Architectural Requirements Specifications and Design

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Andrew le Roux	15311644
Cian Steenkamp	15095682
Darren Adams	14256232
Drew Langley	11039753
Martha Mohlala	10353403
Nsovo Baloyi	12163262
Rikard Schouwstra	15012299

10 March 2017

Contents

1 External Interface Requirements

1.1 User Interfaces

The mobile application will interface with the supported input and output features of the host's operating system. Inputs include text that the user will enter for login or searching a venue. Outputs include the type of fonts to display text or graphics to show images or draw the map.

1.2 Hardware Interfaces

Since neither the mobile application nor the web portal have any designated hardware, it does not have any direct hardware interfaces. The WiFi software in the mobile phone manages the built-in WiFi and the hardware connection to the database server is managed by the underlying operating system on the mobile phone and the web server.

1.3 Software Interfaces

The mobile application communicates with the WiFi software in order to get signal strength information from multiple WiFi access points to determine (using triangulation) where the user is located. The communication software between the database and mobile application consists of operation concerning creating, reading, removing and modifying the data.

1.4 Communication Interfaces

The communication between the different parts of the system are important since they depend on each other. However, in what way the communication is achieved is not important for the system and is therefore handled by the underlying operating systems for both the mobile application and the back-end of the system.

2 Performance Requirements

2.1 Performance

- Offline activities should have a response time of +/- 2 seconds (instantaneous) when responding to an activity, while online activities such as calculating routes should have a response time of +/- 2-4 seconds so that the users have an uninterrupted experience.

- It should also allow the integration of a variety of services.

2.2 Reliability

- It should be able to handle +/- 50 000 users concurrently (simultaneously) when implemented into a suitable production environment.

- The application should be reliable, in that it will provide the fastest route every time without fail and complete all other computations successfully.

- All activities should be completed with a 0.1 error allowance.

- The application should provide accurate locations in a constantly changing environment.

2.3 Security

- Data transmission should be securely transmitted without unauthorized access, or loss of information.

3 Design Constraints

- The system should be accessible on smart devices, such as Android and iOS devices.
- The system should not use GPS, but only the WiFi network.
- The proposed system should be able to be integrated into the Computer Science Department's Web site.
- The system should be a modular system, to reduce the dependencies in the system.
- Software Fault Tolerance: If a malfunction cannot be avoided, then the software design should be constrained so that the system can recover without causing damage to the system.
- The system should have an aesthetically pleasing and easy to use interface.
- The system must be able to run on smart devices which has limited processing power, battery life and storage space. The system must thus use resources efficiently.
- The system needs to use open source technologies.

4 System Software Attributes

4.1 Reliability

- Any information that is stored on the database must remain correct when being transferred to the user interface.
- The services offered by the system should be available to users except for when the system is undergoing maintenance.
- The system should reply to user requests in the shortest time interval possible.
- The system must be fault tolerant, it needs to maintain a certain level of performance and offer other services that are not affected by this fault to the users.
- In the event of a fault the system must be able to recover within the shortest time period possible and recover any data that may have been lost.
- The system should be able to respond appropriately if it receives bad input data from the user.

4.2 Scalability

- The system must be able to cater for increases in the work load, for example large number of users or activities at any given time, without impacting the performance of the system.
- If the system does not cater for increases in workload it should at least provide the ability to be readily enlarged.

4.3 Maintainability

- The system must be designed in a modular fashion that provides high cohesion and loose coupling, this will allow parts of the system to be easily maintained without affecting the rest of the system.
- Maintenance should be able to be carried out by different maintenance teams, therefore the system must be easy to learn and understand.

4.4 Integrability

- Since we are following a modular design, components of the system that are separately developed should work correctly together.
- Follow coding standards specified by the client to allow for easy integration and employ continuous integration in our design process.

4.5 Usability

- The system must be easy to learn.
- System must cater for user mistakes, by providing the user with the undo or roll back options.
- The user interface must be easy to use and must be intuitive.
- The system should display options in a logical manner.
- Incorporate widgets and icons that the target users may be familiar with.
- The user manual should have a detailed description of the system.
- A help option must be provided to the users.

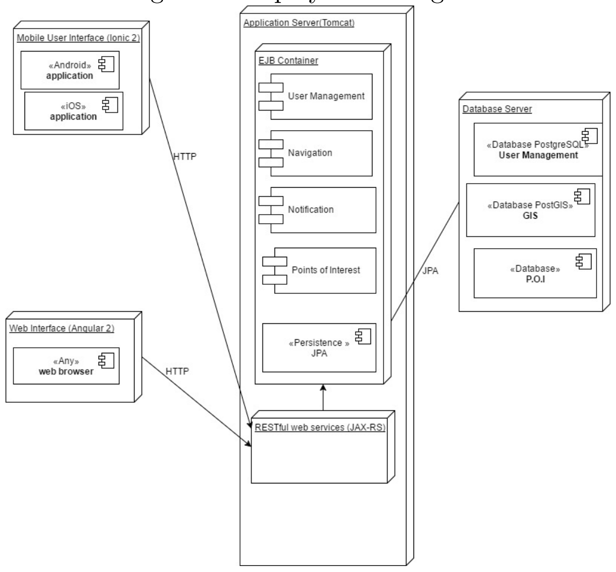
4.6 Interoperability

- The system must be able to communicate with the University of Pretoria WiFi system, because the WiFi access points will be used for the navigation.

5 Modules

5.1 Overview of System

Figure 1: Deployment Diagram



The figure above shows the deployment diagram. It show the various deployments of the components as well as the technologies to be used.

