Advanced API for FoxyTag

Bachelor work carried out in order to obtain the Bachelor of Science in Information Systems

by :

Loïc POISOT

Bachelor work advisers :

Peter DAEHNE, HES professor

Michel DERIAZ, supervisor

Carouge, 8 June 2012

Geneva School of business Administration (HEG-GE)

Business Computer Sciences

Declaration

This Bachelor work is done in the context of the final exam of the Geneva school of business administration, in order to obtain an HES bachelor degree in Business computer sciences. The student accepts, if applicable, the privacy statement. The use of the conclusions and recommendations written down in this report, without foreseeing their value, do not commit either the responsibility of the author, or that of the Bachelor work adviser, the member of the jury and the Geneva school of business administration.

« I certify having worked alone for the present work, without having used other sources than those quoted into the bibliography. »

Done in Carouge the 8 June 2012

Loïc POISOT

My thanks

First of all I would like to thank Michel Deriaz for the provided bachelor subject, and more importantly, for the opening spirit he had about my propositions, and the good appreciation of my remarks.

I would also like to thank Peter Daehne for the trust he had in me, for the freedom he let me. It was very important for me to have this freedom and this confidence, because I cannot work without. I also want to thank him for the continuous support he provided me in difficult moments, and for his encouraging words.

I want to thank my friends and family that helped me when I was out of time and give me a little part of their precious time.

My last thanks go to Yves and Marc for their happiness and for this team spirit. I would also say a special thanks to Anja Bekkelien who provided me a special support, and for the English correction of the document.

To all of you, Sincerely, Thank you.

Summary

This bachelor work will guide you through the secrets of reverse engineering, and the creation of an API.

Firstly the main project was to add functionality to an existing “API”. I quickly discovered that this was not an API, we had to dissociate the API and the application. I tried for some time to do that, but I also discovered that it would not be interesting; time consuming and very difficult to create a good API using this starting point, because there were some strong architectural problems.

Here came the most interesting part of the work: Create form scratch a whole new API that was designed to be very easy to use, easily maintainable, easily extendable, strong, robust, and autonomous.

I used all my knowledge and tried to do my best to give to the public the best API I could give.

I hope I successfully achieved this objective.

Table of Contents

Declaration i

My thanks ii

Summary iii

Table of Contents iv

List of Tables vi

List of Figures vi

Introduction 1

1. The project methodology: Scrum 2

1.1 Introduction 2
1.2 Definition 2
1.3 The Scrum Team 3
1.4 Using scrum 4

2. Improving the Current API 6

2.1 Class diagram 6
2.2 Problems found 7
2.3 Rethinking the API 9
2.4 The wrong start 10

3. Foxy Challenge Trip 11

4. The new API 12

4.1 How to start? 12
4.2 The architecture 14
4.2.1 The process to define it 14
4.2.1.1 Identify the responsibilities 14
4.2.1.2 Regroup functionalities and create the Class responsibilities collaboration cards 16
4.3 Building the API 19
4.3.1 The implemented patterns 19
4.3.2 The event system 22
4.3.3 The ServerProxy 23
4.3.3.1 The AsyncRequest 24
4.3.3.2 The Event System 25
4.3.4 The SpeedCamView 25
4.3.4.1 Rotation of the north sign 26
4.3.5 The DataBuffer 29
4.3.5.1 Description 29
4.3.5.2 Get new Tags 29
4.3.5.3 Optimize connections 30
4.3.5.4 Compute / anticipate the future user position 31
4.3.5.5 Verifying that the DataBuffer works correctly 32
4.4 Document the API 35
4.4.1.1 The UserGuide 35

