NavUP
Architectural Requirements, Specifications and Design

Compiled By

Boikanyo Modiko - u15227678
Linda Potgieter - u14070091
Marthinus Richter - u15160671
Nkosenhle Ncube - u13247914
Quinton Swanepoel - u15245510
Ruan Klinkert - u14022282

201

TEAM Tlc

1 Introduction

1.1 Deliverable

We are required to use the high level functional requirements of the NavUp system given in a separate document to identify the architectural design specifications that satisfy the identified functional requirements from our last assignment, focusing on the subsystems architectural designs.

1.2 Scope

The final objective is to create a NavUP mobile application as well as a web-based interface. The NavUP mobile app should be available on both android and ios platforms while the web-based interface will be used for administration and maintenance. The primary objective of NavUP is to allow students, visitors and staff to successfully navigate around campus in a efficient manner.

1.3 Definitions, Acronyms and Abbreviations

Term	**Definition**
App	Mobile application
CRUD	Create, Read, Update and Delete
GPS	Global Positioning System
TUCBW	This Use Case Begins With
TUCEW	This Use Case Ends With
UP	University of Pretoria

2 Overall Architectural Design

2.1 Overview

2.2 Deployment Diagram

Figure 1: Deployment diagram

3 User Management Module

3.1 UML Class Diagram

3.2 Use Case Diagram

3.3 Architectural Design of Module

The user management module will handle the login and registration of various users, namely Admin Users, Registered Users, and Guest Users. Guest users will still be able login into and make use of the NavUP system without registering but no data will saved on the system for them. The system stores all registered users' details including those from other modules.

The user management system makes use of two major design patterns, Template and Memento.

3.4 Technologies Used

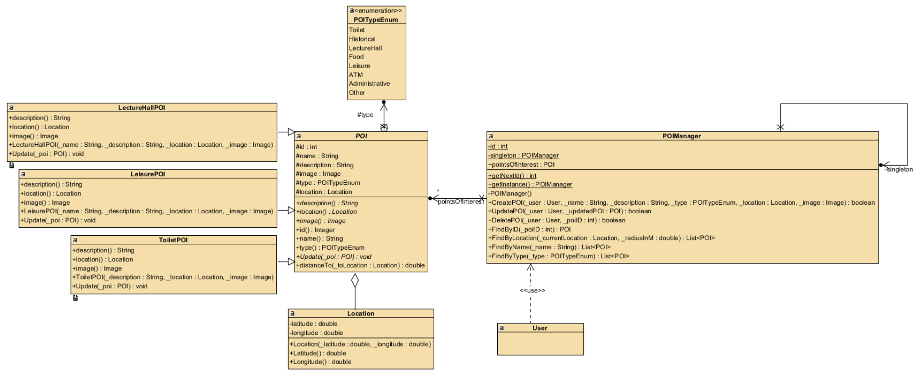
A MySQL database will be used to manage basic user activities such as registration, authentication, and user privileges. The database will store information regarding all users and how each user should be able to interact with the system.

The admin user should be able to perform basic CRUD operations such as adding and removing users, creating events, and updating details regarding points of interest.

3.5 Non-Trivial Implementation Tasks

4 Points of Interest Module

4.1 UML Class Diagram



<<enumeration>>
POITypeEnum
Toilet
Historical
LectureHall
Food
Leisure
ATM
Administrative
Other
#type
LectureHallPOI
+description() : String
+location() : Location
+image() : Image
+LectureHallPOI(_name : String, _description : String, _location : Location, _image : Image)
+Update(_poi : POI) : void
LeisurePOI
+description() : String
+location() : Location
+image() : Image
+LeisurePOI(_name : String, _description : String, _location : Location, _image : Image)
+Update(_poi : POI) : void
ToiletPOI
+description() : String
+location() : Location
+image() : Image
+ToiletPOI(_description : String, _location : Location, _image : Image)
+Update(_poi : POI) : void
POI
#id : int
#name : String
#description : String
#image : Image
#type : POITypeEnum
#location : Location
+description() : String
+location() : Location
+image() : Image
+id() : Integer
+name() : String
+type() : POITypeEnum
+Update(_poi : POI) : void
+distanceTo(_toLocation : Location) : double
*
pointsOfInterest
POIManager
-id : int
-singleton : POIManager
~pointsOfInterest : POI
+getNextId() : int
+getInstance() : POIManager
-POIManager()
+CreatePOI(_user : User, _name : String, _description : String, _type : POITypeEnum, _location : Location, _image : Image) : boolean
+UpdatePOI(_user : User, _updatedPOI : POI) : boolean
+DeletePOI(_user : User, _poiID : int) : boolean
+FindByID(_poiID : int) : POI
+FindByLocation(_currentLocation : Location, _radiusInM : double) : List<POI>
+FindByName(_name : String) : List<POI>
+FindByType(_type : POITypeEnum) : List<POI>
-singleton
<<use>>
User
Location
-latitude : double
-longitude : double
+Location(_latitude : double, _longitude : double)
+Latitude() : double
+Longitude() : double