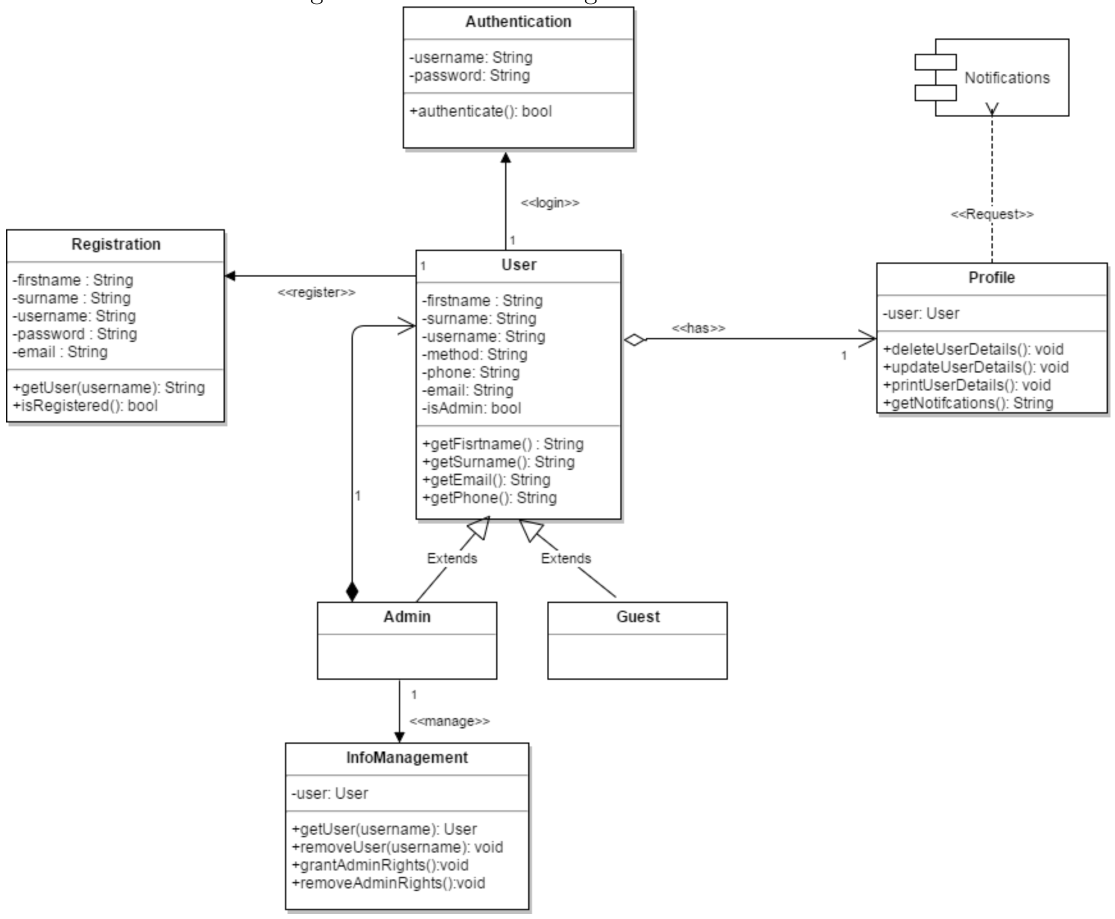
The system will implement a layered architectural pattern. This allows us to separate the user interfaces from the business processes as well as the database services. It adds flexibly and maintainability to the system.

The backend systems will be deployed on a JavaEE application server. Other devices may communicate with the server through an API there by implementing a façade. The web interface will run on any web browser and the mobile interface will be accessible on iOS and Android.

5.2 User Management

5.2.1 Class Diagram

Figure 2: Users Class Diagram



5.2.2 Activity Diagrams

Figure 3: Activity Diagram - Log In

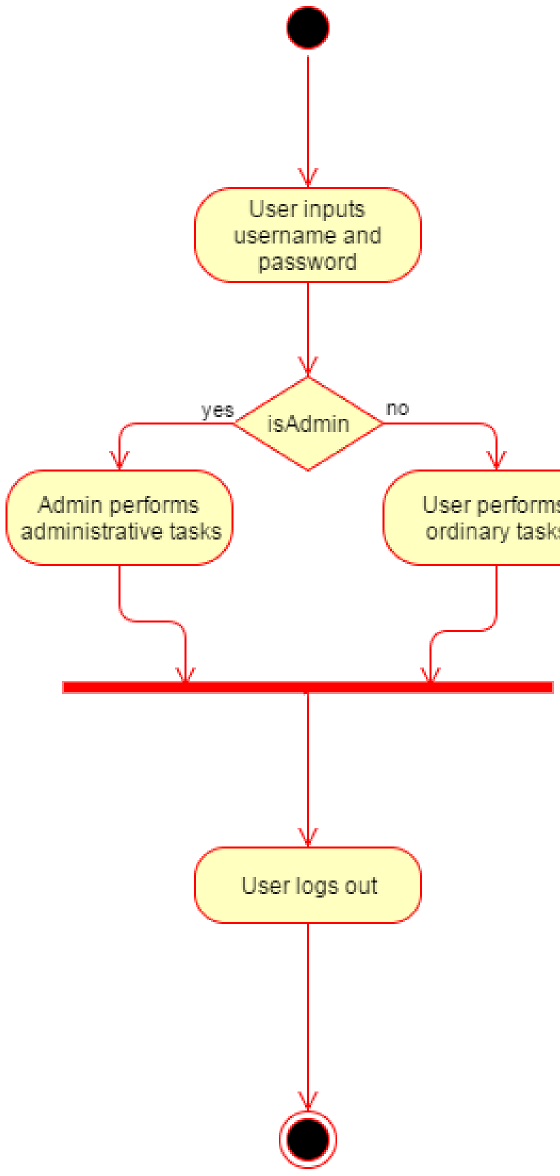

User inputs username and password
yes
isAdmin
no
Admin performs administrative tasks
User performs ordinary tasks
User logs out