4.4.1.2 The DeveloperGuide 35
4.4.1.3 The javadoc.. 35

Conclusion and future work...36

Webography 37

List of Tables

Table 1 Problems, incoherence, and improvements of the current API 7

Table 2 Lines of code comparison between Henri's work and mine 10

List of Figures

Figure 1 Scrum process 2

Figure 2 Timeline 3

Figure 3 Class diagram of Henri's "API" 6

Figure 4 Road to take for Foxy challenge trip 11

Figure 5 Speed cams on the way 12

Figure 6 Multi access problem 19

Figure 7 The facade pattern 20

Figure 8 multiple remote server access 21

Figure 9 with proxy pattern 21

Figure 10 Sequence diagram for server communication 24

Figure 11 Example of SpeedCamsView 26

Figure 12 The north sign problem 27

Figure 13 How text is printed 27

Figure 14 Optimizing connections 30

Figure 15 The DataBuffer validation 34

Figure 16 Javadoc with Stylesheet 36

Introduction

During the summer of 2011, a student, Henri La, carried out a bachelor work studying location based advertisement in speed camera mobile applications. To accomplish his work, he developed some Android applications based on the FoxyTag system.

The idea of an Android FoxyTag client API that allows building FoxyTag applications more easily and quickly began to grow in the mind of Michel Deriaz (Director of FoxyTag).

This is the birth of my bachelor work. My main objectives were to modify Henri's work in order to build a professional quality API that could be used in the industry. It is a huge task to think about how an API should be, and to make the good choices between simplicity, capabilities and modifiability. In addition to Henri's work, I had the objectives to add sections (average speed camera) support, and offline mode. If I had had time, I would also have studied how to provide / build a turn to turn navigation system.

I began by studying Henri's work, and I tried to find out how it could be modified in order to become a good and independent API. The output was a list of problems and recommendations to apply in order to make Henri's work easier to understand.

I tried to apply these recommendations, but it quickly appeared to me that it would be incredibly difficult to carry out reverse engineering on Henri's Work.

I tried several approaches in order to reuse huge pieces of functional code, but I discovered that responsibilities were not clearly established, there was no documentation, and the components had low coherence and strong coupling, which made chances of reusability quite low.

With this observation, I decided it would be best to start from "scratch".

1. The project methodology: Scrum

1.1 Introduction

During this project I was involved with a Scrum methodology and in a Scrum team.

What is Scrum?

Scrum is one of the agile management methodologies. Its goal is (like all other agile methods) to reduce the time that development teams need to react to client's orientation changes or new requirements. Scrum also ensures a clear overview of the project, and if it does not help to prevent delays, it helps to detect them as soon as they appear, and to react accordingly.

One of the main points of scrum is that you can expect deliverables/working increment of software at the end of each sprint.

The timeline of the scrum methodology is composed of releases that contain sprints.

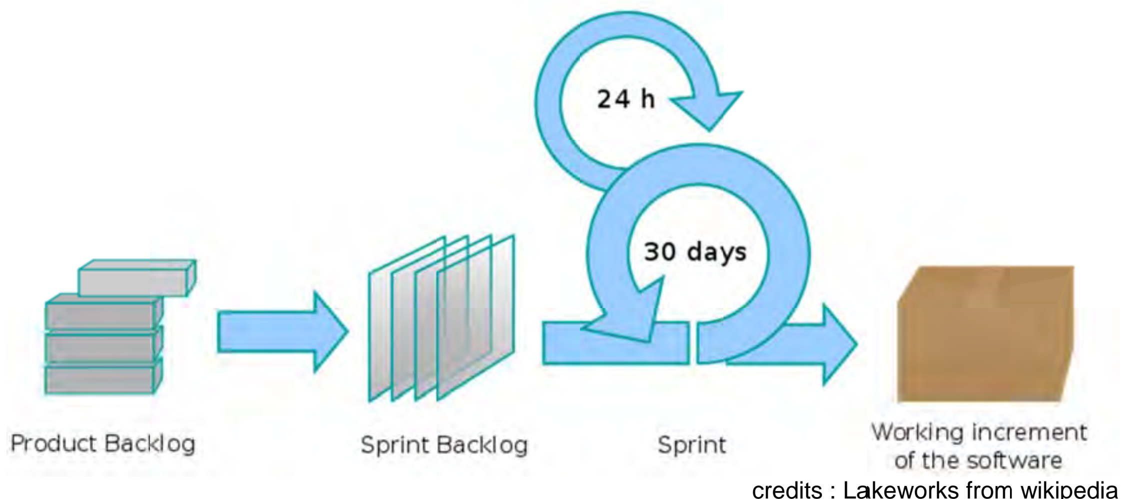
Releases represent huge amounts of time (they usually contains from 2 to 4 sprints), and at each release end, you can expect to see a big progression. Sprints usually last from 1 to 4 weeks.

