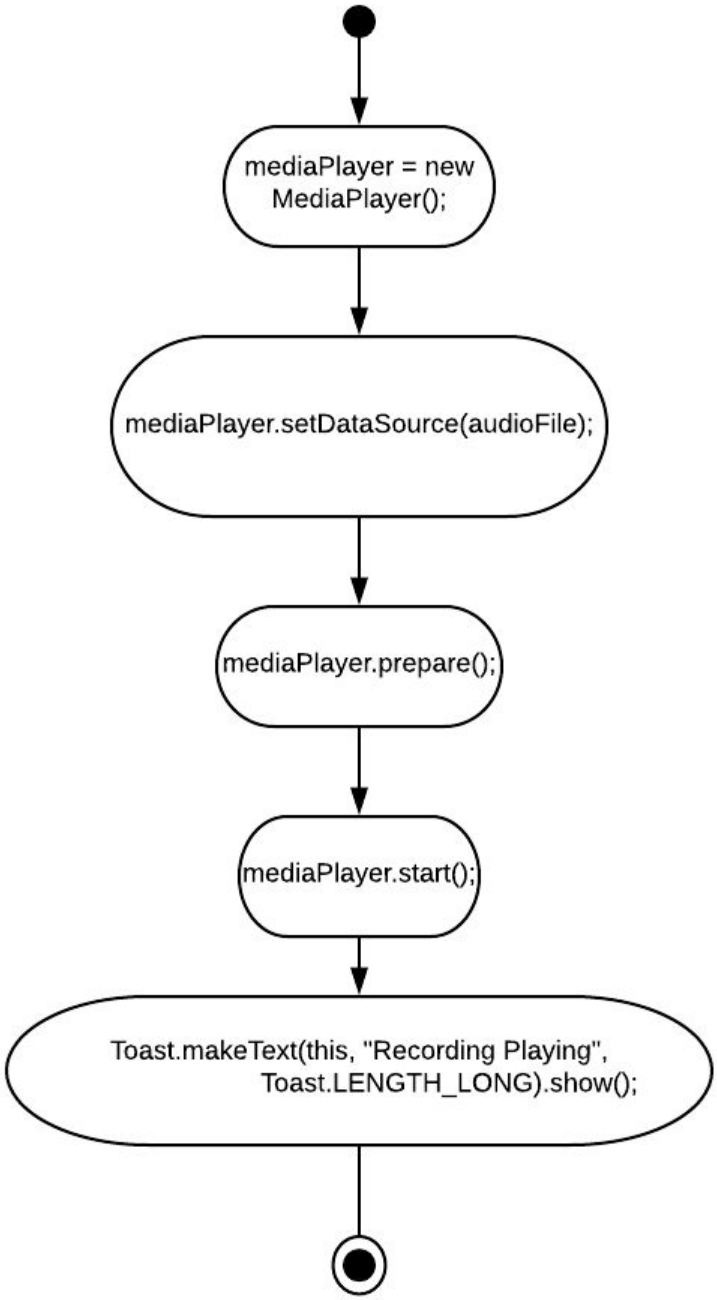
UML Activity Diagram for addGeogenceToChild(child) :=