Figure 4: Activity Diagram - Register

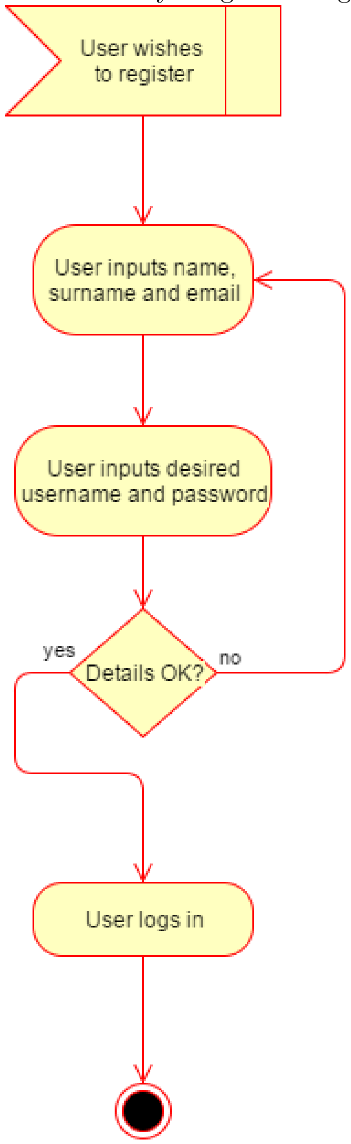
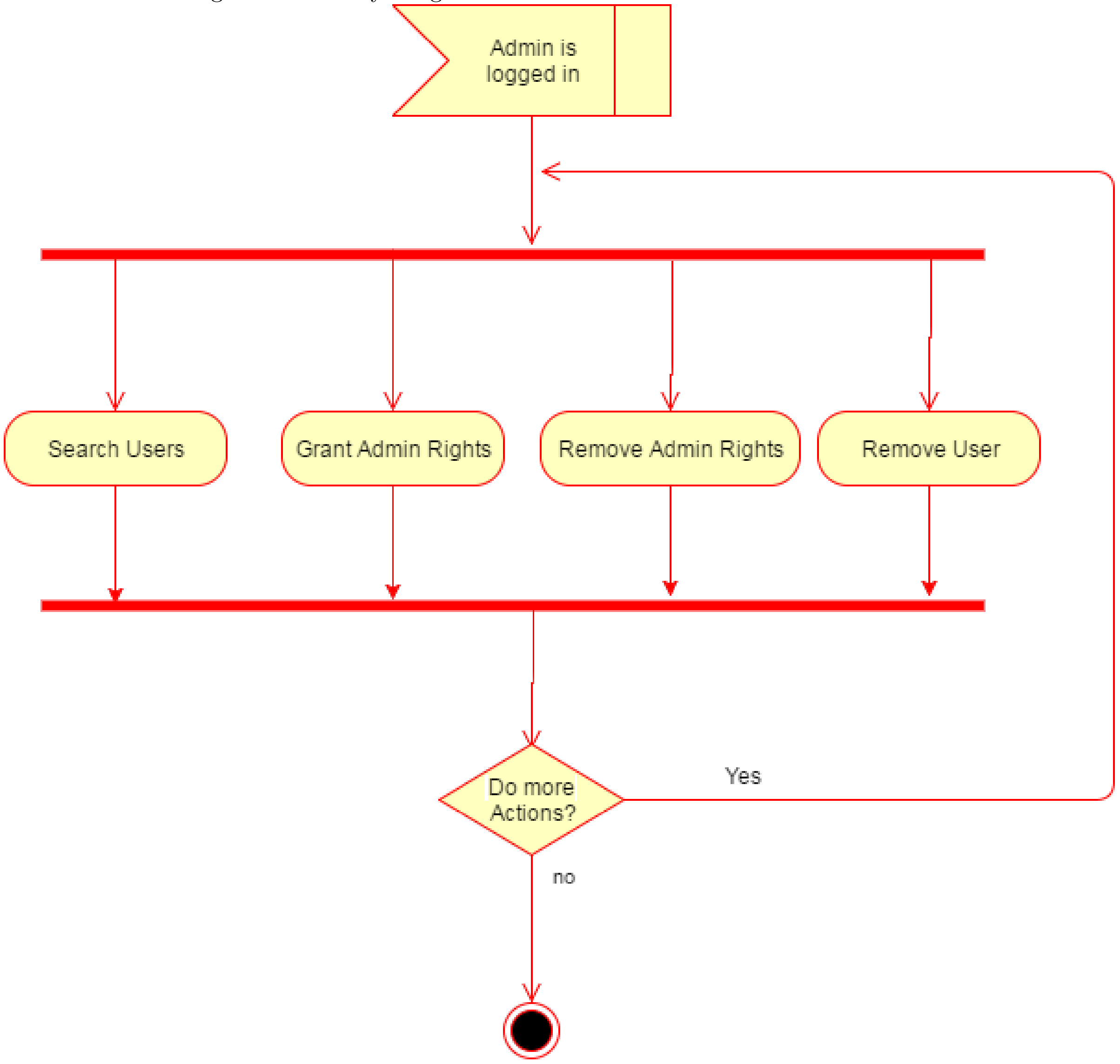
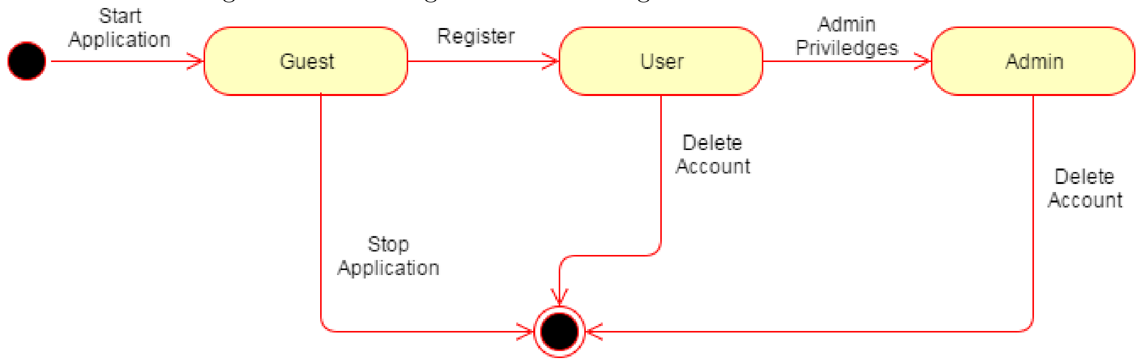

