UNIVERSITY OF ZAGREB
FACULTY OF ORGANIZATION AND INFORMATICS
VARAŽDIN

Marko Bartolić, mbartoli@foi.hr
Rene Škuljević, rskuljev@foi.hr
Tomislav Vunak, tvunak@foi.hr

NAVIGATION ON A SKI SLOPE

TECHNICAL DOCUMENTATION FOR SOFTWARE ANALYSIS AND DEVELOPMENT PROJECT

Varaždin, 2016.

UNIVERSITY OF ZAGREB
FACULTY OF ORGANIZATION AND INFORMATICS
VARAŽDIN

Team number: T10
Team members:
Marko Bartolić, 0016092146
Rene Škuljević, 0016092375
Tomislav Vunak, 0016091502

NAVIGATION ON A SKI SLOPE

TECHNICAL DOCUMENTATION FOR SOFTWARE ANALYSIS AND DEVELOPMENT PROJECT

Mentors:
Doc. dr. sc. Zlatko Stapić
Dr.Sc. Ivan Švogor
Evolaris mentors:
Gerald Binder
Thomas Rößler

Varaždin, 2016.

Table of contents

1 Introduction

1.1 Purpose of this document

The purpose of this document is to give a detailed description of the requirements for the navigation on a ski slope mobile software, called “EvoSki”. This document will explain the purpose and features of the application, interfaces of the application, what the application will do, the constraints under which it must operate, how the system will react to external stimuli and some basic guidelines for parties interested in continuing the development of this project and application. This document is intended for both the stakeholders and the developers of the application and will be proposed to the Evolaris company.

1.2 Intended Audience

The intended audiences are:

- Course mentors, to analyze the design and implementation of EvoSki app
- Evolaris mentors, to analyze the design and implementation of EvoSki app
- Authors of this document
- Eventual further developers

1.3 Scope

This project has two main problems to solve. One is to develop an algorithm that would notify the person skiing about the upcoming turns and in which direction he must turn and the second one is to notify him if deviates from track.

Software to be produced is called “EvoSki”. It is a mobile application, developed for devices running the Android operating system, specifically the Recon Snow2 ski goggles. This software is intended for use by skiers on the ski tracks. Using the integrated display in the goggles, this software will notify the person skiing in real time if he left the track and point him in which direction he should turn in the upcoming junctions. Information about tracks and turns can be obtained by accessing and using Evolaris' web server.

Main goal of the project is to develop the aforementioned algorithms so that Evolaris can continue their development and integration in real life systems.

2 General overview

This project is a part of course on the Faculty of Organization and Informatics which collaborates with the Evolaris company. Evolaris already has a similar application developed and they would like to add some new functionalities which we were tasked to try and develop. We were given technical documentation and sample data on their existing web server so we can develop the fore mentioned functionalities.

2.1 System characteristics

The application is installed on the mobile device which runs on Android operating system. Application runs in real time and uses GPS module of the device to locate the device, and thus the skier. Application allows the user to send the desired GPS coordinates (which represent the starting and ending point of the track) to the server which returns the path between that two points. Application notifies the user if he deviates away from the track and also signals him in which direction he should turn while approaching junctions.

2.2 System architecture

The *ws_plugin* module of the application is based on a client-server architecture. The client sends the desired track start and end coordinates along with some additional information to the server. The server responds by returning the path between these two points. The path is defined by series of points on the track which represent the junctions on the track. This communication is achieved by using the Retrofit library, which is a REST type client for Android, and in this application is built in a modular way. In this application Retrofit is a part of Model-View-Presenter architecture. Communication with server is described later in this document.

2.3 Infrastructure services

Application currently has error handling in the type of displaying messages if something unexpected happens. We made sure that the application does not crash, but instead just notifies the user what problem happened.

As for the other services, such as security, logging, etc., this application currently has none, because it is not defined as a vital part of the project, nor there was real need for them.

3 Application usage

This section shows and explains the functionalities of the application.

EvoSki application has the following functionalities:

- Obtaining user GPS coordinates

  Application uses GPS to determine current location of the device. This is needed for positioning the skier on the ski track. Application needs access to device GPS module and Location permission.

- Getting track coordinates from web server

  Application uses HTTP to communicate with the web server in order to fetch track points. The application needs access to the internet.

- Getting track coordinates locally

  Application can access and read the local file on the device in order to get track coordinates. Application needs Storage permission. See section 3.4 for details.

- Choice between three algorithms for smoothing the skier's movement

  User can choose between "Real points", "Smooth point algorithm" and "Moving average algorithm" in the Options menu. Algorithms are described in section 5.3.

- Recognizing where user should turn next and how sharp the turn is

  Application can use information about turning points received from web server (section 3.5) or information defined locally (section 3.3 and 3.4) and accordingly point in which direction user should turn.

- Showing the distance to the next turn and the distance to the end of the track

  Application calculates and displays distance from the device to the next turn and remaining distance to track end.

- Shows the remaining number of turns to the track end

  Application calculates and displays the number of remaining turns to the track end.

- Notifying the user if he deviates from track

  Application approximately calculates in real time if the skier is on the track and if he is not, then it notifies him that he left the track. This algorithm is explained in section 5.3.

As user starts the application, he can currently choose between three types of track options. Each track option represents a different type of getting the track information. Upon selecting any of these three options, the application calculates and shows the distance between the current user's location and the next turn on track.

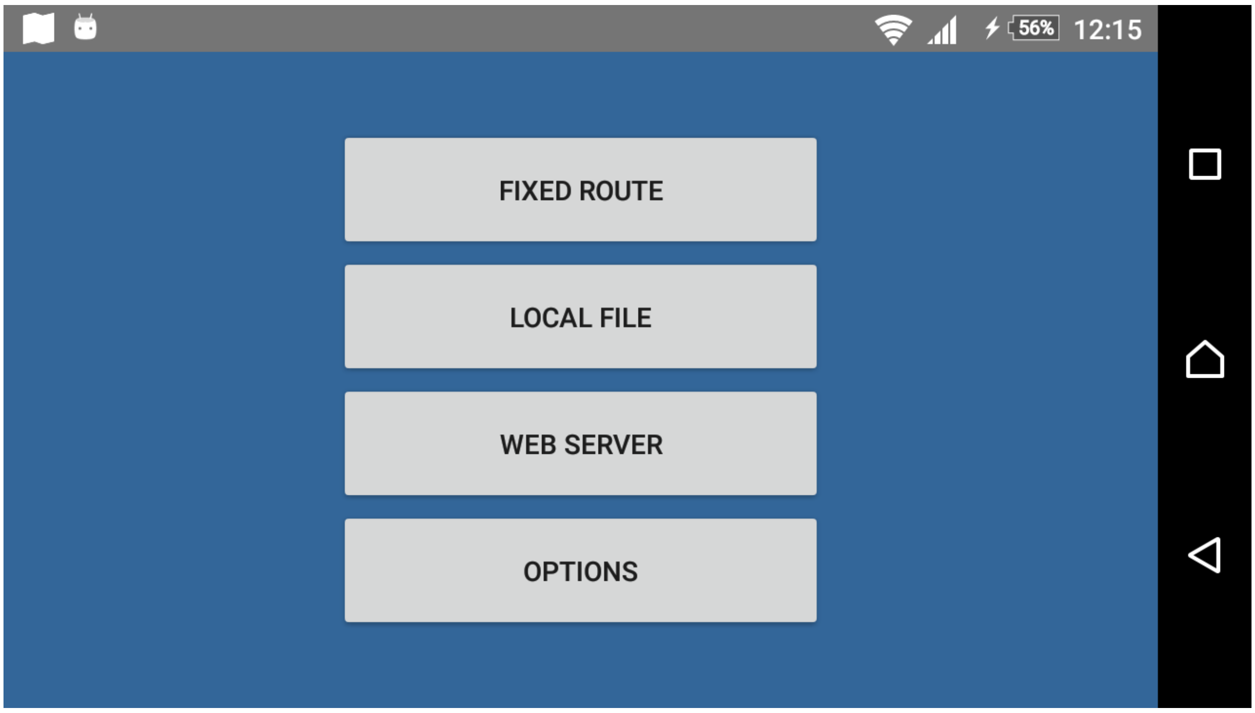


Figure 1. Application main menu

3.1 System architecture

System architecture that supports the EvoSki application consists of several elements. Communication between these elements is shown in Figure 2.