3.3.3 WaypointCLASS:
UML Activity Diagram for recordAudio(filePath) :=

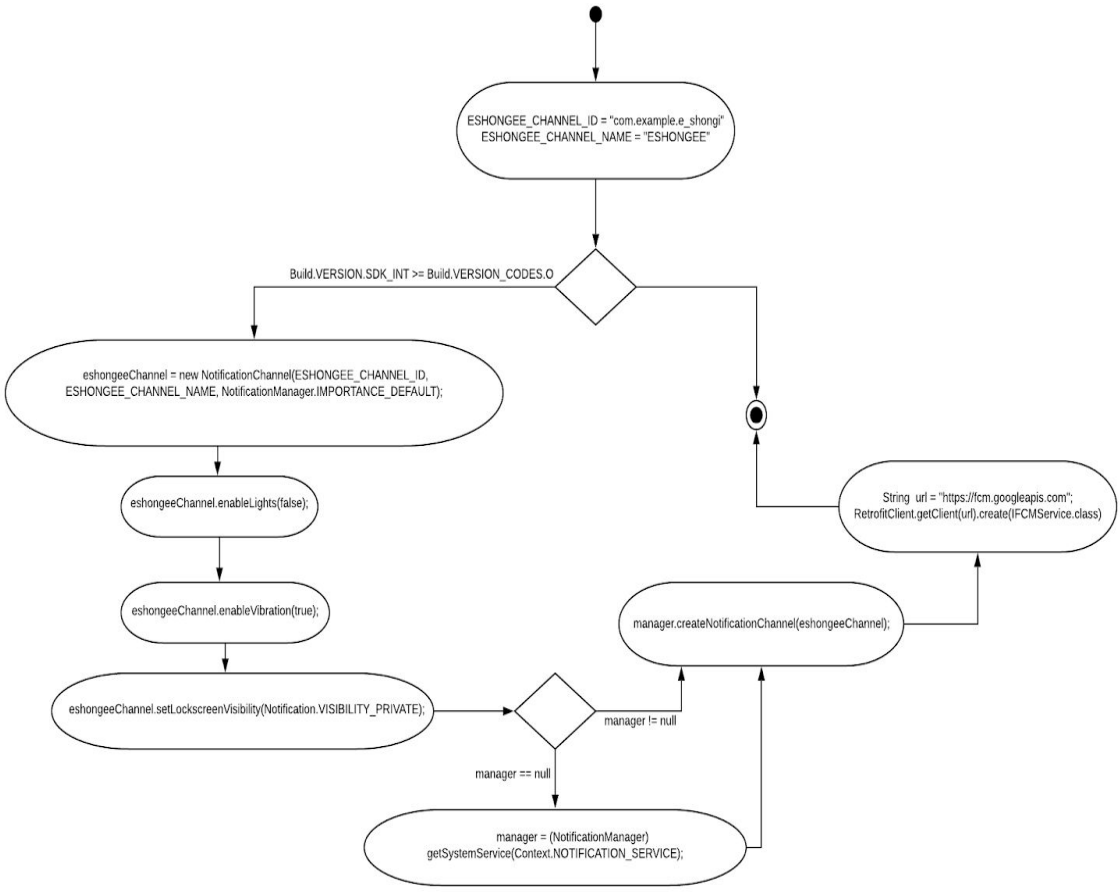


UML Activity Diagram for playAudio(audioFile) :=

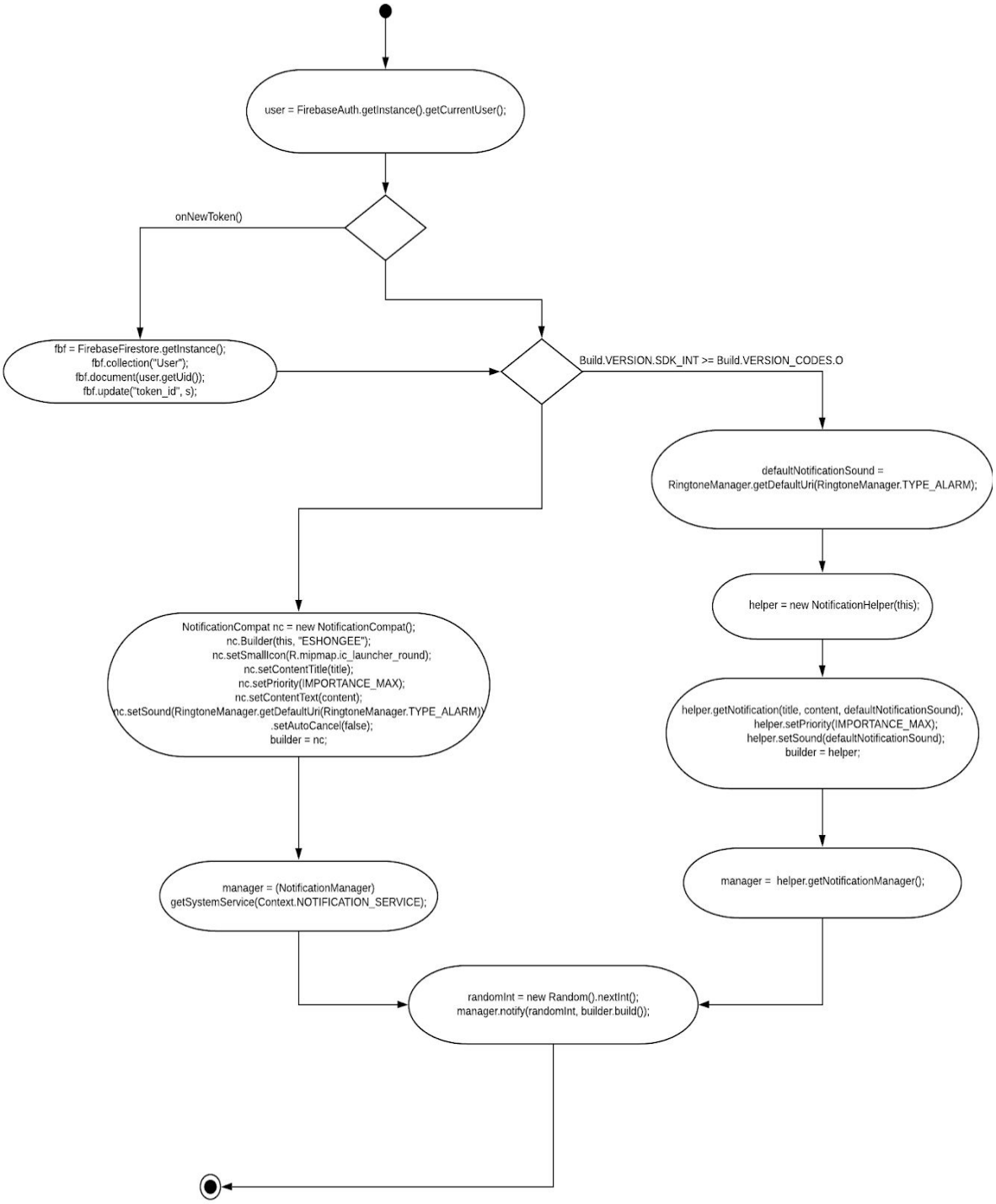


3.3.4NOTIFICATION CLASS:=

UML Activity Diagram for initializeMessagingService() :=