User wishes to register
User inputs name, surname and email
User inputs desired username and password
Details OK?
yes
no
User logs in

Figure 5: Activity Diagram - Administrators



5.2.3 State Diagram

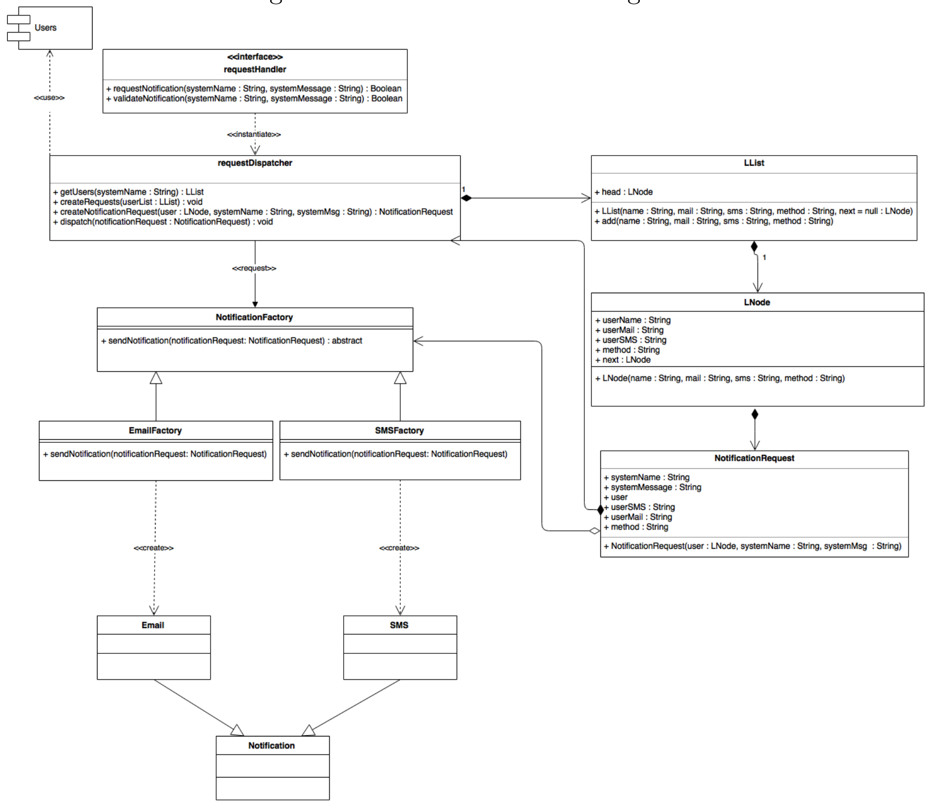
Figure 6: State Diagram - User Management