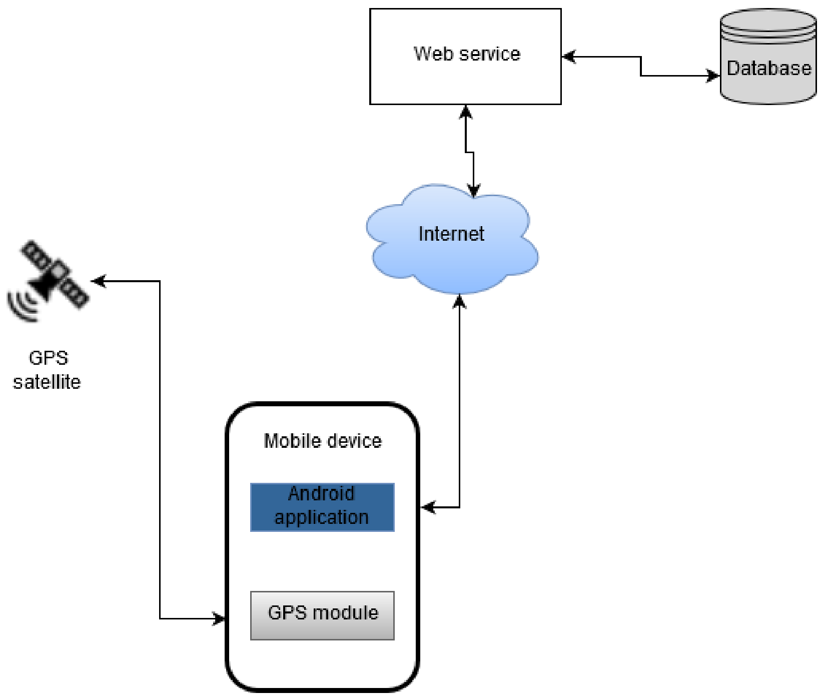


Figure 2. System architecture

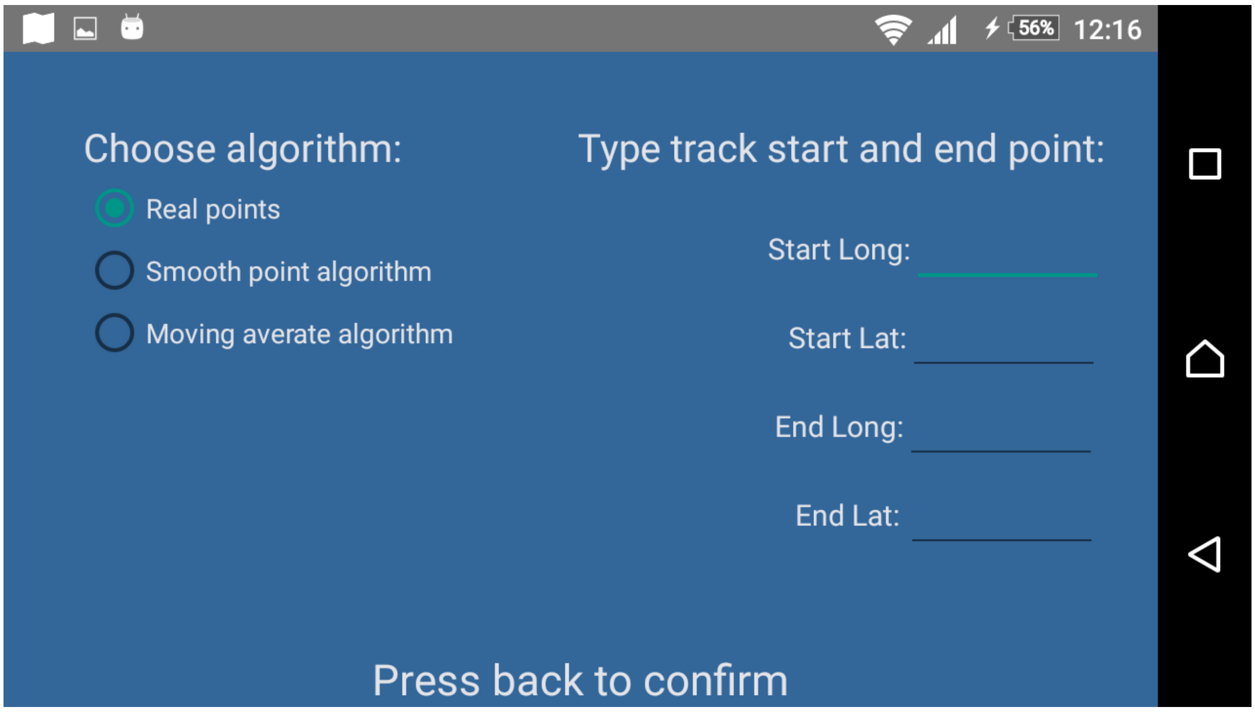
Communication between architectural components:

1. Application requests GPS data from the GPS satellites.
2. Application connects to the internet and requests track data from the Evolaris' web service.
3. Web service fetches track data from the database and sends it to the application.
4. Application also reads txt file from the device, if that option is chosen.

3.2 Options menu

The "Options" menu allows the user to select one of three smoothing algorithms which will be used in the track calculations – "Real points", "Moving average" and "Smooth point". Algorithms are described in section 5.3.

Through this menu, user can also type in desired track start and end coordinates which are stored by pressing the "back" button, and if the "Web server" option in the main menu is selected, that coordinates are sent to the server. If neither option is selected, "Real points" remains as the default one.


12:16

56%

Choose algorithm:

[✓ Real points]

Smooth point algorithm

Moving averate algorithm

Type track start and end point:

Start Long:

Start Lat:

End Long:

End Lat:

Press back to confirm

Figure 3. In-app options menu

3.3 Fixed route

"Fixed route" is a small track that goes through the streets of Varaždin. Coordinates and turns of this track are hardcoded in the source code of the application and that track was primarily used for testing purposes.

3.4 Local file

"Local file" is an option for loading track coordinates and turns from a txt file directly from the device. Currently, file name must be "TrackRoute.txt", and that file must be stored in default "Documents" folder of the device (root\Documents).

Contents of the file must be as follows:

Longitude
Latitude
number which represents sign, i.e. sharpness of the turn (any **positive** values indicates **right turn**, and any **negative** value indicates **left turn**, while 0 indicates no turns e.g. -1 is left turn, 1 is right turn, 0 is no turn)

Example of text file:

```
46.308044    //- Longitude p1
16.345183    //- Latitude p1
0            //- sign p1
46.307642    //- Longitude p2
16.343479    //- Latitude p2
2            //- sign p2
46.308439    //- Longitude p3
16.342981    //- Latitude p3
-2           //- sign p3
```

That represents one point on track. Each of that information must be written in next line in file. Further points can be added right in the next line after that, by following the same rules. Obviously, if the file doesn’t exist, or the records inside are improperly defined, this option would not work.