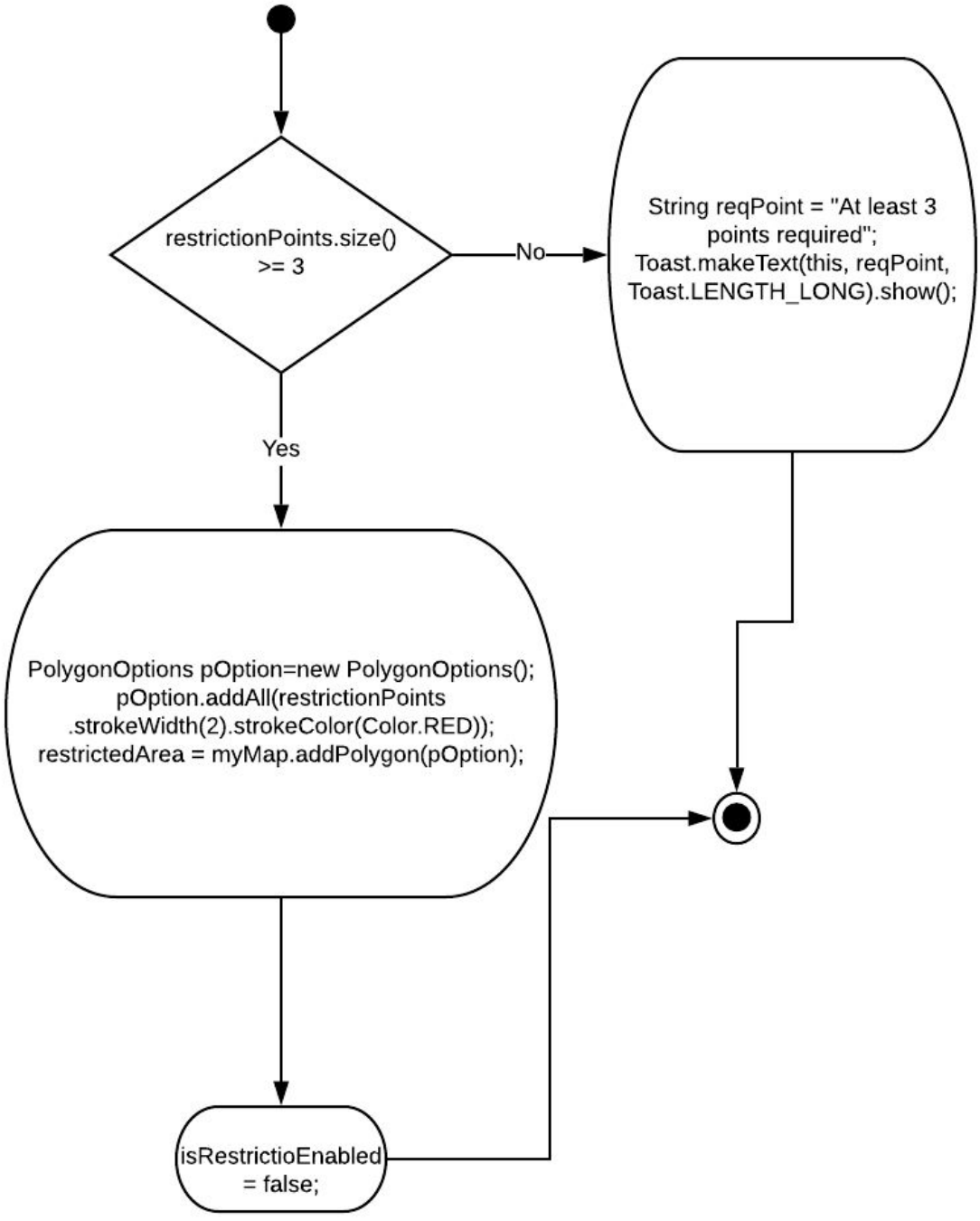


UML Activity Diagram for sendNotification(title, content) :=

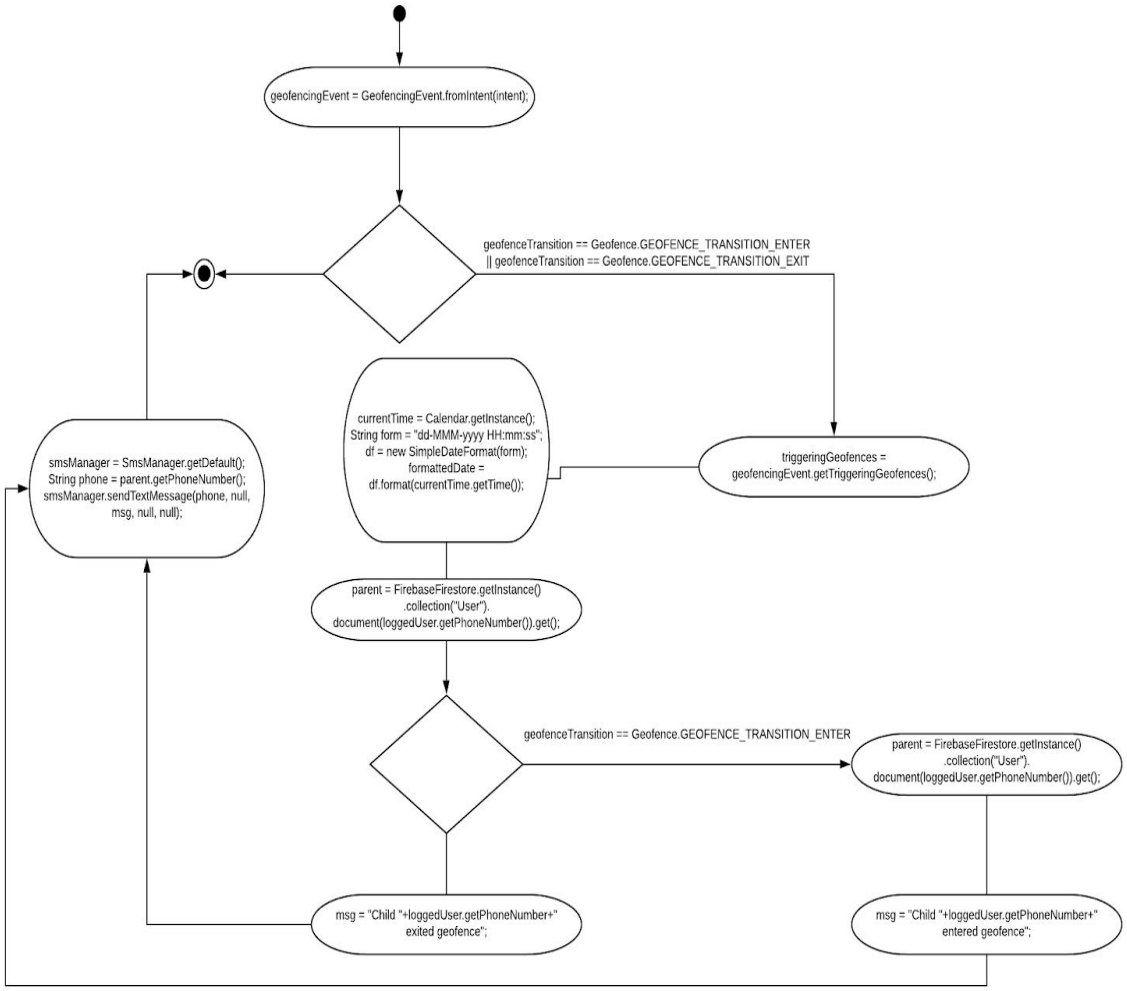


Geofence CLASS:=

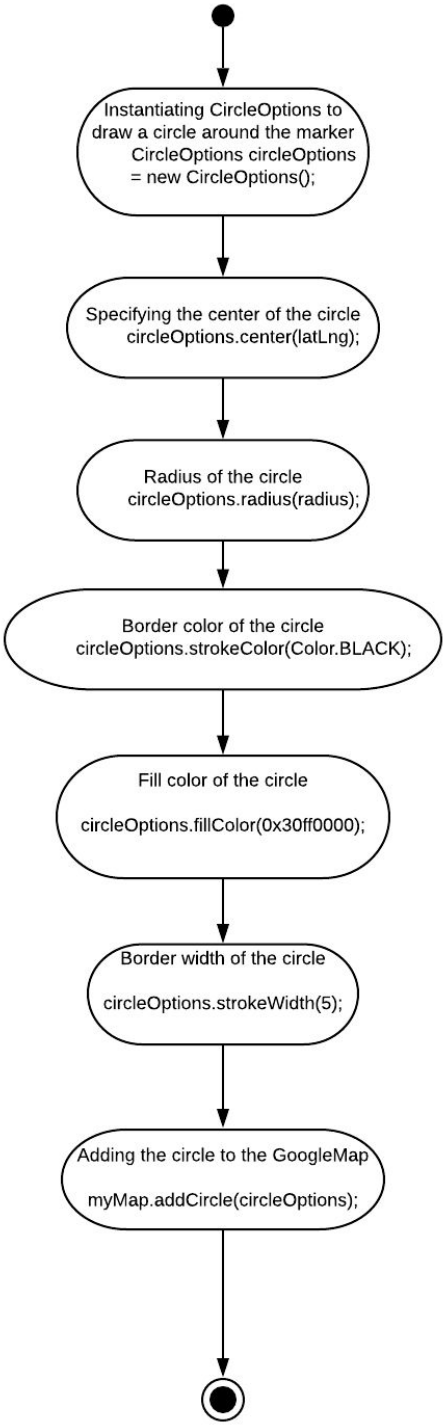
UML Activity Diagram for drawPolyline(restrictionPoints) :=