Figure 2: Class Diagram for the Points of interest module

4.2 Use Case Diagram

4.3 Architectural Design of Module

The Points of Interest module will be used to persist and deliver data regarding locations that a user might find of interest. It will interact with the Events module, as well as the GIS module in order to retrieve and save it geographical location.

One can assume that different types of Points of Interest(POI) will have different fields, constraints and requirements, therefore the POI module will make use of the factory design pattern, in order to create objects without exposing the creation logic to the client and allow us to refer to the newly created object using a common interface.

The Points of Interest Manager class will make use of the Singleton design pattern to ensure that only one instance of the class exists, it will also act as the factory class to the points of interest class.

At the core a Point of Interest object will have a location, description, unique id, type, name and image. The concrete object will then be able to impose additional attributes and constraints, such as Gender for bathrooms for example.

Finally, the abstract point of interest class will be equipped with a stringify function to get a JSON representation of the object.

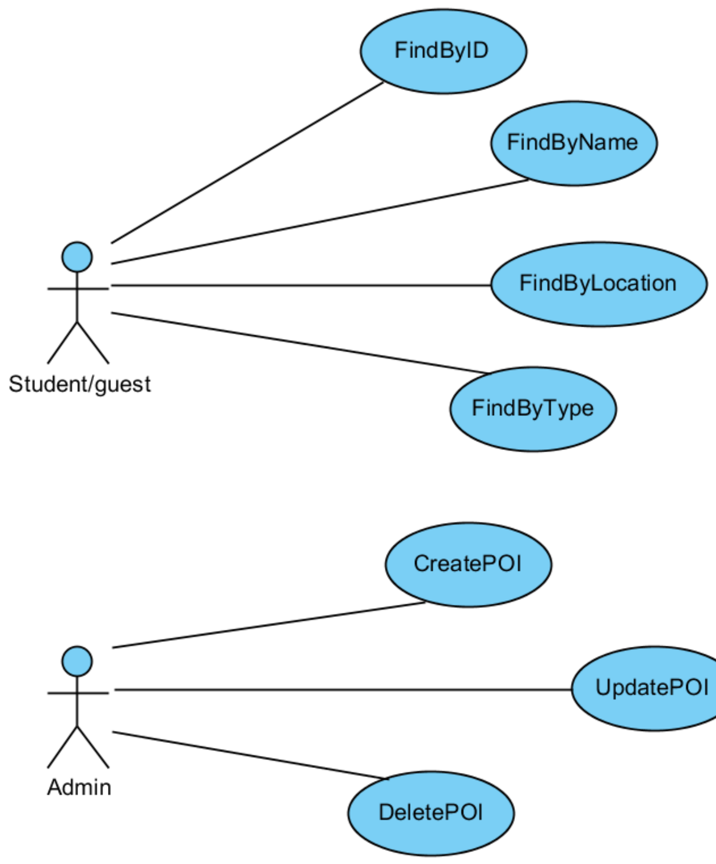


Figure 3: Use Case Diagram for the Points of interest module

4.4 Technologies Used

It can be assumed that different types of Points of Interest will have different fields, constraints and requirements, therefore a traditional relational database will not be suitable for this type of data. Hence we will rather go for a document-oriented database like MongoDB, as the database server.

The Points of Interest CRUD functions will be exposed as Simple Object Access Protocol (SOAP) web services, using ASP.NET core and Entity framework carrying a JSON payload.

The logic of the module will be written in .NET.