3.5 Web server

“Web server” is an option that allows the user to send track start and end points to the Evolaris’ web server, which then returns the whole track between these two points. This communication is achieved through API provided by Evolaris. Application uses Retrofit to communicate with the web service which sends retrieved data from the database back to the application. Next, that response is translated from the JSON object to the Java classes which the application uses to show the user his current GPS coordinates and remaining turns to the track finish.

In order to use this option, the “Options” dialog on the main menu must be opened first. There the user can type the GPS coordinates of track start and end points. By pressing the “back” button, these coordinates are saved and by pressing “Web server” that coordinates are sent to the Evolaris’ web server which returns the whole track between that defined points. For this option to work, device has to have some type of internet connection and start and end coordinates of the track must be typed inside the “Options” screen.

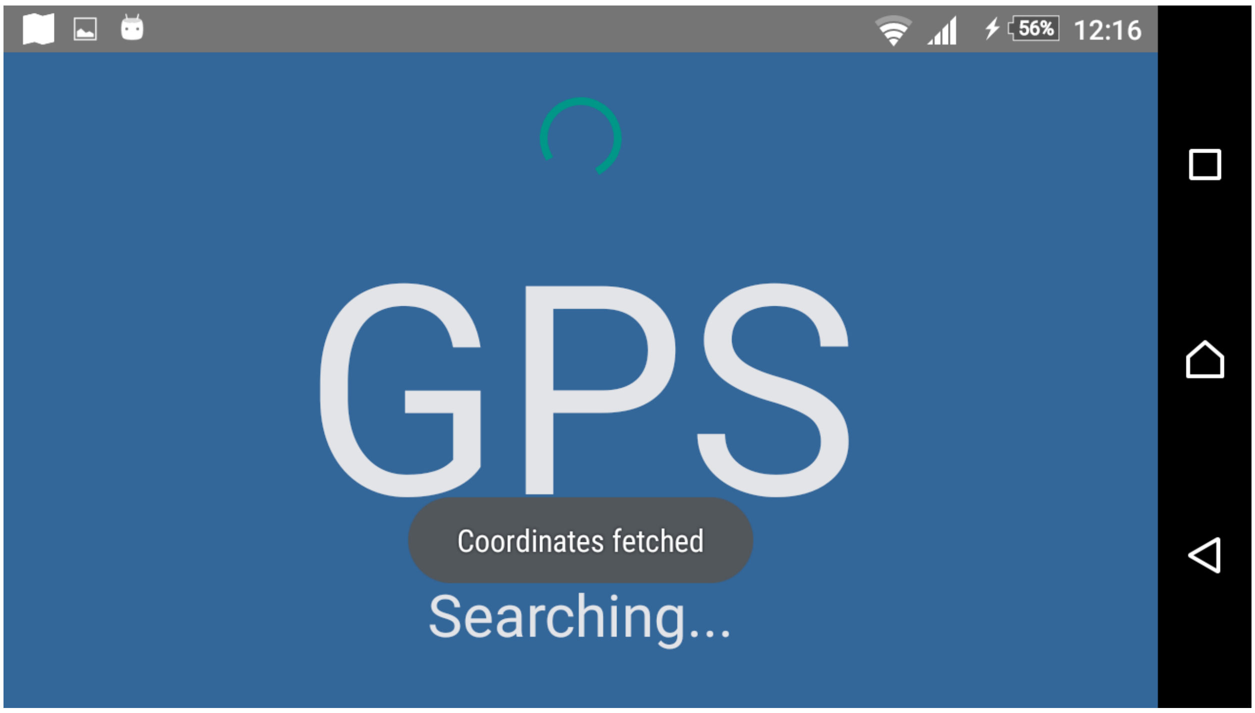


Figure 4. Coordinates fetched from web server

3.6 Working with Lockito for mock locations

Lockito is an free android application developed by Damien Villeneuve and it is available for download through Google Play Store. This is a GPS mock location application that “allows you to make your phone follow a fake itinerary, with total control over the speed and GPS signal accuracy.” This application was used to trick the device into sending the “fake” GPS locations to the EvoSki application, in order to make development and testing easier.

In order to use this application, follow this steps:

1. Download and install the application from the Google Play Store (https://play.google.com/store/apps/details?id=fr.dvilleneuve.lockito).
2. On the device, open *Settings* → *Developer options* → *Mock locations* and set Lockito as default mock location application.
3. Start the application and grant all permissions.

4. Pressing the “plus” sign, add a new itinerary and name it however you like.
5. On the map, long click on the location you wish to set as a starting point of the route until marker is shown on map (this and following step require access to the internet).
6. Then long click on the location you wish to set as ending point of the route until marker is shown on map and route is drawn map.
7. Options at the top of the screen can be modified to set desired speed of movement between points and accuracy of the “fake” GPS signal.
8. After everything is set as desired, press End and then Save button.
9. Upon selecting desired track, it can be started by pressing the Play button, or if the application is sent to background, selected track can be started by pulling down the top menu on the device.
10. Lockito then simulates the movement between selected points and tricks the device by sending “fake” GPS signal.

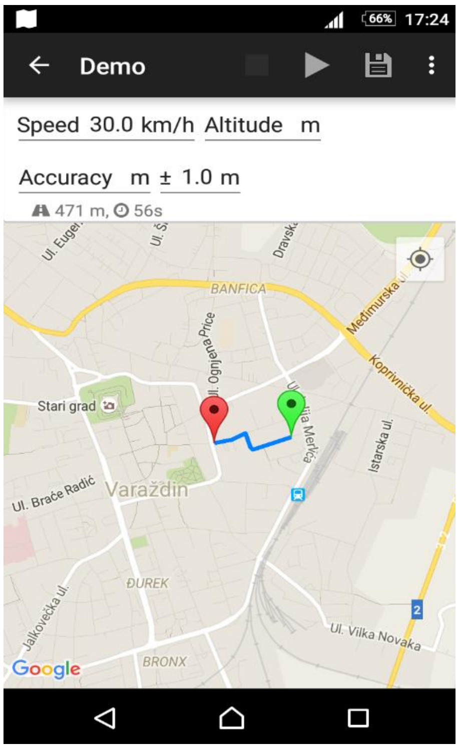


Figure 5. Lockito demo

3.7 Working with application

Application detects and monitors user’s movement pattern and locations. As he approaches the turn, i.e., when he is 30m away from the turn, the application shows the appropriate arrow which directs the user where he should turn, depending on user's position in relation to turning point and the track itself. In order for this to work properly, the skier should be skiing in a default left-right skiing pattern. This is important because the algorithm that detects skier's movement relies on this assumption.

At the same time, the application monitors user's position and if he is more than 100m away from the nearest point on track for every call of the "onLocationChanged" method five times in a row, the application notifies the user that he deviated too far away from the track.