UML Activity Diagram for monitorGeofence(loggedUser) :=

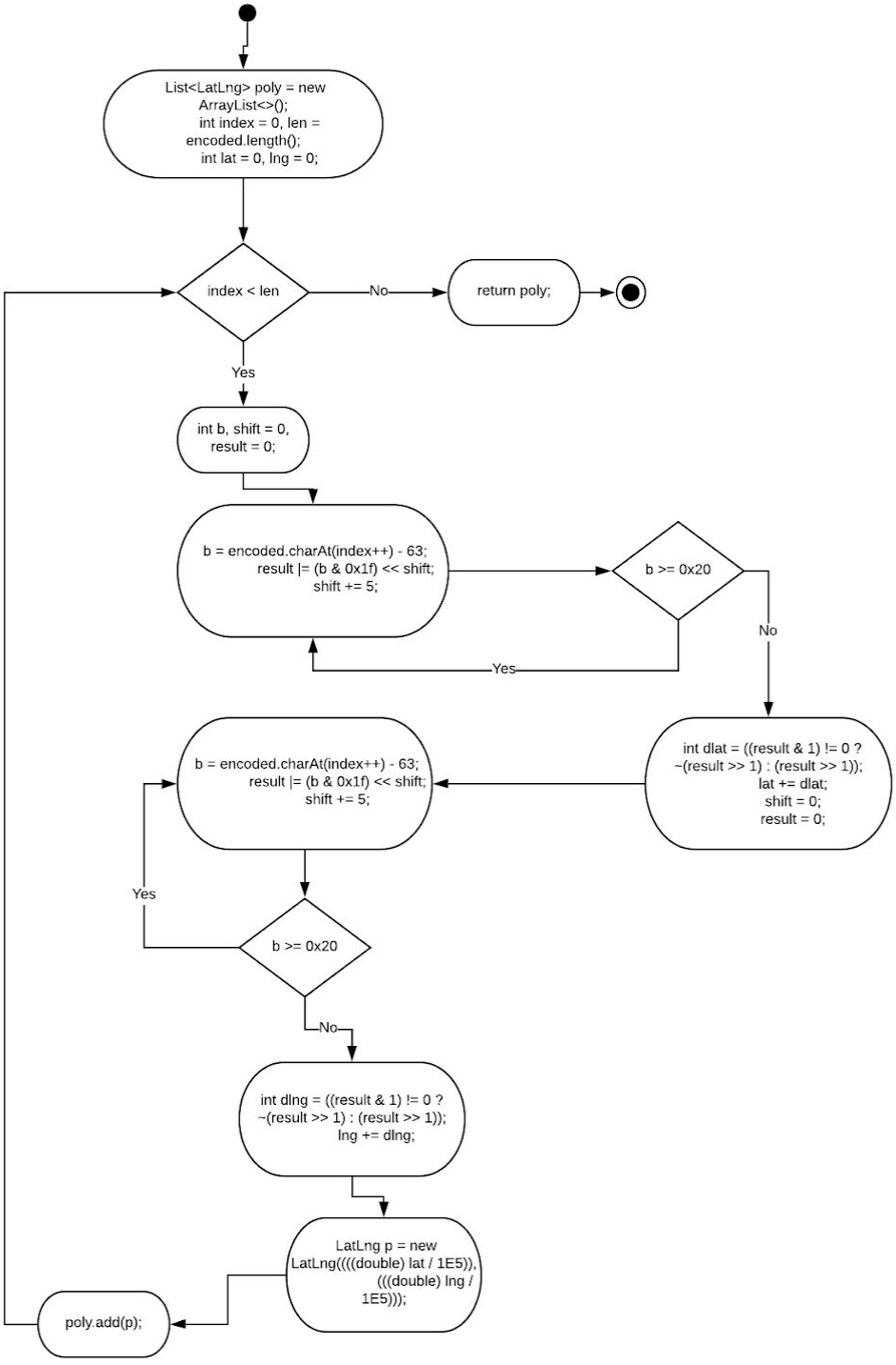


UML Activity Diagram for drawCircle(centerLatLng, radius) :=

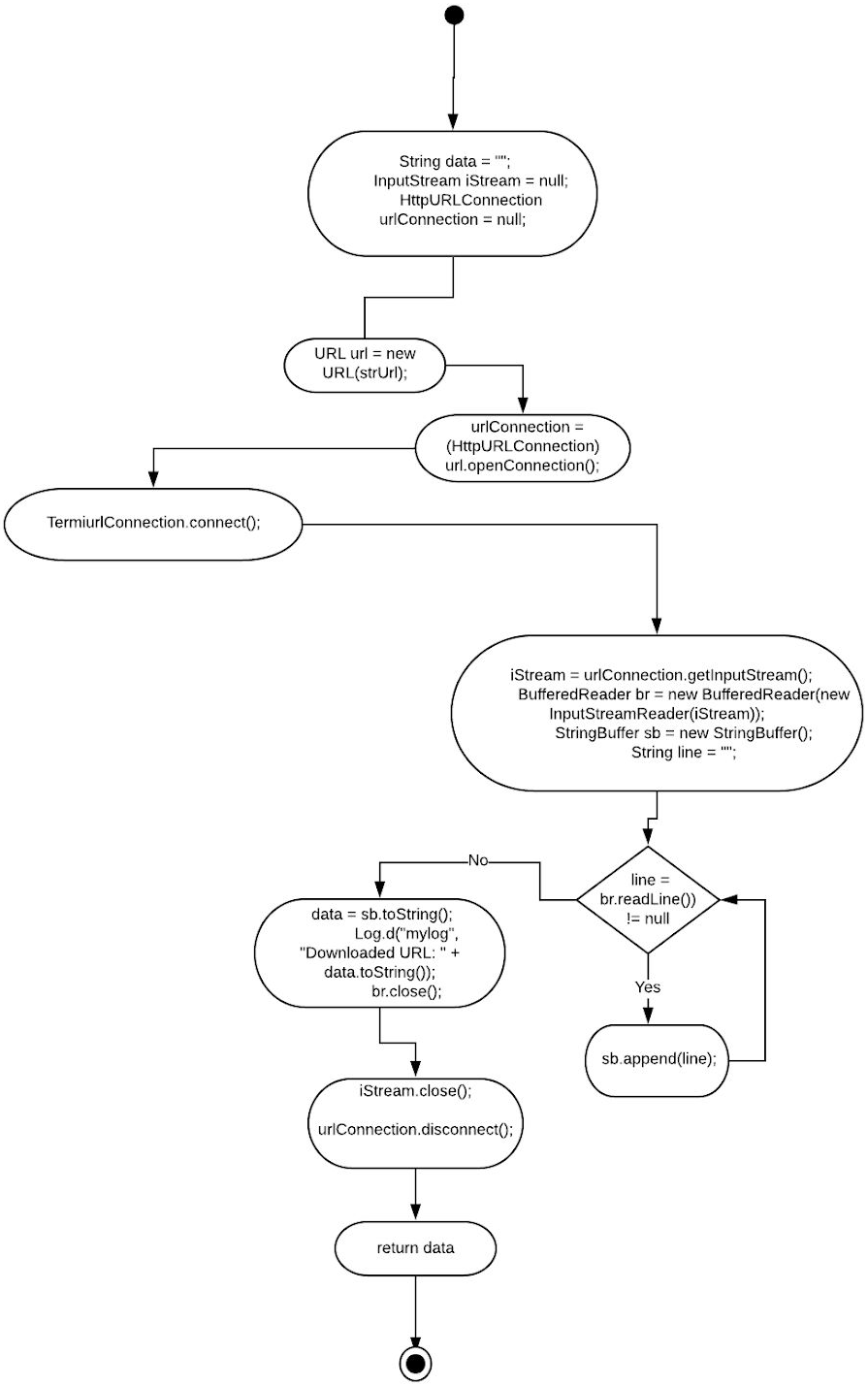


3.3.5 Route CLASS:=

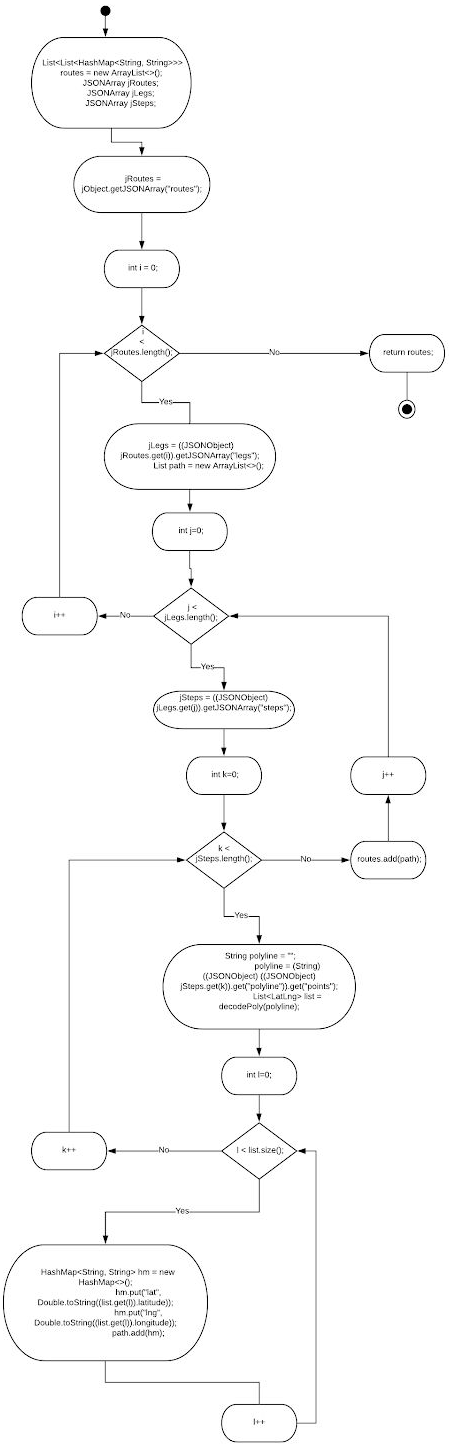
UML Activity Diagram for decodePoly(endcodedPoly) :=