4.5 Non-Trivial Implementation Tasks

The point of interest module will be called in two ways. Firstly, by administrative user who would be able to add, remove and update points of interest around campus. Secondly, students and guests will be able to use it, by searching for specific locations by name, type of location, specific areas etc. Lastly the points of interest classes SearchByLocation function will be called by the navigation module, as the user navigates campus.

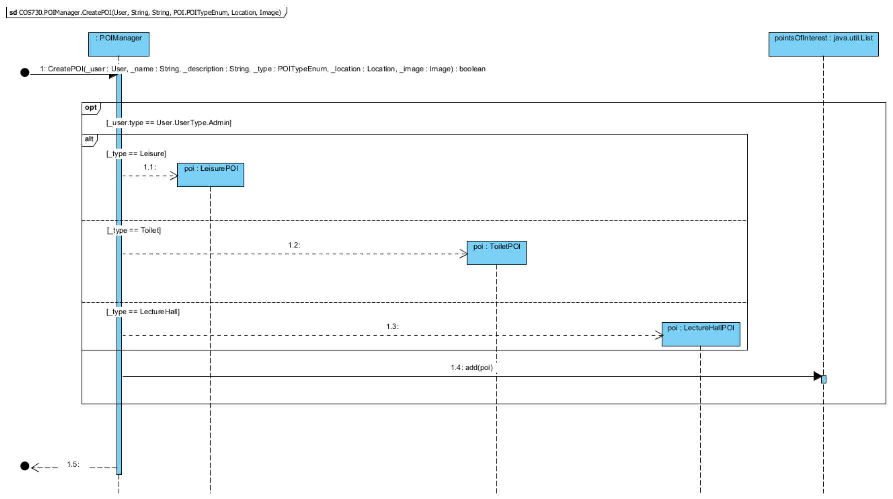

sd COS730.POIManager.CreatePOI(User, String, String, POI.POITypeEnum, Location, Image)
: POIManager
pointsOfInterest : java.util.List
1: CreatePOI(_user : User, _name : String, _description : String, _type : POITypeEnum, _location : Location, _image : Image) : boolean
opt
[_user.type == User.UserType.Admin]
alt
[_type == Leisure]
1.1:
poi : LeisurePOI
[_type == Toilet]
1.2:
poi : ToiletPOI
[_type == LectureHall]
1.3:
poi : LectureHallPOI
1.4: add(poi)
1.5:

Figure 4: Sequence Diagram for the CreatePOI function

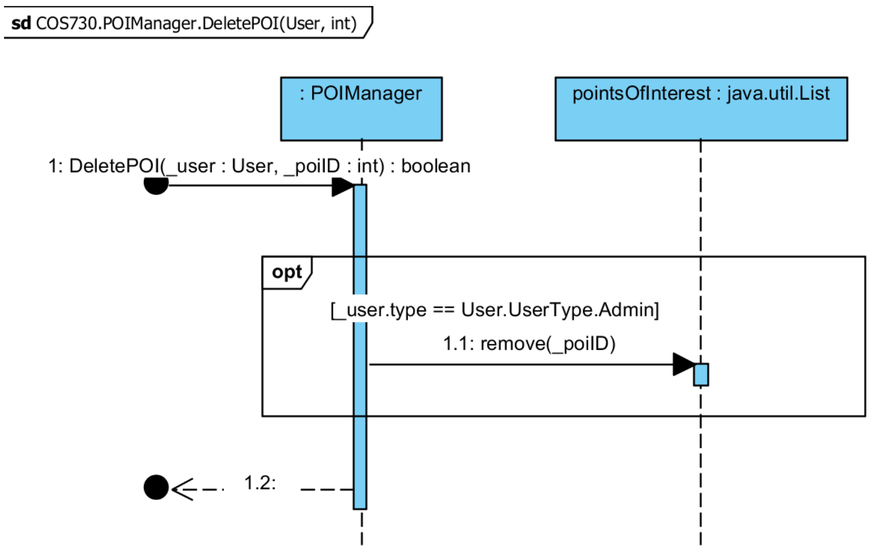

sd COS730.POIManager.DeletePOI(User, int)
: POIManager
pointsOfInterest : java.util.List
1: DeletePOI(_user : User, _poiID : int) : boolean
opt
[_user.type == User.UserType.Admin]
1.1: remove(_poiID)
1.2:

Figure 5: Sequence Diagram for the DeletePOI function

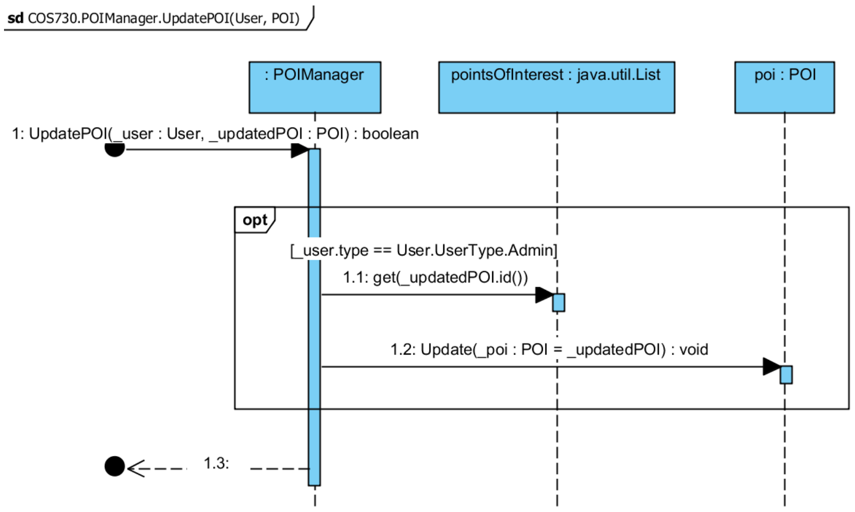

sd COS730.POIManager.UpdatePOI(User, POI)
: POIManager
pointsOfInterest : java.util.List
poi : POI
1: UpdatePOI(_user : User, _updatedPOI : POI) : boolean
opt
[_user.type == User.UserType.Admin]
1.1: get(_updatedPOI.id())
1.2: Update(_poi : POI = _updatedPOI) : void
1.3:

Figure 6: Sequence Diagram for the UpdatePOI function

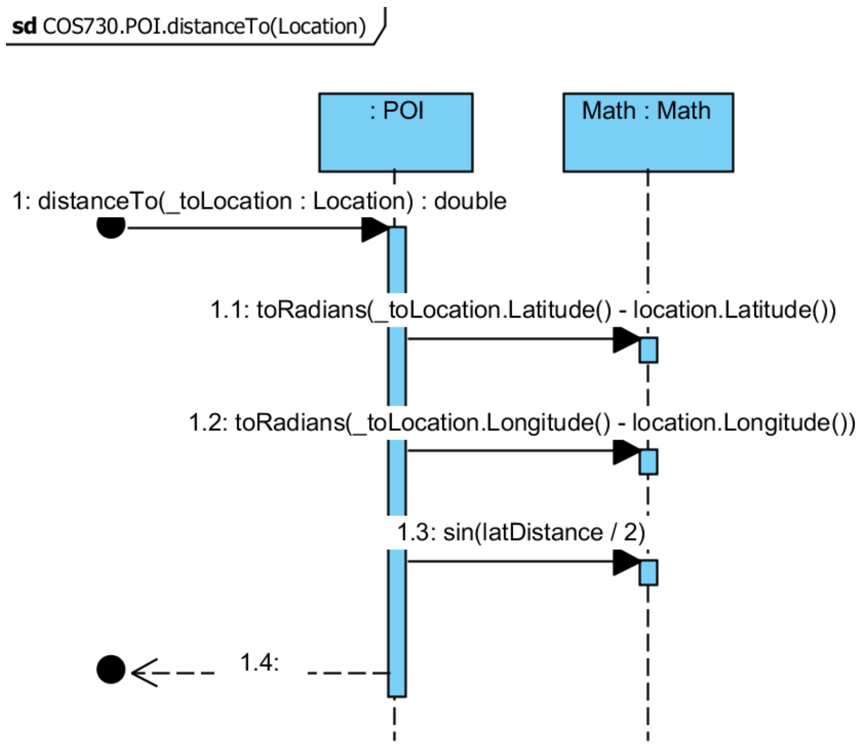

sd COS730.POI.distanceTo(Location)
: POI
Math : Math
1: distanceTo(_toLocation : Location) : double
1.1: toRadians(_toLocation.Latitude() - location.Latitude())
1.2: toRadians(_toLocation.Longitude() - location.Longitude())
1.3: sin(latDistance / 2)
1.4:

Figure 7: Sequence Diagram for the DistanceTo function

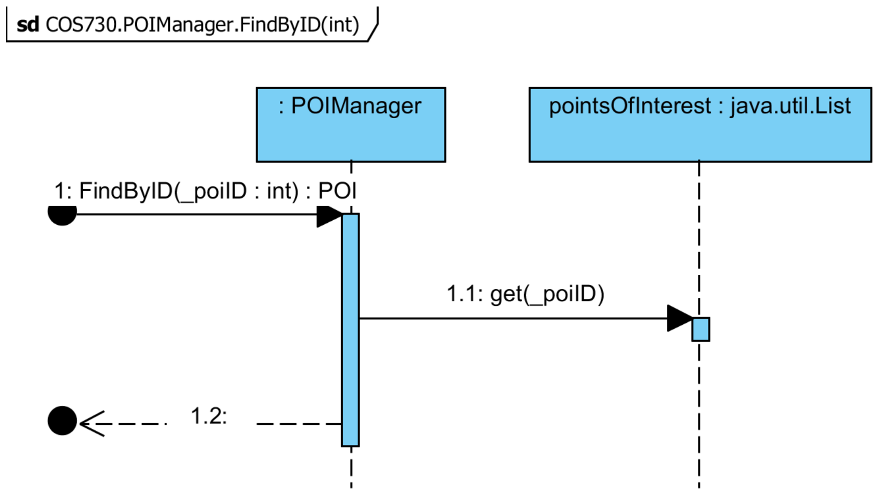

sd COS730.POIManager.FindByID(int)
: POIManager
pointsOfInterest : java.util.List
1: FindByID(_poiID : int) : POI
1.1: get(_poiID)
1.2:

Figure 8: Sequence Diagram for the FindByID function

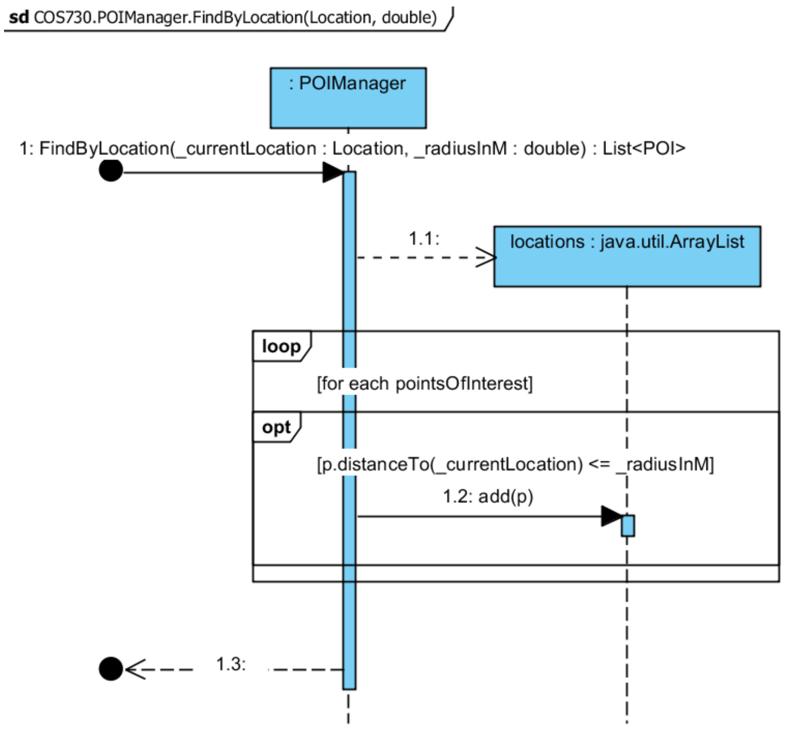
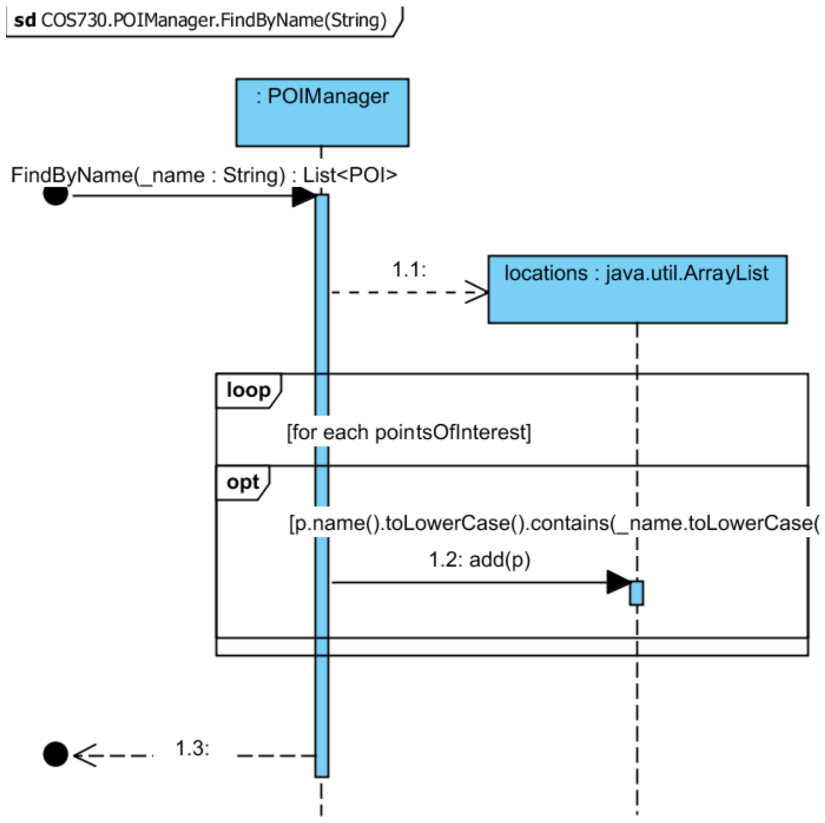