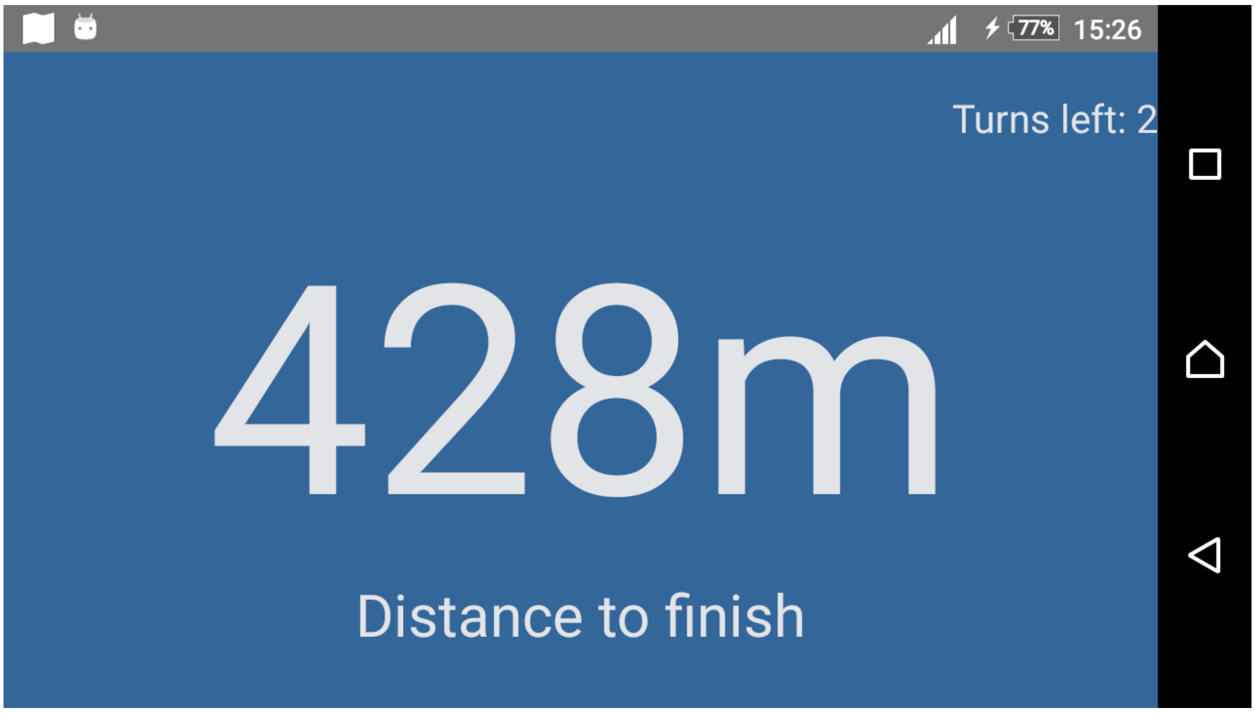


Figure 6. Application shows remaining distance to finish and number of turns left

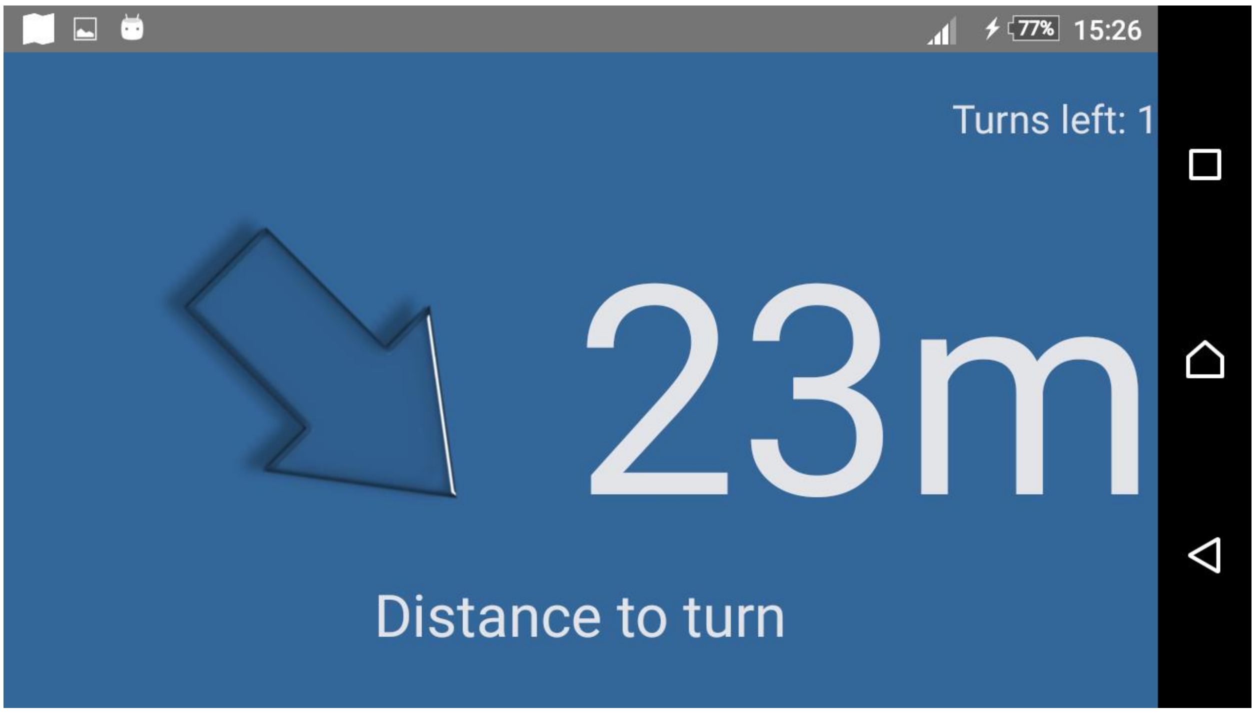


Figure 7. Application shows distance to the next turn, when that distance is less than 50 m

4 System context

This section describes the external interface, i.e., communication with the Evolaris' web server. Evolaris provided us with access to their web server and data which we used for getting information about tracks. Application sends information about start and finish points through API as a HTTP request to web service. Thus, the application and the device which it is installed on needs some kind of connection to the internet. The server calculates navigation from start to finish points and returns data in JSON format. The speed of that information flow depends mainly on the speed of internet connection available, but there shouldn't be any problems or major delays, since this communication doesn't use a lot of data. It is important to note that the server doesn't recognize road curves as turns on the track, but only intersections. Therefore, all the curves on the road are ignored and as such cannot be used as turns for a ski track.

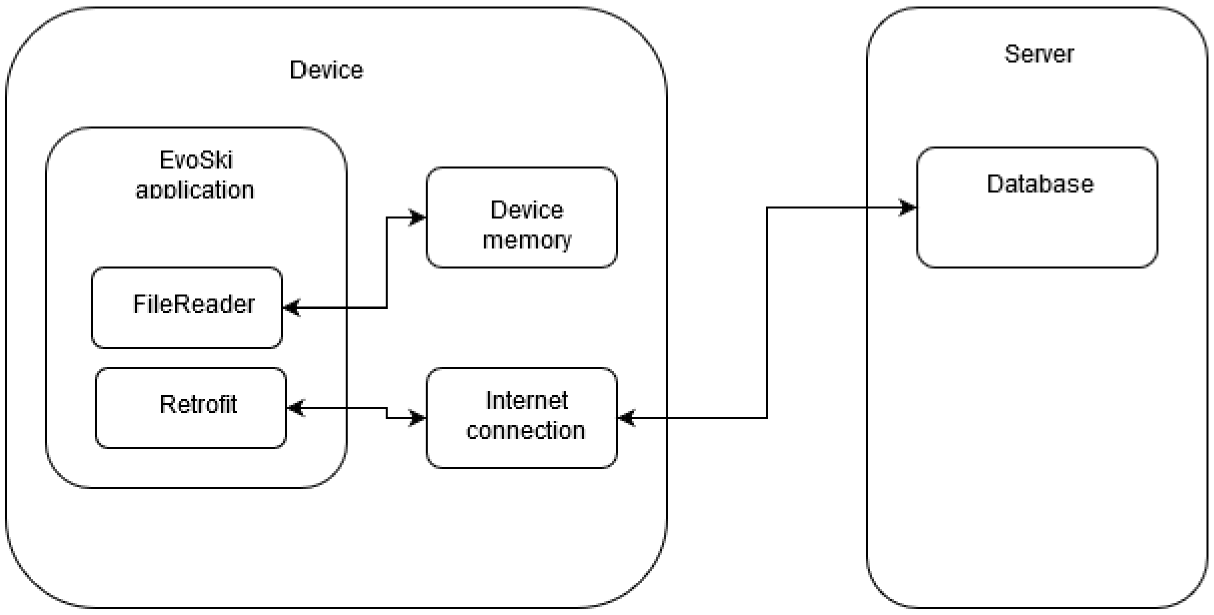


Figure 8. Modules diagram

4.1 Request to server

Basic URL for routing is:

http://routing2.maptoolkit.net/route?