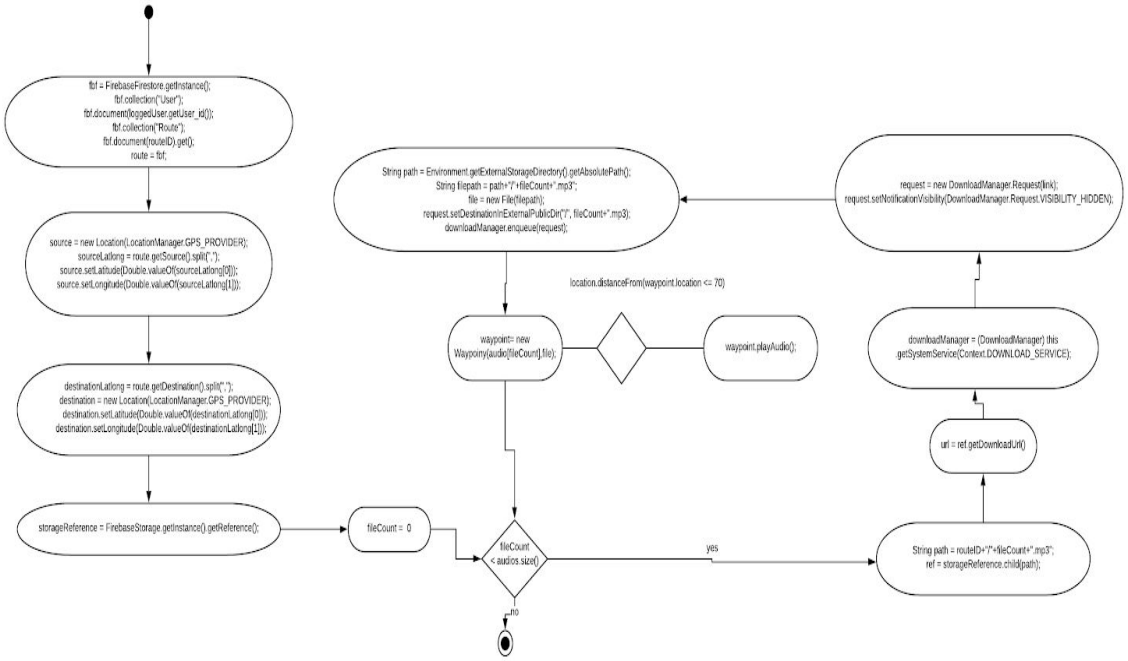
UML Activity Diagram for downloadUrl(urlStr) :=



UML Activity Diagram for parse(jsonData) :=



UML Activity Diagram for getSharedRoute(routeId) :=



https://www.lucidchart.com/documents/edit/5caeef0c-26df-4794-a848-97b463dd8e28/0_0

3.4 State Diagrams:

UML state diagrams are used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by the events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, we examined all use cases that are relevant to the design class throughout its life. These use cases provided information that helped us to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) are represented using a UML statechart as illustrated in the mentioned.The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form:

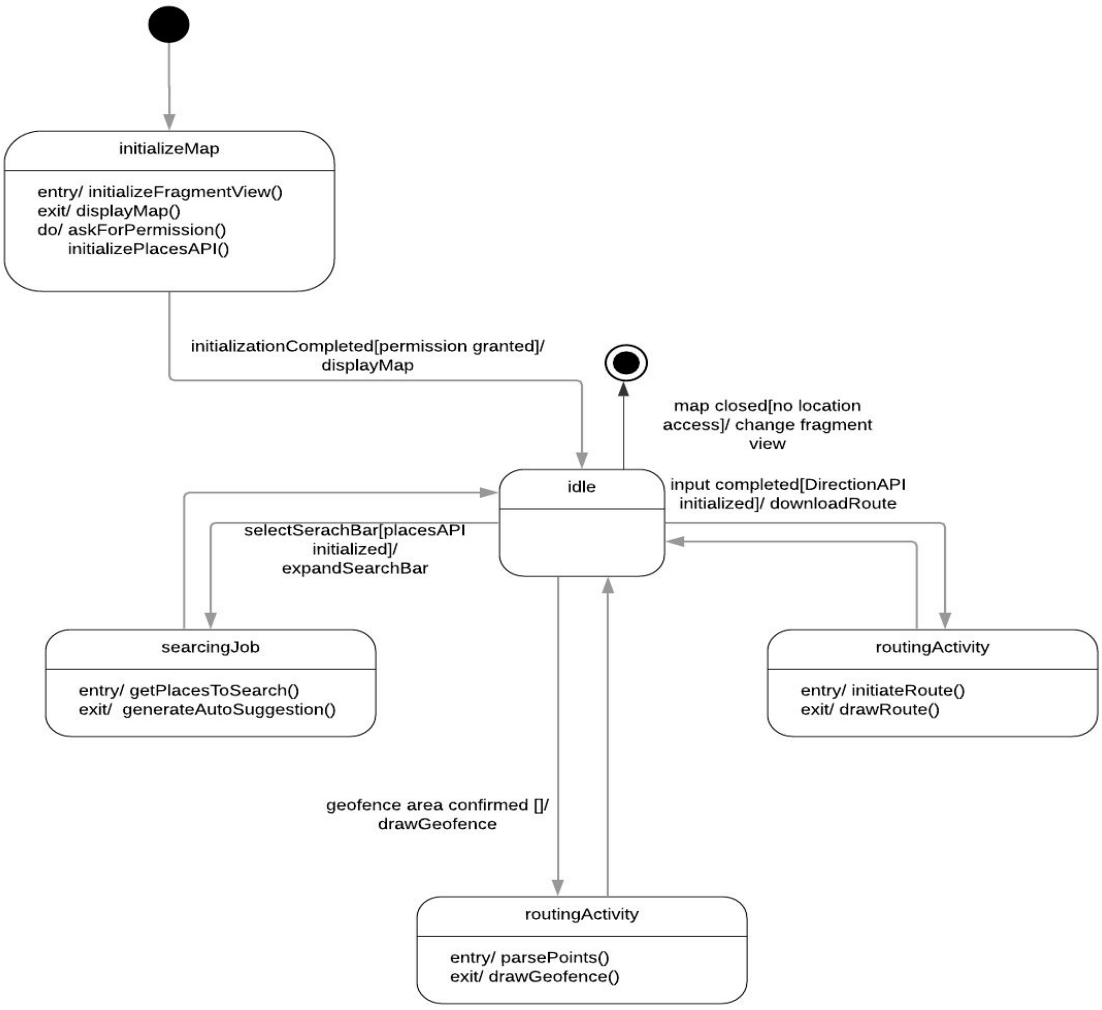
Event-name (parameter-list)[guard-condition]/action expression where event-name identifies the event, parameter-list incorporates data that are associated with the event, guard-condition is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and action expression defines an action that occurs as the transition takes place.