1.2 Definition

Here I am going to explain some of the key terms and concepts of scrum.

Figure 1

Scrum process



credits : Lakeworks from wikipedia

Sprint: A small amount of time, usually between 1 and 4 weeks.

Story: A job to do. Can be one of these: user story, technical story, and study story.

- User story came from use cases for example
- A technical story can be for example building the virtual development machine
- A study story can be the fact of learning a technology, reading reports …

Product Backlog: A list that contains all the project/product stories. This list is always updated and is not definitive.

Sprint Backlog: Like the Product backlog, it is a list of stories. But all these stories should be achieved during the corresponding sprint.

1.3 The Scrum Team

During this project I was involved in a Scrum team composed of: Michel Deriaz, Anja Bekkelien, Marc Falcy, and Yves Grasset.

The scrum team is composed of different roles:

- **The product owner** is like the customer of the team. He is not the customer, but can be identified as the link between the team and the customer. He is responsible for writing down user Stories, priorities them and add them to the product backlog. The product owner can be part of the development team, but it is not recommended.
- **The scrum master** is like the angel of the team. He tries to ward off distractions and sort out the problems of the work/team circle. He is responsible for the team respecting the Scrum methodology and respecting it's philosophy
- **The development team** is the rest of the team working on projects.

Ideally, we should have one team, one product owner, and one scrum master for each project. It is not recommended that the scrum master and the product owner be the same person.

Here, because we are a small team, and we all work on different projects, Anja, Marc, Yves and I are part of the team as the development team. Michel plays the part of scrum master and product owner.

1.4 Using scrum

For us the sprints were 2 weeks long. You can see here how my time was allocated. As I am writing these words, we are the 8th of May, so my current sprint is sprint number 3 of the 2nd release.

Figure 2

Timeline

At the start of the project, we defined the length of the releases according to the amount of time allocated to the bachelor work.

The Product Owner defines the stories on his own, and places them in the Sandbox. The Sandbox is a temporary place where stories waiting validation are kept. The developer looks at the stories and validates them or discusses with the Product Owner in order to modify, re-evaluate, or divide them in different parts. When the Product Owner and the developer agree on the content of the Sandbox, all the stories in the Sandbox are moved to the product backlog.

At the end of each sprint, each developer comes to Michel, in order to fill the sprint backlog for the next sprint.

At the beginning of every sprint, we do the Monday morning the sprint review for the previous sprint, and the sprint planning for this sprint. In this meeting, we present the work that was done during the last sprint. The work is presented by Michel Deriaz, and not by us. This is to ensure that Michel has understood well what we have done. We present the finished stories and the ones that need to be reported in the next sprint.

After this presentation, Michel introduces the stories of each project for the next sprint. We discuss them, and for each one, we play planning poker: we all have cards with numbers (a Fibonacci sequence), and we vote one time for each story. By doing that, we can compare the time estimated by different people, ensure that the story is understood by everyone and discuss to reach an agreement on good time estimation.

Every morning at 9am there is a daily 15 minutes scrum. During this daily scrum, the whole team is present, and we basically answer one by one these questions:

- What did I do yesterday?
- What will I do today?
- What are the problems I encountered?

This way we can know who have problems, detect delays rapidly, and react in consequence.