The following HTTP parameters are needed:

Name	Description
point	Specify at least two points for which the route should be calculated. The order is important – latitude first, longitude second.
routeType	The vehicle for which the route should be calculated – **car**, **bike** or **foot**
voice_instructions	This option enables voice instructions for text-to-speech engines. 0 = disabled 1 = enabled
language	Language of the instructions in ISO 639-1 codes (two letters).

Figure 9. Parameters for HTTP request

Request example:

http://routing2.maptoolkit.net/route?point=46.308044,16.345183&point=46.307944,16.340941&routeType=car&voice_instructions=0&language=en

4.2 Server response

Response is returned in JSON format. The following table lists all the output that the server returns.

Name	Description
info.took	How many ms the request took on the server, without network latency taken into account.
paths	An array of possible paths.
paths[0].distance	The overall distance of the route, in meters.
paths[0].time	The overall time of the route, in ms.
paths[0].points	The polyline encoded coordinates of the path. Order is lat,lon,elelevation as it is no geoJson.
paths[0].points_encoded	Is true if the points are encoded, if not paths[0].points contains the geo json of the path (then order is lon, lat, elevation).
paths[0].bbox	The bounding box of the route, format: minLon, minLat, maxLon, maxLat
paths[0].instructions	Contains information about the instructions for this route. The last instruction is always the Finish instruction and takes 0 ms and 0 meter.
paths[0].instructions[0].text	A description what the user has to do in order to

	follow the route. The language depends on the locale parameter.
paths[0].instructions[0].coordinate	The first coordinate of the interval.
paths[0].instructions[0].distance	The distance for this instruction, in meters.
paths[0].instructions[0].time	The duration for this instruction, in ms.
paths[0].instructions[0].interval	An array containing the first and the last index (relative to paths[0].points) of the points for this instruction.

paths[0].instructions[0].sign	A number which specifies the sign to show e.g. for right turn etc:
	TURN_SHARP_LEFT = -3
	TURN_LEFT = -2
	TURN_SLIGHT_LEFT = -1
	CONTINUE_ON_STREET = 0
	TURN_SLIGHT_RIGHT = 1
	TURN_RIGHT = 2
	TURN_SHARP_RIGHT = 3
	FINISH = 4
	VIA_REACHED = 5
	USE_ROUNDABOUT = 6 paths[0].

paths[0].voice_instruction	[optional] Only available the the request parameter voice_instructions is defined. Contains information about the voice instructions for this route.
paths[0].instructions[0…n].coordinate	The coordinate for the voice instruction.
paths[0].instructions[0…n].text	A description what the user has to do in order to follow the route. The language depends on the locale parameter.

Figure 10. Server response parameters

If there is no internet connection available, the application will notify the user about that and therefore the communication with the server cannot be established. Below you can see an example of server response.

```
{
  "info": {
    "copyrights": [
      "Toursprung",
      "OpenStreetMap contributors"
    ],
    "took": 5
  },
  "paths": [
    {
      "distance": 397.488,
      "weight": 397.488056,
      "descend": 1.8329925537109375,
      "ascend": 3.1659927368164062,
      "points_encoded": true,
      "points": "ipsyGilwbBv@zENdAAXMNuB~@OFZhBXlBP~@^`D",
      "bbox": [
        16.340947,
        46.307695,
        16.345178,
        46.308444
      ],
      "time": 39101,
      "instructions": [
        {
          "distance": 218.402,
          "name": "Slavenska",
          "text": "Continue onto <strong>Slavenska</strong>",
          "interval": [
            0,
            6
          ],
          "voice_instructions": [
            {
              "coordinate": [
                46.30805,
                16.34517
              ],
              "text": "Continue onto <strong>Slavenska</strong> and in 220 meter turn left onto <strong>Petra KreĹˇimira IV, 25058</strong>"
            }
          ],
          "sign": 0,
          "time": 26208,
          "coordinate": [
            46.30844,
            16.34315
          ],
          "speed": 30
        },
        {
          "distance": 179.086,
          "name": "Petra KreĹˇimira IV, 25058",
          "text": "Turn left onto <strong>Petra KreĹˇimira IV, 25058</strong>",
          "interval": [
            6,
            10
          ],
          "voice_instructions": [
            {
              "coordinate": [
                46.30769000440957,
                16.343719942675573
              ],
              "text": "In 100 meter turn left onto <strong>Petra KreĹˇimira IV, 25058</strong>"
            },
            {
              "coordinate": [
                46.30844,
                16.34315
              ],
              "text": "Turn left onto <strong>Petra KreĹˇimira IV, 25058</strong>"
            }
          ],
          "sign": -2,
          "time": 12893,
          "coordinate": [
            46.30792,
            16.34094
          ],
80.                 "speed": 50
81.             }
82.         ]
83.     }
84.   ]
85. }
```

Figure 11. Server response

Information about turns received from server can be seen on figure 11. Specifically, command block “paths” contains the entire route. Inside there is “instructions “command block which contains all the turning points on the track, along with their coordinates. In order for the application to work, the information it needs are just coordinates of the turn and sign (sharpness of the turn). Each turning point is contained inside curly brackets. In this example there are two turns. First one starts at 25. line of code and ends at 49. line of code, while the other turn starts at 50. line of code and ends at 81. line of code. Each turn also contains additional information like time, speed and street names, which are not relevant to this project.

One thing to note is that server for some reason returns one point before defined end point, and that point always has sign value -2, meaning turn left.

5 System design

This section explains the internal logic of the application. This project was done using the object oriented design approach, in order to isolate different layers that make the overall architecture of the application. Using the MVP (model-view-presenter) paradigm we aimed to create one module (*ws_plugin*) which would be easily separated and integrated back into the application if necessary. This also enables easy integration of any additional modules into the application if desired.

5.1 Technologies used

In order for any further development of this project to be possible, this section lists all the technologies and software used in the development process so far.

Modelling tools

- Visual Paradigm for activity diagrams

Version control system:

- Github repository available at:
https://github.com/rskuljev/Navigation-on-a-ski-slope/

Web service and database:

- Used for testing purposes: FileZilla Client and MySQL database
- Used in application: web service given to us by Evolaris

Additional libraries and SDK:

- Retrofit
- Recon SDK

Application development tools:

- Android Studio
- Genymotion
- Microsoft Visual Studio 2010 (used for algorithm development)

GPS mock locations app

- Lockito

5.2 Classes

The following table lists classes used in the development process of this project and their description. Classes defined in *ws_plugin*'s mvp package are part of the MVP architecture and are not described below.