5.3 Notifications

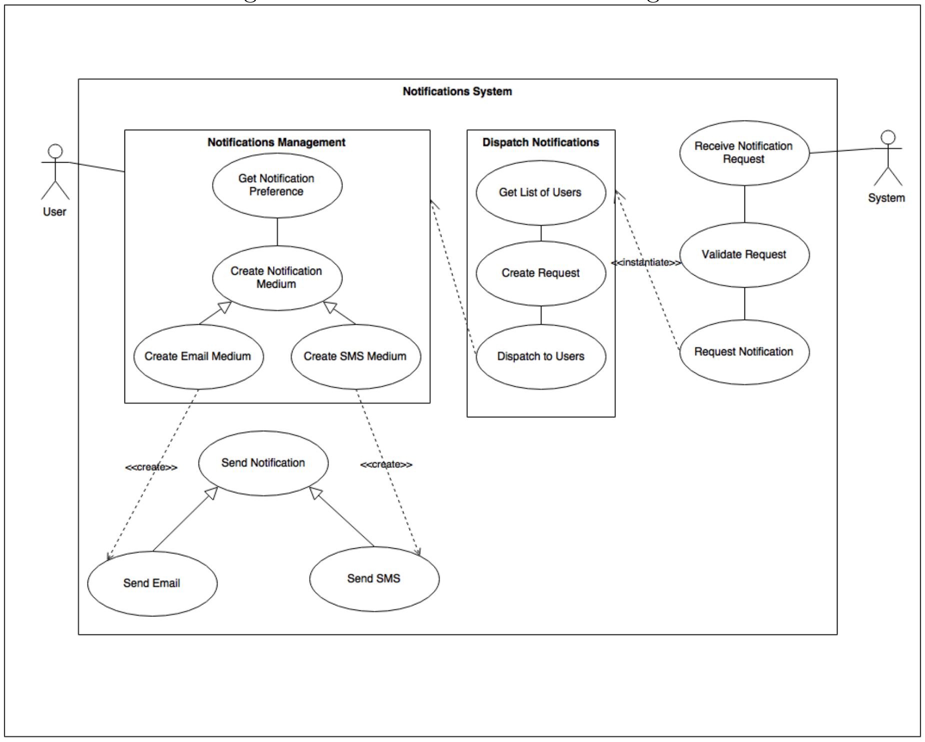
5.3.1 Class Diagram

Figure 7: Notifications Class Diagram


Users
<<interface>>
requestHandler
+ requestNotification(systemName : String, systemMessage : String) : Boolean
+ validateNotification(systemName : String, systemMessage : String) : Boolean
<<use>>
<<instantiate>>
requestDispatcher
+ getUsers(systemName : String) : LList
+ createRequests(userList : LList) : void
+ createNotificationRequest(user : LNode, systemName : String, systemMsg : String) : NotificationRequest
+ dispatch(notificationRequest : NotificationRequest) : void
1
LList
+ head : LNode
+ LList(name : String, mail : String, sms : String, method : String, next = null : LNode)
+ add(name : String, mail : String, sms : String, method : String)
1
<<request>>
NotificationFactory
+ sendNotification(notificationRequest: NotificationRequest) : abstract
LNode
+ userName : String
+ userMail : String
+ userSMS : String
+ method : String
+ next : LNode
+ LNode(name : String, mail : String, sms : String, method : String)
EmailFactory
+ sendNotification(notificationRequest: NotificationRequest)
SMSFactory
+ sendNotification(notificationRequest: NotificationRequest)
NotificationRequest
+ systemName : String
+ systemMessage : String
+ user
+ userSMS : String
+ userMail : String
+ method : String
+ NotificationRequest(user : LNode, systemName : String, systemMsg : String)
<<create>>
<<create>>
Email
SMS
Notification

5.3.2 Use Case Diagrams

Figure 8: Notifications Use Case Diagram



5.3.3 Chosen Design Patterns

We have chosen to incorporate the Factory Design Pattern in our design of the notifications module for NavUP. Due to the fact that notifications are generated and then sent to the respective mediums, it makes sense to implement the above pattern for the following reasons:

- Extendibility : Should NavUP need to facilitate a greater range of mediums to communicate system updates, and entire re-code of the module would not be necessary.
- Ease-of-use : The notifications management interface need not know what class of the object it is creating, instead leaving it up to the respective child classes to decide. This abstraction makes life easier for programmers to use the module.
- Modularity : Having the factory become an interface for sending through various mediums, creates a somewhat modular approach within the subsystem.

5.4 Navigation

5.4.1 Class Diagram

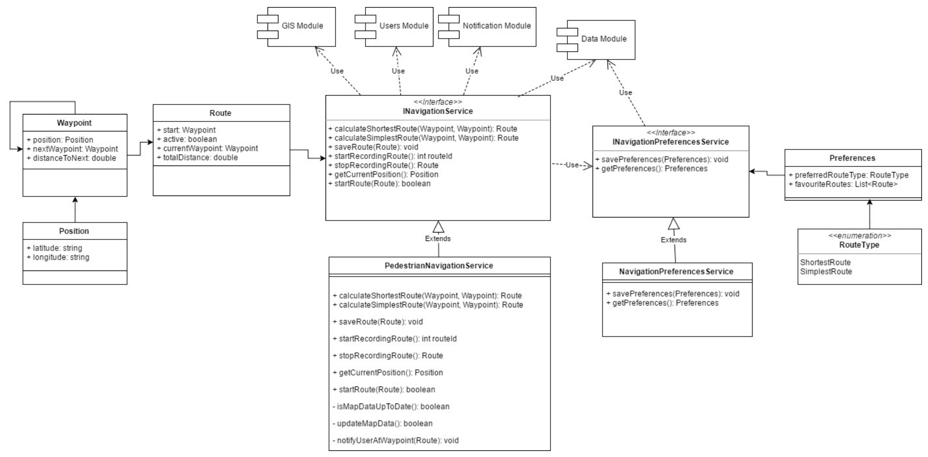

GIS Module
Users Module
Notification Module
Data Module
Use
Use
Use
Use
Use
Waypoint
+ position: Position
+ nextWaypoint: Waypoint
+ distanceToNext: double
Route
+ start: Waypoint
+ active: boolean
+ currentWaypoint: Waypoint
+ totalDistance: double
<<Interface>>
INavigationService
+ calculateShortestRoute(Waypoint, Waypoint): Route
+ calculateSimplestRoute(Waypoint, Waypoint): Route
+ saveRoute(Route): void
+ startRecordingRoute(): int routeId
+ stopRecordingRoute(): Route
+ getCurrentPosition(): Position
+ startRoute(Route): boolean
Use
<<Interface>>
INavigationPreferencesService
+ savePreferences(Preferences): void
+ getPreferences(): Preferences
Preferences
+ preferredRouteType: RouteType
+ favouriteRoutes: List<Route>
Position
+ latitude: string
+ longitude: string
Extends
PedestrianNavigationService
+ calculateShortestRoute(Waypoint, Waypoint): Route
+ calculateSimplestRoute(Waypoint, Waypoint): Route
+ saveRoute(Route): void
+ startRecordingRoute(): int routeId
+ stopRecordingRoute(): Route
+ getCurrentPosition(): Position
+ startRoute(Route): boolean
- isMapDataUpToDate(): boolean
- updateMapData(): boolean
- notifyUserAtWaypoint(Route): void
Extends
NavigationPreferencesService
+ savePreferences(Preferences): void
+ getPreferences(): Preferences
<<enumeration>>
RouteType
ShortestRoute
SimplestRoute