2. Improving the Current API

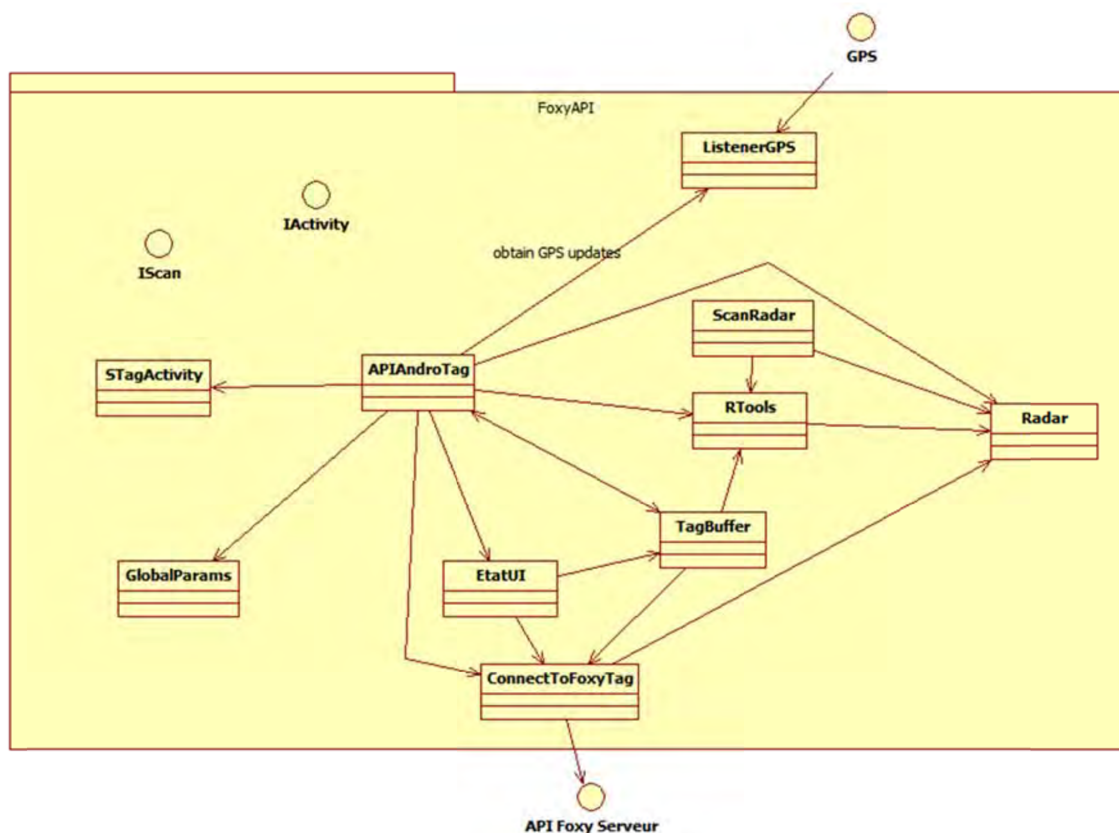
I started my work trying to improve Henry's "API". The first challenge was to understand how this "API" was built, and learn to think like Henry in order to be able to understand his code.

2.1 Class diagram

In order to better understand Henri's work, I decided that a class diagram would be a good start to identify the different classes, responsibilities and collaborations.

Figure 3

Class diagram of Henri's "API"



2.2 Problems found

After a superficial code mining, I found some incoherence, problems, and improvements to carry out.

Hereunder, you will find both the problems encountered and the potential improvements:

Table 1

Problems, incoherence, and improvements of the current API

Class	**utility**	**Accessibility**	**Remarks**	**Requires**
APIAndroTag	Main entry point that the client app will use	Public	Is actually the main entry point for the API. This should become the **unique** entry point	STagActivity TagBuffer ConnectToFoxyTag EtatUI GlobalParams ListenerGPS Radar RTools STagActivity TagBuffer
ConnectToFoxyTag	Manage the communication with the server	Public	3 functions uses the same code -> DRY (don't repeat yourself) -> factorize the code. Create a method that allows you to launch a request specified in parameter, and to get the result String	Radar
EtatUI	Store the UI state	Public	Name in French Error messages hardcoded	ConnectToFoxyTag TagBuffer
GlobalParams	Contains all configuration parameters	Public	The parameters are Final. That means the client cannot modify these parameters. He should be able to do that	Nothing
IActivity	interface	Public	Empty - Verify that it is truly necessary	Nothing