Class name	**Description**
app module	**Main module of the application.**
FileCoordinates	Class which is using LocalFileRead class.
FixedCoordinates	Stored fixed track points.
AverageDirection	Algorithm for calculating skier's average direction.
ConvertingGpsCoordToXY	Class for converting GPS coordinates to X and Y coordinates which algorithms use.
DistanceFromPoint	Calculating distance between two given points.
LocalFileRead	Class for accessing and reading local txt file stored on mobile device.
MainActivity	Main class for navigation through application.
MyLocationGPS	Class which is processing information when user is moving and collecting GPS points.
Options	Class for setting different options for application.
SplashScreen	Class called on application startup which shows application intro.
UserLocationStatus	Calculating skier's position on track which depends on track points.
AlgorithmsInterface	Interface for using different algorithms.
MovingAverage	Algorithm which is using statistic methods for smoothing GPS points.
RealPointsAlgorithm	Algorithm which returns real points without smoothing.
SmoothingAlgorithm	Algorithm developed for GPS points smoothing.
core module	**Module that other two use for preferences and fetching.**
FetchTrackCoordinatesInterface	Interface for fetching track coordinates.
FetchTrackCoordinatesListener	Listener for fetching track coordinates.
MyTrackPoints	Class which defines main data type used in application.
PreferenceManagerHelper	Class for getting preference data.
ws_plugin module	**Module for communication with web server.**
Coordinates	Coordinates model.
Instruction	Instruction model.
CoordiantesInteractorImpl	Class which implements Retrofit to fetch information from the web server.

CoordiantesInteractor	Interface for interaction with web server coordinates.
RequestAPI	API for request to the web server.
WebServerCoordinates	Requests and stores information from web server.

Figure 12. List of classes and their descriptions

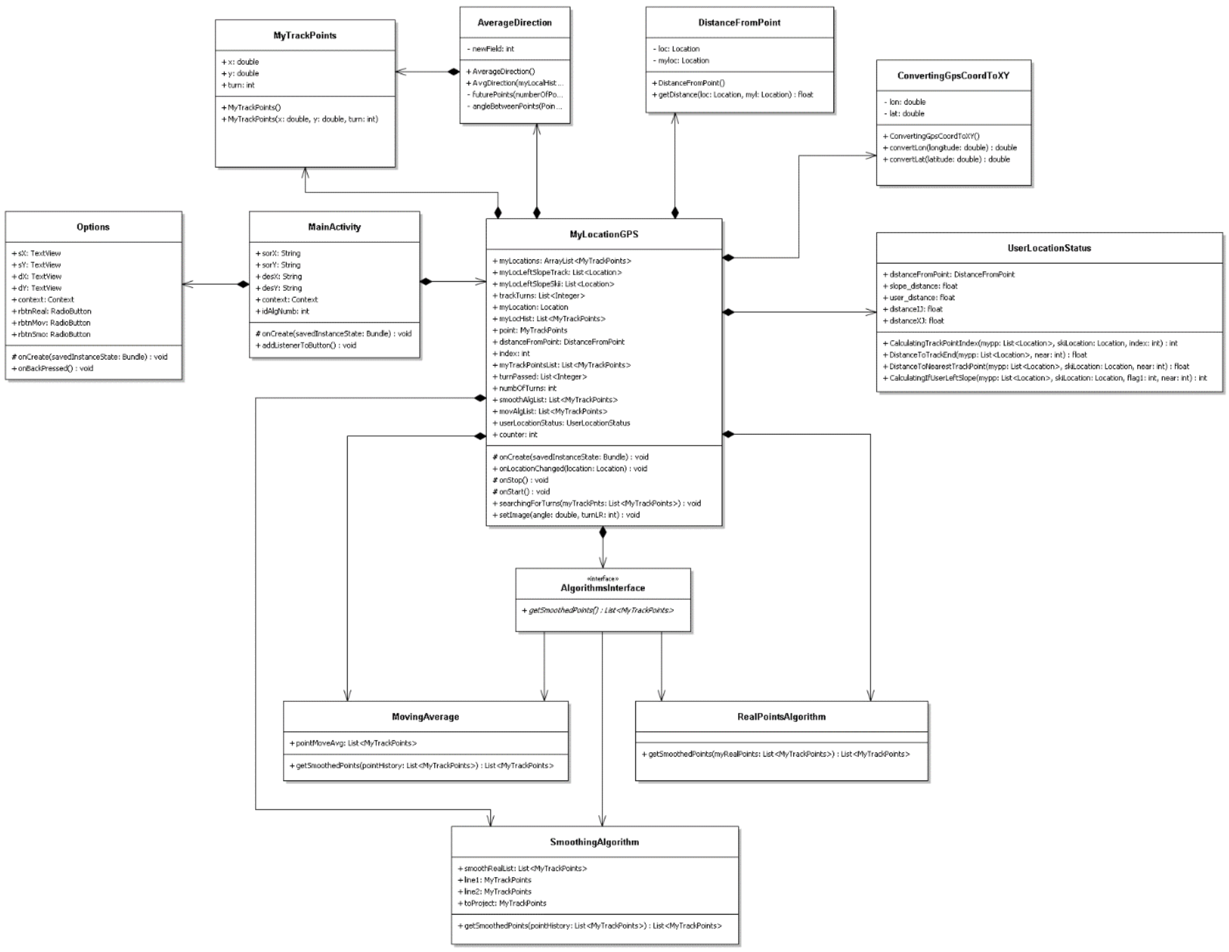

MyTrackPoints
+x: double
+y: double
+turn: int
+MyTrackPoints()
+MyTrackPoints(x: double, y: double, turn: int)
AverageDirection
- newField: int
+AverageDirection()
+AvgDirection(myLocalHist...
- futurePoints(numberOfPo...
- angleBetweenPoints(Poin...
DistanceFromPoint
- loc: Location
- myloc: Location
+DistanceFromPoint()
+getDistance(loc: Location, myl: Location) : float
ConvertingGpsCoordToXY
- lon: double
- lat: double
+ConvertingGpsCoordToXY()
+convertLon(longitude: double) : double
+convertLat(latitude: double) : double
Options
+sX: TextView
+sY: TextView
+dX: TextView
+dY: TextView
+context: Context
+rbtnReal: RadioButton
+rbtnMov: RadioButton
+rbtnSmo: RadioButton
#onCreate(savedInstanceState: Bundle) : void
+onBackPressed() : void
MainActivity
+sorX: String
+sorY: String
+desX: String
+desY: String
+context: Context
+idAlgNumb: int
#onCreate(savedInstanceState: Bundle) : void
+addListenerToButton() : void
MyLocationGPS
+myLocations: ArrayList<MyTrackPoints>
+myLocLeftSlopeTrack: List<Location>
+myLocLeftSlopeSkii: List<Location>
+trackTurns: List<Integer>
+myLocation: Location
+myLocHist: List<MyTrackPoints>
+point: MyTrackPoints
+distanceFromPoint: DistanceFromPoint
+index: int
+myTrackPointsList: List<MyTrackPoints>
+turnPassed: List<Integer>
+numbOfTurns: int
+smoothAlgList: List<MyTrackPoints>
+movAlgList: List<MyTrackPoints>
+userLocationStatus: UserLocationStatus
+counter: int
#onCreate(savedInstanceState: Bundle) : void
+onLocationChanged(location: Location) : void
#onStop() : void
#onStart() : void
+searchingForTurns(myTrackPnts: List<MyTrackPoints>) : void
+setImage(angle: double, turnLR: int) : void
UserLocationStatus
+distanceFromPoint: DistanceFromPoint
+slope_distance: float
+user_distance: float
+distanceIJ: float
+distanceXJ: float
+CalculatingTrackPointIndex(mypp: List<Location>, skiLocation: Location, index: int) : int
+DistanceToTrackEnd(mypp: List<Location>, near: int) : float
+DistanceToNearestTrackPoint(mypp: List<Location>, skiLocation: Location, near: int) : float
+CalculatingIfUserLeftSlope(mypp: List<Location>, skiLocation: Location, flag1: int, near: int) : int
«interface»
AlgorithmsInterface
+getSmoothedPoints() : List<MyTrackPoints>
MovingAverage
+pointMoveAvg: List<MyTrackPoints>
+getSmoothedPoints(pointHistory: List<MyTrackPoints>) : List<MyTrackPoints>
RealPointsAlgorithm
+getSmoothedPoints(myRealPoints: List<MyTrackPoints>) : List<MyTrackPoints>
SmoothingAlgorithm
+smoothRealList: List<MyTrackPoints>
+line1: MyTrackPoints
+line2: MyTrackPoints
+toProject: MyTrackPoints
+getSmoothedPoints(pointHistory: List<MyTrackPoints>) : List<MyTrackPoints>