Figure 9: Navigation Module Class Diagram

5.4.2 Use Case Diagrams

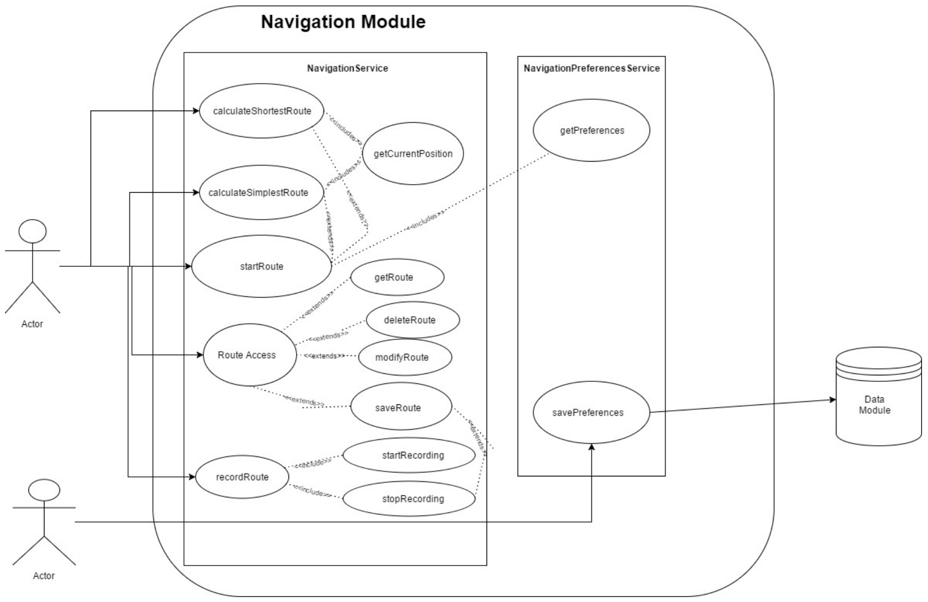


Figure 10: Navigation Module Use Case Diagram

5.4.3 Other Diagrams

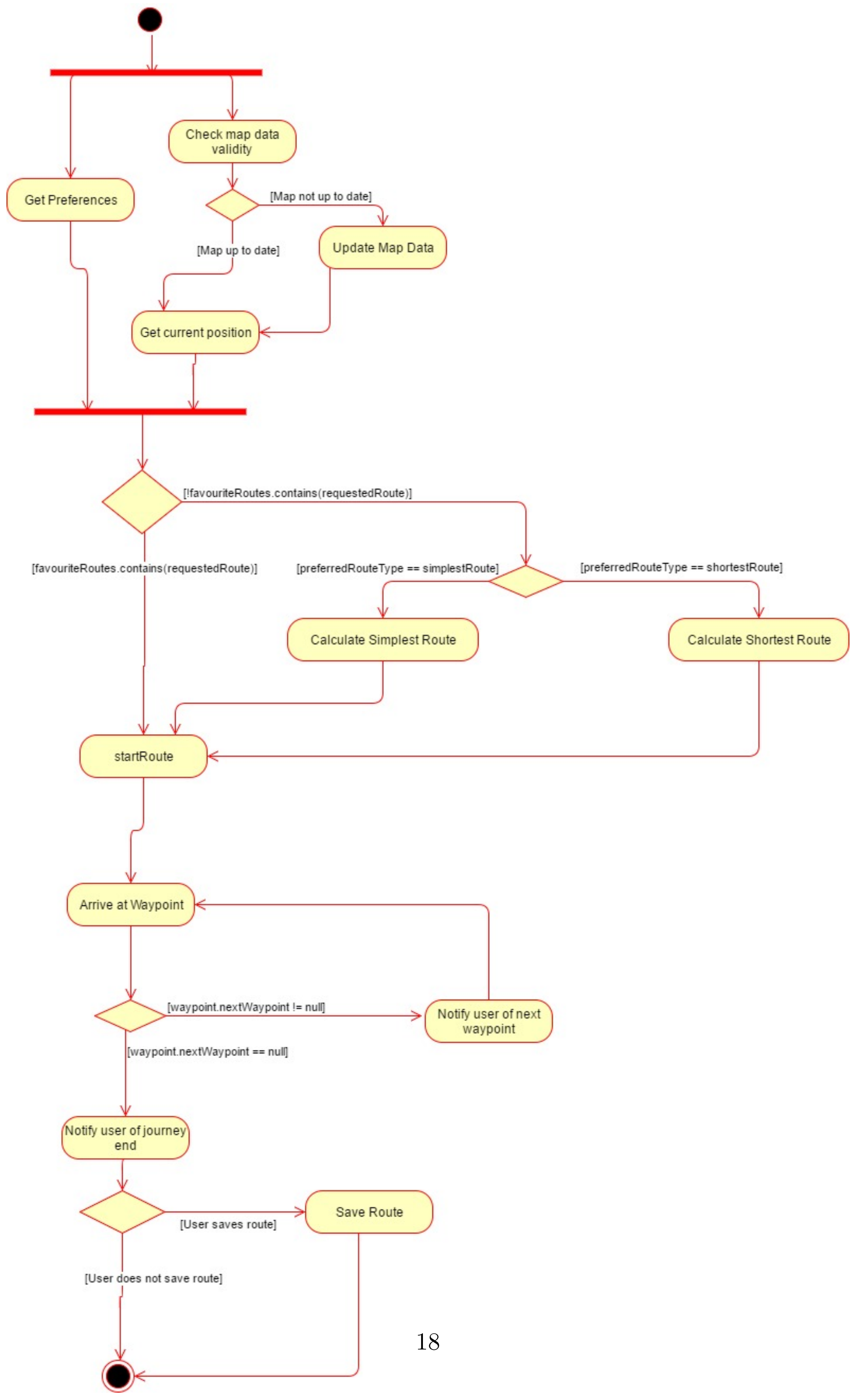

Check map data validity
Get Preferences
[Map not up to date]
Update Map Data
[Map up to date]
Get current position
[!favouriteRoutes.contains(requestedRoute)]
[favouriteRoutes.contains(requestedRoute)]
[preferredRouteType == simplestRoute]
[preferredRouteType == shortestRoute]
Calculate Simplest Route
Calculate Shortest Route
startRoute
Arrive at Waypoint
[waypoint.nextWaypoint != null]
Notify user of next waypoint
[waypoint.nextWaypoint == null]
Notify user of journey end
[User saves route]
Save Route
[User does not save route]