sd COS730.POIManager.FindByLocation(Location, double)
: POIManager
1: FindByLocation(_currentLocation : Location, _radiusInM : double) : List<POI>
1.1:
locations : java.util.ArrayList
loop
[for each pointsOfInterest]
opt
[p.distanceTo(_currentLocation) <= _radiusInM]
1.2: add(p)
1.3:

Figure 9: Sequence Diagram for the FindByLocation function



sd COS730.POIManager.FindByName(String)
: POIManager
FindByName(_name : String) : List<POI>
1.1:
locations : java.util.ArrayList
loop
[for each pointsOfInterest]
opt
[p.name().toLowerCase().contains(_name.toLowerCase(
1.2: add(p)
1.3:

Figure 10: Sequence Diagram for the FindByName function

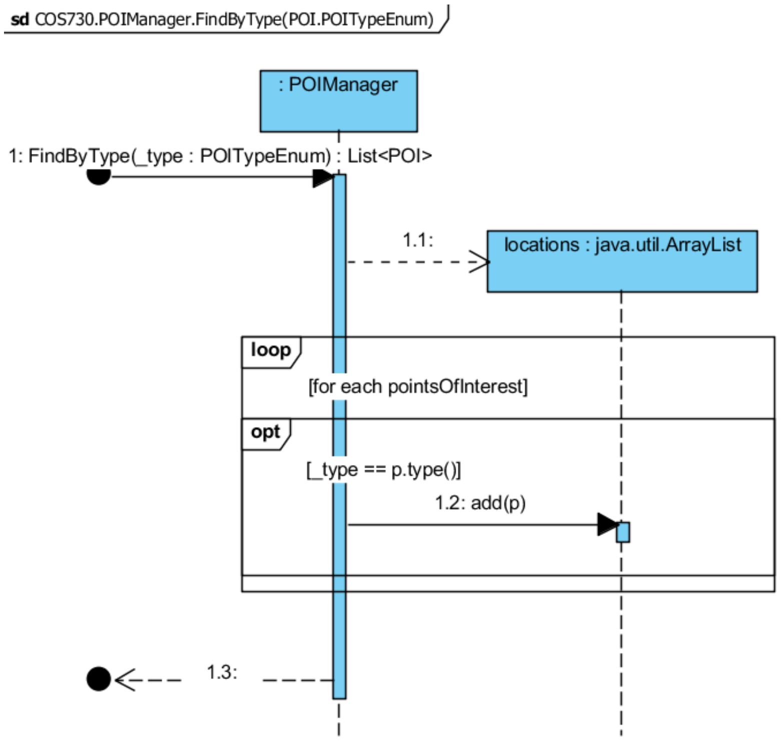

sd COS730.POIManager.FindByType(POI.POITypeEnum)
: POIManager
1: FindByType(_type : POITypeEnum) : List<POI>
1.1:
locations : java.util.ArrayList
loop
[for each pointsOfInterest]
opt
[_type == p.type()]
1.2: add(p)
1.3:

Figure 11: Sequence Diagram for the FindByType function

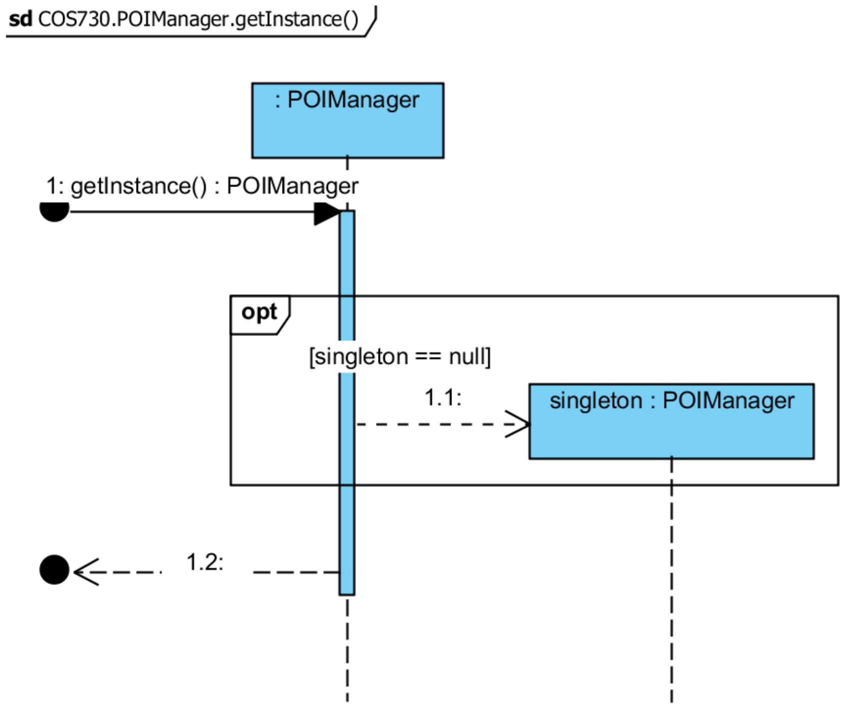


Figure 12: Sequence Diagram for the getInstance function

5 Events Module

5.1 UML Class Diagram

5.2 Use Case Diagram

5.3 Architectural Design of Module

The events module will be able to inform different users about events on campus. The module will make use of the location services of the navigation system, thus when a user passes a particular point of interest which has public events, the user will receive a notification via sms or email about these events.

The NavUP system will make use of a variety of design patterns. In the events module there will be an events superclass which abstracts the core fuctionality and members that an event object will provide. This means it will be possible to specify different events based on the type, i.e. a special lecture, exhibition, performance etc. In order to add functionality to these different types of events, the decorator design pattern will be used. A concrete decorator will be define to implemnt core funtionality defined by the abstarct event class, and the decorator class will hold a reference to an event object which to which functionality can be added dynamically.

A single event object will have 4 different attributes namely a name for the event, a date, a time, and a location which is a Point of Intereste object. Different subcalsses will have addiotional attributes, such as a special lecture having a lecturers attribute, performances having performers information, and exhibitions having a topic description. The decorator pattern will allow us to add additional functionalities to each subclass based on the additional attributes which enhances specificity.

Each concrete event class will also be equiped with a Serialization function which will return the string containing information about the event which will be used in the notifaction that is sent to the user, which in the end is the goal of the module.

5.4 Technologies Used

5.5 Non-Trivial Implementation Tasks

The events module will be accessed when the user is using the NavUP system to navigate the UP campus. The system will continuously get the current location of the student and when a student passes a point of interest the system will check for events at the particular location. These events will then be compiled into a notification that the user will receive either as an SMS, email, or both. Once a notification has been sent the system will return to tracking the users location if the user is still using the system for navigation.