Figure 13. Class diagram (algorithms)

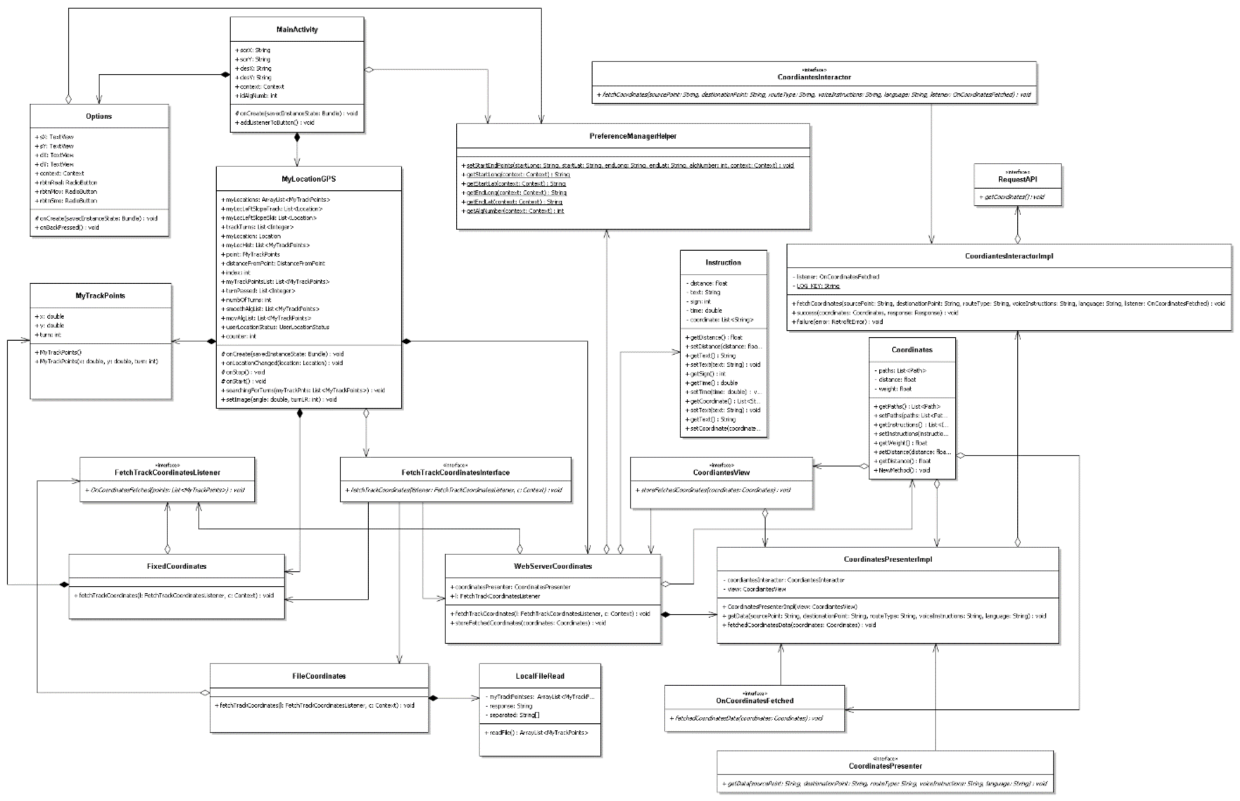


Figure 14. Class diagram (fetching coordinates)

5.3 Algorithms

This section explains each algorithm used in application.

5.3.1 Real points algorithm

This is actually not an algorithm and it doesn’t do any operations on the data it receives. What it does is that this option allows passing original and unmodified data to other classes. That way it is possible to see what the actual, non-smoothed track route looks like.

5.3.2 Moving average algorithm

This algorithm implements the statistical calculation called moving average. In order for this calculation to work, it needs at least three sequential points. So in order to calculate one smoothed point, the second point must be doubled and added to the first and third point. This addition is then divided by four. Next point is calculated in the same way, except the first (original) point is removed and the next upcoming point is included in the calculation. Perhaps a better explanation is given by Wikipedia.org:

> “Given a series of numbers and a fixed subset size, the first element of the moving average is obtained by taking the average of the initial fixed subset of the number series. Then the subset is modified by "shifting forward"; that is, excluding the first number of the series and including the next number following the original subset in the series. This creates a new subset of numbers, which is averaged. This process is repeated over the entire data series.”

Moving average. Source: https://en.wikipedia.org/wiki/Moving_average

Example for calculating latitude:

$$x_p = \frac{x_1 + 2x_2 + x_3}{4}$$

Note: longitude is calculated in the same way, except the X is replaced with Y.

5.3.3 Smooth point algorithm

This algorithm works in a way that it “smooths” GPS points which represent locations of the skier. Algorithm looks at three sequential GPS points of the skier and it connects first and third point and makes an imaginary vector between them. The second point (which is between first and third one) is moved and put on that vector. Next step is to ignore the first point and take one more so that again there are three points. Now it takes that moved point (which is now considered first) and the next two sequential points and again connects this moved point (now first) and the third point (which is newly added). After that, the middle point is again moved and put on the imaginary vector between first and third point. This process repeats until there are no more points to be added.

In this way, the imperfections of GPS signal and non-perfect skiing pattern are somewhat ignored and it is possible to predict in which direction the skier is going and at what angle he will enter the turn i.e. junction.

5.3.4 Average direction

This is the first algorithm in the "AverageDirection" class. As its name says, it calculates the average direction in which the skier is moving. It works with extreme left and extreme right points of the skier's movement. So, when the user is skiing, he makes zig-zag patterns (goes left-right). That extreme left and right points are collected and stored in an array. Average point is calculated by adding two extreme left points and dividing that addition by two and adding two extreme right points and also dividing that addition by two. This process is repeated until there are no more extreme points left.