Referring to the following figures each state may define entry and exit/ actions that occur as a transition into the state occurs and as a transition out of the state occurs, respectively. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The do/ indicator provides a mechanism for indicating activities that occur while in the state, and the include/ indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

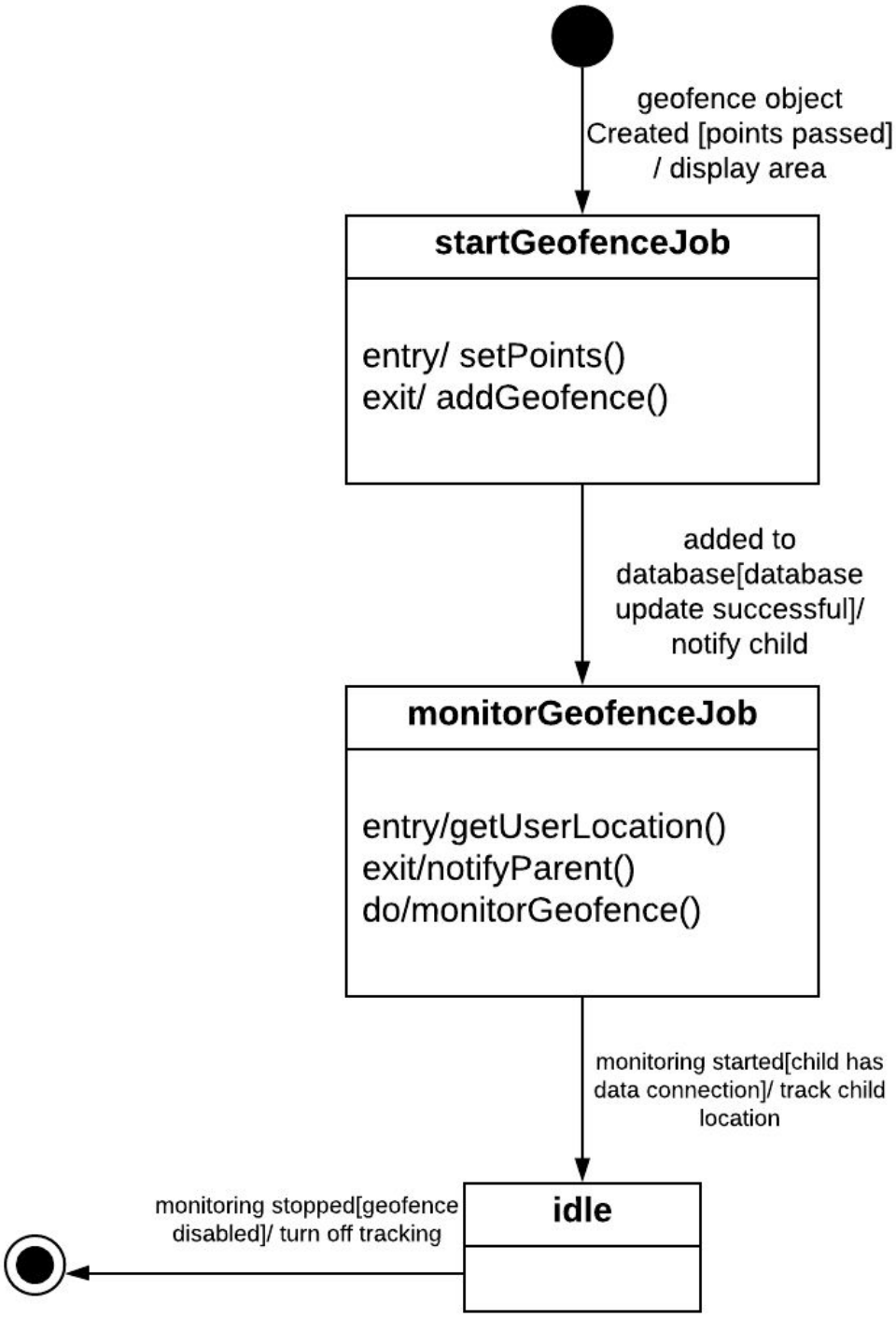
3.4.1 Map State Diagram:

When transition into the initializeMap state occurs (entry), initializeFragmentView() action takes place, as transition out of the state occurs (exit), displayMap() action takes place, while in the state (do), askForPermission() and initializePlacesAPI() activities are occured. **InitializationCompleted[permission granted]/displayMap :** this means when the guard-condition: permission granted must be met before the event: initializationCompleted can occur. As the transition takes place, action expression: displayMap occurs.

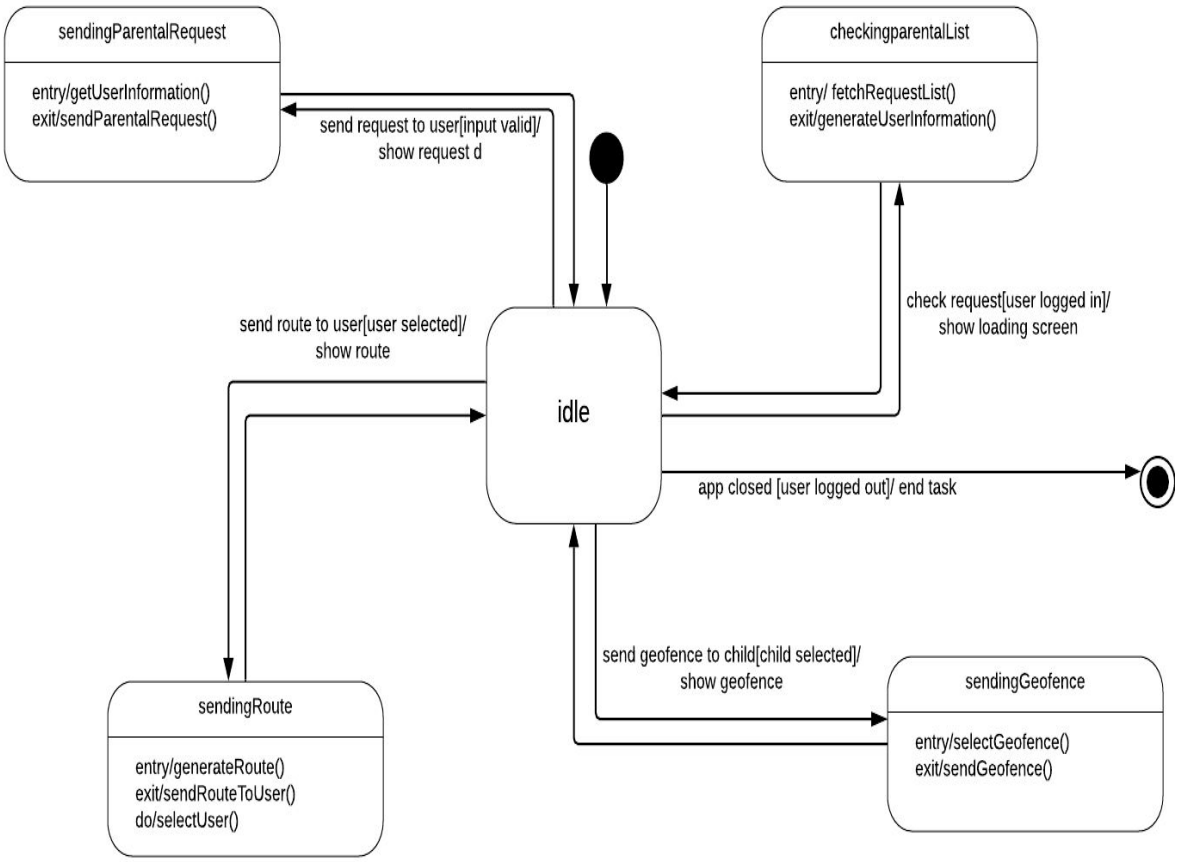
The state transitions are shown with arrows in the diagram:



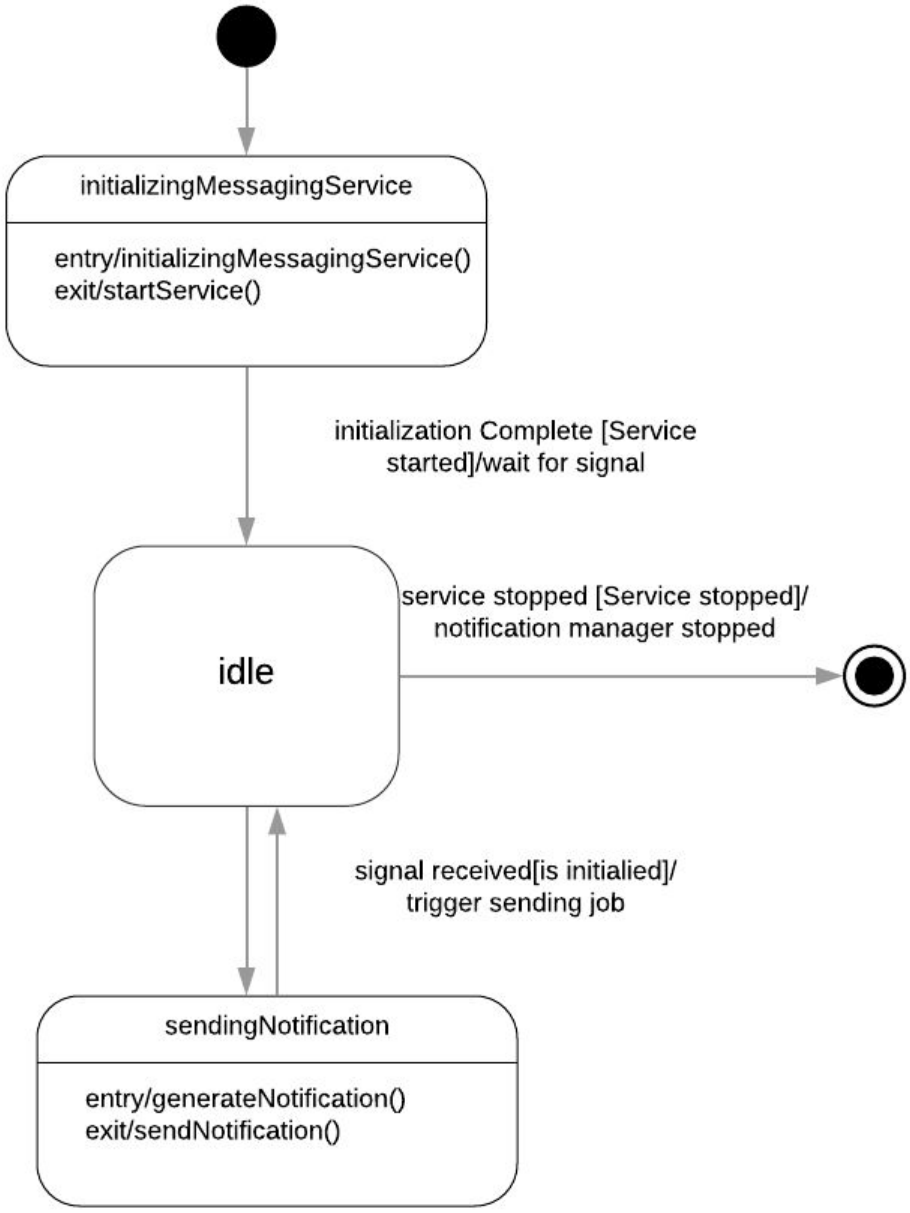
3.4.2 Geofence State Diagram:



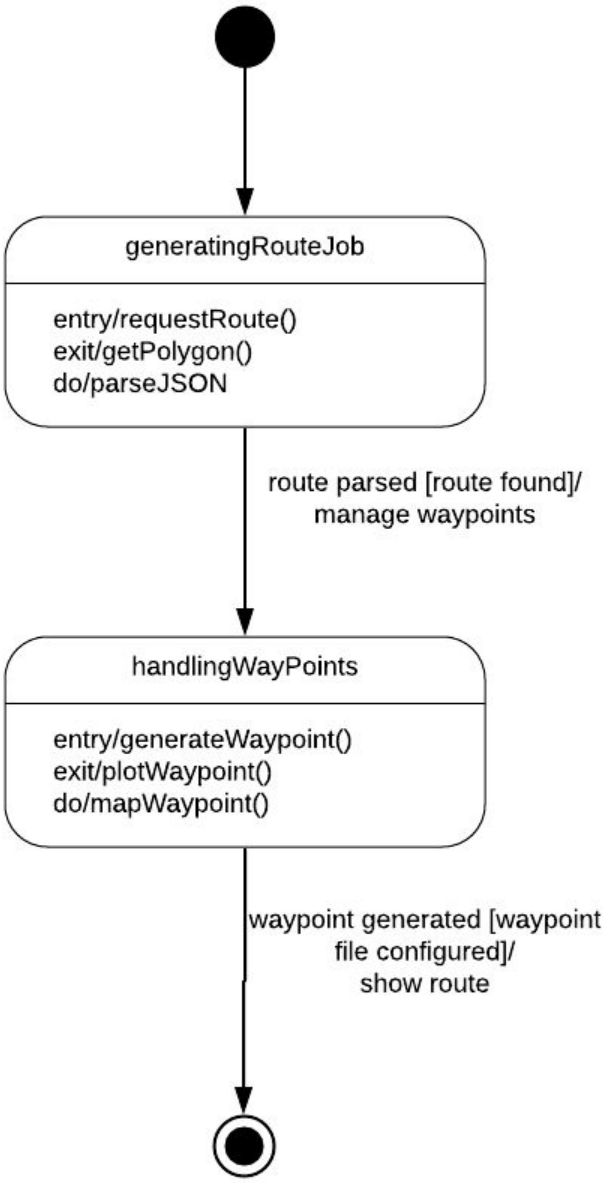
3.4.3 User State Diagram:



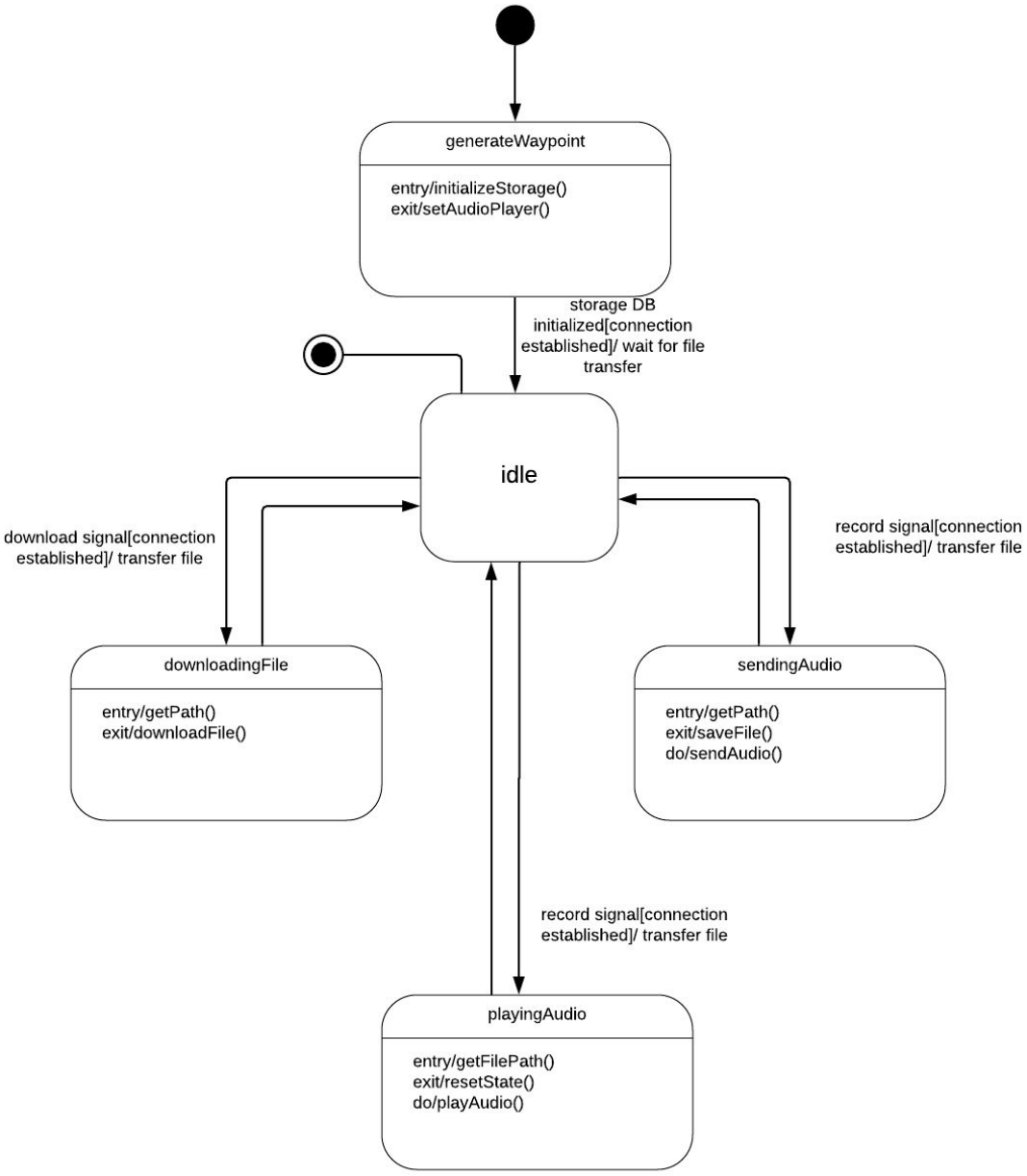
3.4.4 Notification State Diagram:



3.4.5 Route State Diagram:



3.4.6 Waypoint State Diagram:



3.5 Deployment Diagram

A deployment diagram is a UML diagram type that shows the execution architecture of a system, including nodes such as hardware or software execution environments, and the middleware connecting them.

Deployment diagrams are typically used to visualize the physical hardware and software of a system. Using it we can understand how the system will be physically deployed on the hardware.

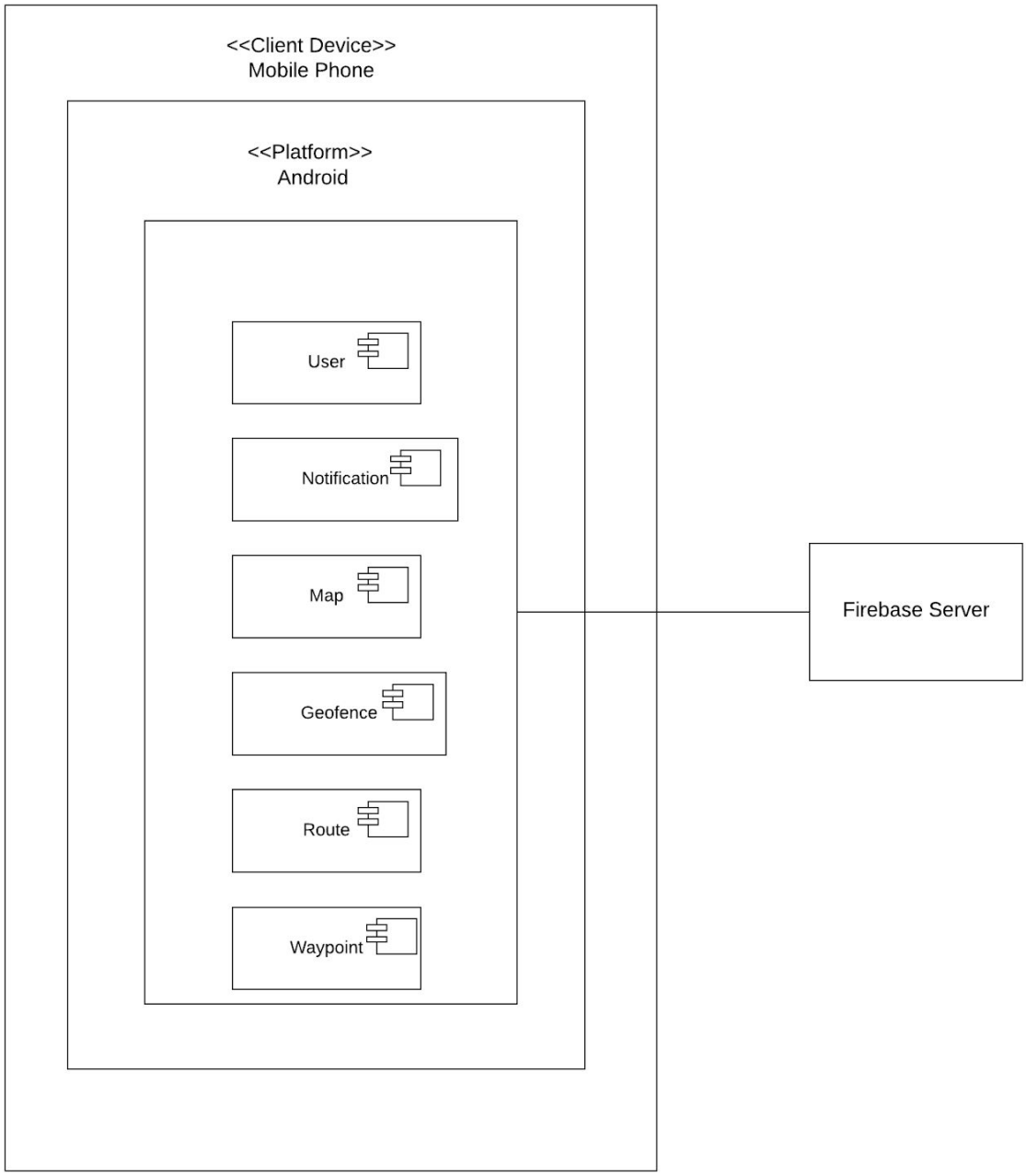
Deployment diagrams help model the hardware topology of a system compared to other UML diagram types which mostly outline the logical components of a system.

Node is a computational resource upon which artifacts are deployed for execution. A node is a physical thing that can execute one or more artifacts. A node may vary in its size depending upon the size of the project.

Node is an essential UML element that describes the execution of code and the communication between various entities of a system. It is denoted by a 3D box with the node-name written inside of it. Nodes help to convey the hardware which is used to deploy the software.

An association between nodes represents a communication path from which information is exchanged in any direction.

Here is the deployment diagram for the E-Shongee app. Here <<Client Device>> Mobile Phone is a node. It represents a physical machine where the app runs. <<Platform>> Android is also a node that represents the platform in which the app is going to be executed. Node: <<Platform>> Android is nested into Node: <<Client Device>> Mobile Phone. The 6 components are nested in the <<Client Device>> Mobile Phone node.


<<Client Device>>
Mobile Phone
<<Platform>>
Android
User
Notification
Map
Geofence
Route
Waypoint
Firebase Server