Figure 11: Navigation Module Activity Diagram

5.5 Points-of-interest

5.5.1 Class Diagram

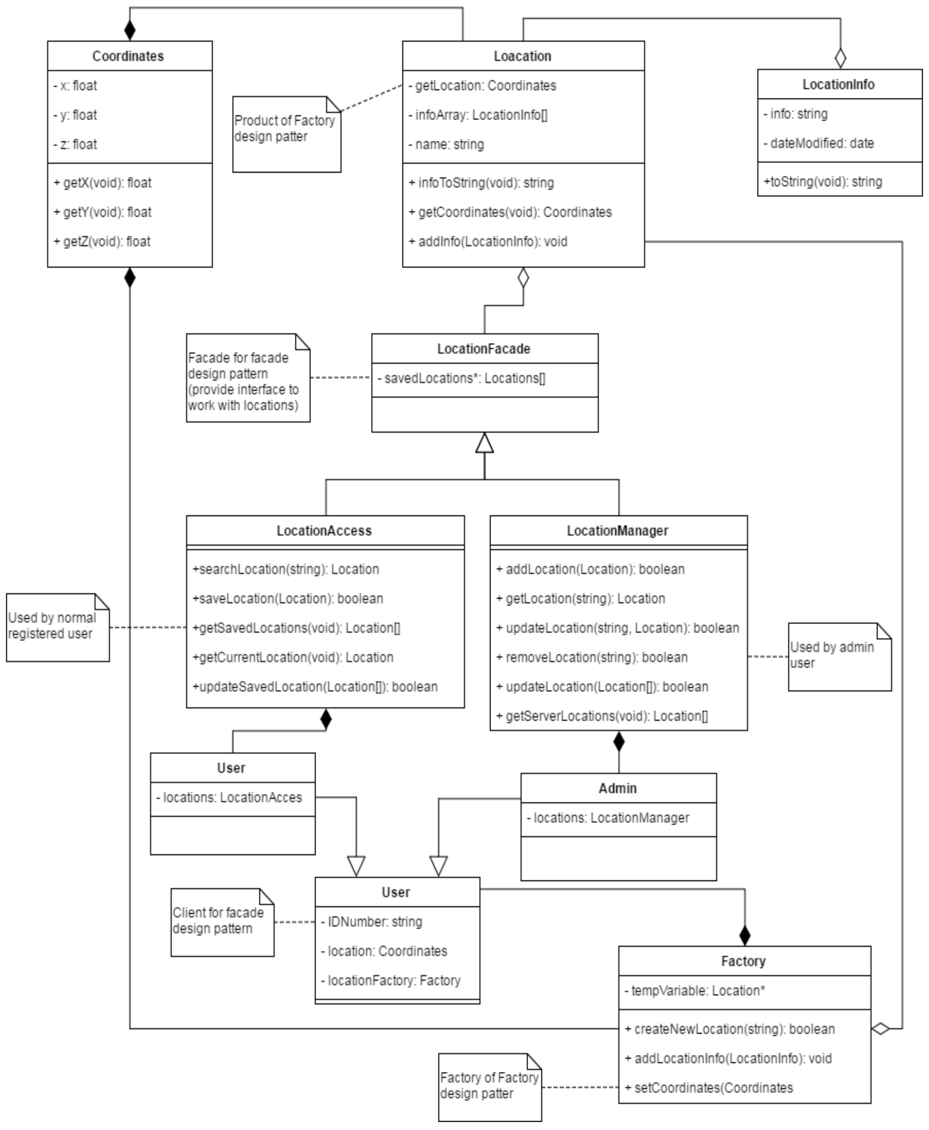

Coordinates
- x: float
- y: float
- z: float
+ getX(void): float
+ getY(void): float
+ getZ(void): float
Loacation
- getLocation: Coordinates
- infoArray: LocationInfo[]
- name: string
+ infoToString(void): string
+ getCoordinates(void): Coordinates
+ addInfo(LocationInfo): void
LocationInfo
- info: string
- dateModified: date
+toString(void): string
Product of Factory design patter
LocationFacade
- savedLocations*: Locations[]
Facade for facade design pattern (provide interface to work with locations)
LocationAccess
+searchLocation(string): Location
+saveLocation(Location): boolean
+getSavedLocations(void): Location[]
+getCurrentLocation(void): Location
+updateSavedLocation(Location[]): boolean
LocationManager
+ addLocation(Location): boolean
+ getLocation(string): Location
+ updateLocation(string, Location): boolean
+ removeLocation(string): boolean
+ updateLocation(Location[]): boolean
+ getServerLocations(void): Location[]
Used by normal registered user
Used by admin user
User
- locations: LocationAcces
Admin
- locations: LocationManager
User
- IDNumber: string
- location: Coordinates
- locationFactory: Factory
Client for facade design pattern
Factory
- tempVariable: Location*
+ createNewLocation(string): boolean
+ addLocationInfo(LocationInfo): void
+ setCoordinates(Coordinates
Factory of Factory design patter