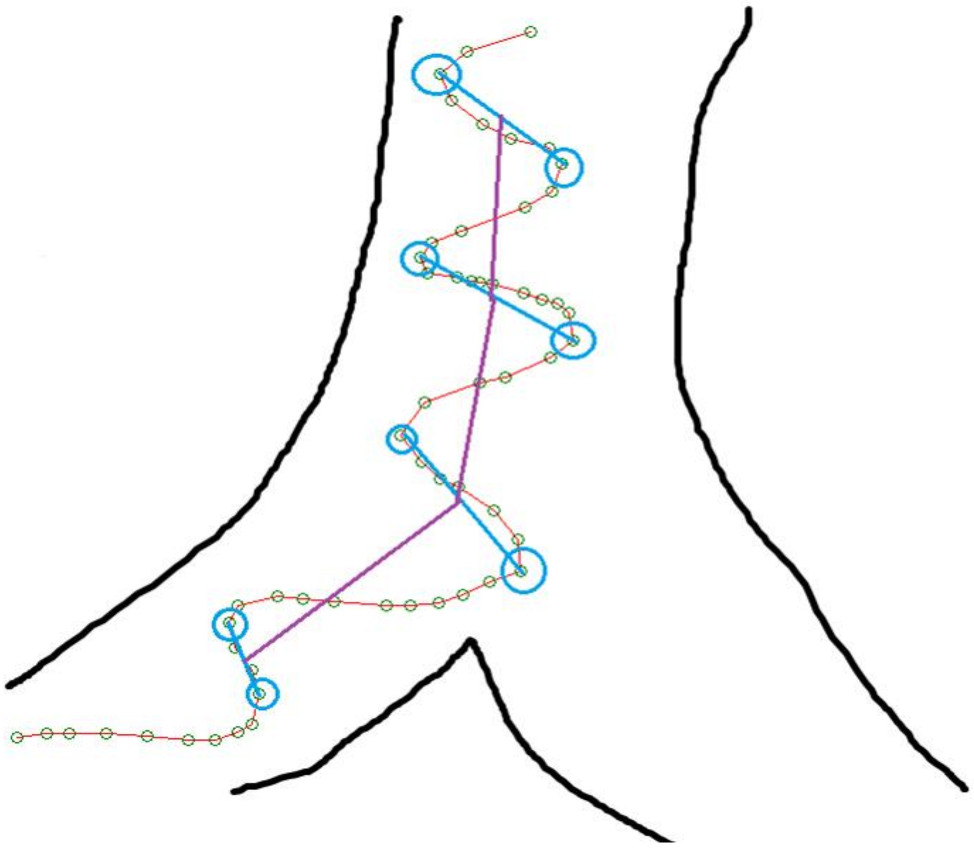


Figure 15. Visualization of Average Direction algorithm

5.3.5 **Future points**

This algorithm is located in the AverageDirection class. It works in a way that it subtracts the second to last point on track and point before that. This is done separately for longitude and latitude. After that, “predicted” future point of the skier can be calculated by adding that second to last point and that newly calculated one.

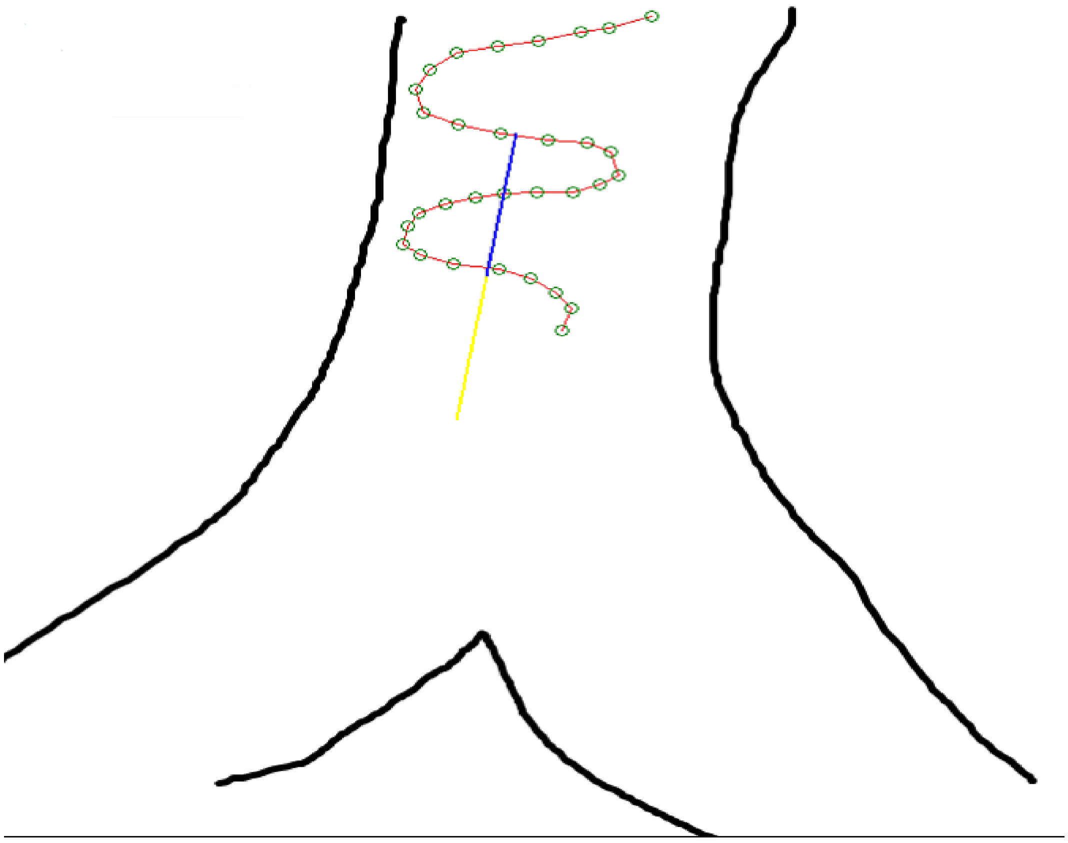


Figure 16. Visualization of Future Points algorithm

5.3.6 **Angle between points**

Angle between points is the third and final algorithm in the AverageDirection class. This algorithm calculates angle between points on junction. First it calculates the vector from the skier’s position to the turning point on track. Then it calculates the vector from that turning point to the next point on track (returned from server). These two vectors form an angle, which is calculated and it represents the angle of entrance in the junction.

5.3.7 **Check if skier left track**

Since width of the ski track is not known and is variable from track to track, the following algorithm works somewhat well under some restraints. The algorithm works in a way that it compares the distance between two sequential points on track and the user's distance to the next point. If the user's distance to the next point is greater than the distance between these two sequential points, then application notifies the user that he is off the track. So, the ideal scenario would be if the server returned track points which have the same distance between them as is the width of the ski track. Clearly this is not an ideal solution for the problem, but considering there is no way to precisely know the width of the track, this solution works relatively good. If the server returns points which are distant one from the other, like a road that doesn't have intersection for a while, the application could notify the user that he is off track when he may not be, thus possibly confusing him.

5.4 Requirements

All requirements are defined below and are rated either Mandatory (M) or Highly Desirable (HD) or Desirable (D), dependent on business need and University Policy.

5.4.1 **Functional Requirements**

5.4.1..1 Common Features

Requirement	**Preference**
1.1.1.1. User can see the sharpness of the turn	M
1.1.1.2 User is notified when he leaves the ski track	HD
1.1.1.3 User is shown picture of turn when he gets near turn	D
1.1.1.4 User can set his own routes via txt file	D
1.1.1.5 User can send track start and end (GPS coordinates) to the web server,	D

which return the track between these two points	
1.1.1.6 Application shows the number of turns left to track end	D

5.4.1..2 Reporting

Requirement	**Preference**
1.1.1.7 Project documentation	M
1.1.1.8 Notes from SCRUM meetings	D

5.4.2 **Production Requirements**

5.4.2..1 Hardware

Requirement	**Preference**
1.1.1.9 Mobile device	M
1.1.1.10 Recon Snow2 goggles	D

5.4.2..2 Software

Requirement	**Preference**
1.1.1.11 Android min API 16	M
1.1.1.12 Lockito – android app for mock GPS locations	D

5.4.3 **Development Requirements**

5.4.3..1 Hardware

Requirement	**Preference**
1.1.1.13 Mobile device	M

1.1.1.14 Recon snow2	D

5.4.3..2 Software

Requirement	**Preference**
1.1.1.15 Android Studio	M
1.1.1.16 Genymotion	D
1.1.1.17 Microsoft Visual Studio	D
1.1.1.18 Evolaris Web Service	M
1.1.1.19 Retrofit library	D