Figure 12: Points of Interest Class Diagram

5.5.2 Use Case Diagram

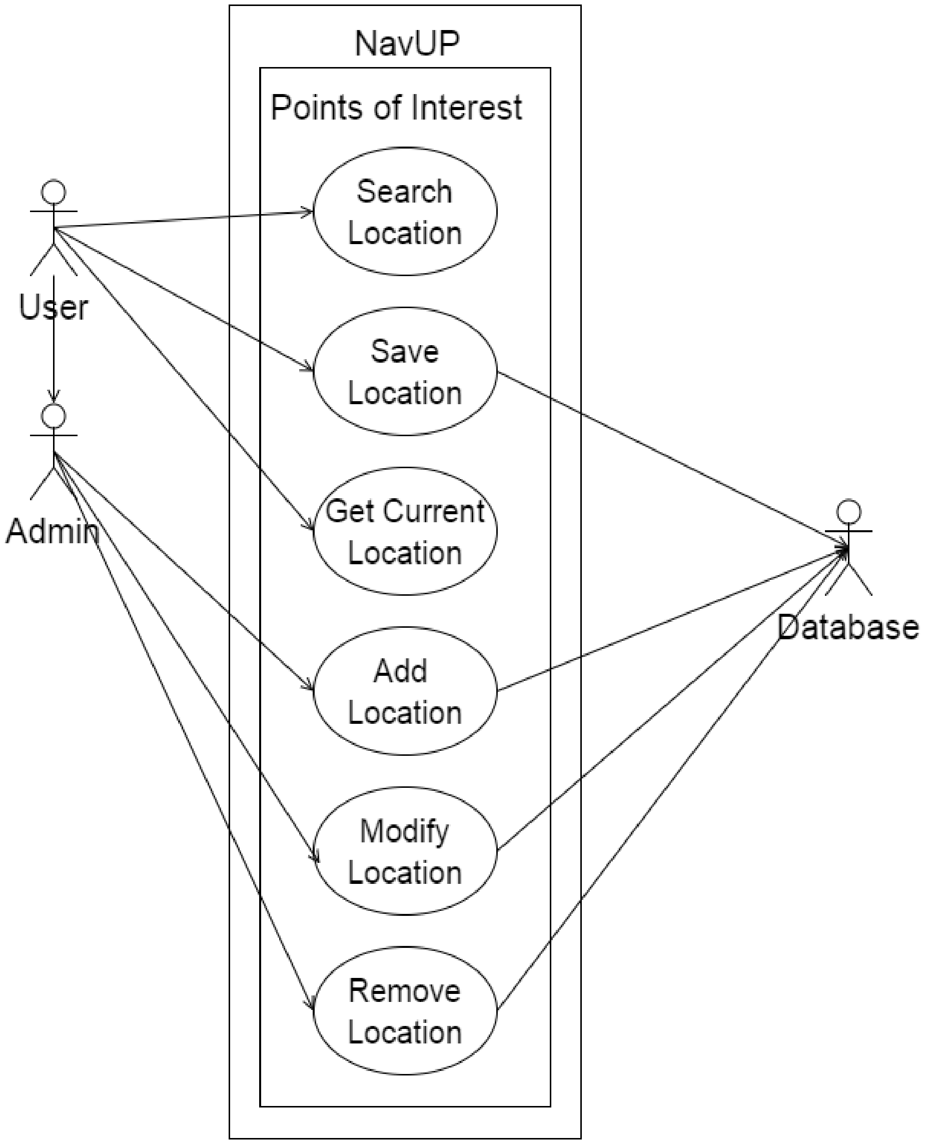


Figure 13: Points of Interest Use Case Diagram

5.5.3 Chosen Design Patterns

For the Points of Interest Class Diagram the Factory and Facade design patterns are used. The Facade design pattern provides a unified interface to a set of interfaces in the Points of Interest subsystem. The Facade defines a higherlevel interface for the Points of Interest that makes the subsystem easier to use. Thus it wraps a complicated subsystem with a simpler interface. The Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

6 Technology Choices

6.1 Application Server

The application server is the component responsible for hosting the business process layer of the NavUP system. It needs to be able to satisfy quality requirements mentioned in System Software Attributes section. The NavUP system will follow a layered reference architecture approached as well as implement a Services-Oriented Architecture(SOA) approach. Together they allow the system to be flexible, easily maintainable and easily deployable.

JavaEE (Java Platform, Enterprise Edition) architecture is is one of the most widely used reference architectures for large, interactive enterprise systems.JavaEE supports, amongst other things, standard access channels for enterprise systems, a solid process execution environment, a range of standard integration channels, hot deployment and clustering. The main aim is to provide a reference architecture for scalable, reliable, deploy-ability, flexibility and modularity.

The application server will run on Tomcat 8 which is an open-source Java Servlet Container which offers great support for the JavaEE. It allows for easy deployment and modularity. Using the Web Services API it also interacts well with other technologies.

6.2 Database

The system has strict scalability, reliability and flexibility quality requirements to satisfy. Many users will be connected to the system at the same time and the DBMS (Database Management System) needs to be reliable. It is important that a DBMS is chosen for which both the scalability and reliability requirements can be met. For this reason the system will use *PostgreSQL* which is a reliable object-relational database.

The NavUP system also needs to have a database to satisfy the requirements of the GIS subsystem for this purpose *PostGIS* will be used. It is a spatial database extender for*PostgresSQL*, so it adds support for geographical objects.

Both *PostgreSQL* and *PostGIS* have great JavaEE support through the use of a persistence API. They also have great community support.

6.3 Persistence API

The persistence API is middleware that provides a layer of abstraction between the persistence provider (database) and the application. It is separated from the database technology as well as the database selected for the system. Java EE offers JPA as its persistence API. JPA is considered as a standard approach for Object to Relational Mapping (ORM) in Java.

JavaEE again is a great choice to carry out the requirements of this system.

6.4 Web Services Framework

The web services framework is a wrapping layer of the application services (business processes) layer making the system services available over the Internet. This allows the application server to be separated from user-interface technologies.The web services framework should be based on a public standard and open-source implementation should be available.

For the web services framework the project will use the Jersey implementation of the JAX-RS provided by JavaEE. A RESTful approach will be used.

6.5 Mobile Application Frameworks

The system does not have any particular mobile OS requirement, for this purpose a hybrid mobile application framework will be used to build the mobile interface part of the system.

Ionic 2 is hybrid mobile application development framework build on top of Angular and Apache Cordova. *Ionic 2* provides tools and services for developing hybrid mobile applications using web technologies like CSS, HTML5 and SASS. Since the NavUP system is going to be designed for both iOS and Android smartphones, *Ionic 2* is the best fit framework to develop the mobile application.

It uses MVC (Model, View and Controller) architectural pattern which separates logical concerns.

Other Frameworks Considered

1. **PhoneGap** : Very old and outdated compared to Ionic
2. **NativeScript** : still new and has less community support

6.6 Web Application Framework

The web application architecture will be built using*Angular*. *Angular* is a rewrite of *AngularJS* which is based on the MVVM and MVC architectures.

Other Frameworks Considered

1. EmberJS
2. ReactJS