IScan	interface	Public	Empty - Verify that it is truly necessary	Nothing
ListenerGPS	Listener that writes GPS informations into AndroTag	Public	Seems to be almost perfect.	Nothing
Radar	Business class that represent a speed cam	Public	Override the constructor so that it takes a parameter android.location.Location in complement / replacement of latitude and longitude	Nothing
RTools	Some tools	Public	Some incoherence: a function named m/sTOkm/h can return miles per hour depending on a Boolean! Factorize some code of resizeImage functions Should be fully static	Nothing
ScanRadar	A view of the speed cams around	Public	Colours cannot be modified.	Radar RTools
STagActivity	Abstract Activity	Public	Completely modify this activity. We should be able to extend this activity, and in a few lines of code be able to build a application	Nothing
TagBuffer	Class that maintain the speed cams around the user and that updates them smartly	Public	TagBuffer contains some methods that predicts the future heading. This should be located in RTools	APIAndroTag ConnectToFoxyTag Radar Rtools

2.3 Rethinking the API

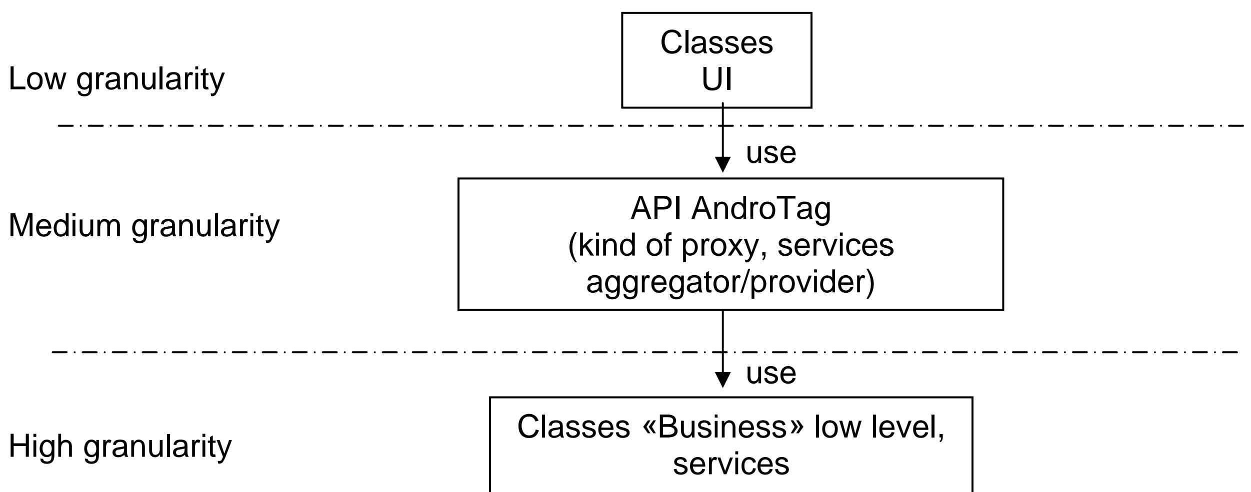
After having submitted these improvement suggestions, I studied the interactions between the API and the client application. What I saw was a big mess which resulted in the client application accessing the API in many different ways. The API and the client application were very strongly linked. This is because the applications and the API were developed in the same time by the same person.

I decided that we should rethink the way the client application accesses the API: restrict the communication path, reduce dependence, and simplify the communication.

As developers are all different, and their needs are different too, I thought that it would be a good idea to provide different granularity access for the API.

By granularity we mean level of detail.

Here is a scheme that explains my way of seeing things.



2.4 The wrong start

I started to work very hard to improve this "API", and to try to dissociate it from the application. As everybody knows, in reverse engineering for maintenance, understanding the code takes up to 80% of one's time whereas correcting it takes only 20%.

But in this case, it was not only maintenance, there were architecture modifications too. It quickly came to my mind that in this configuration, and with this particular code complexity, I would always be under the threat of side effects; I would spend approximately 90% of my time understanding the code, and studying how I could correct / improve it without breaking the entire "API". This "API" was a single component, and not several smaller components. That means that when you make a single change, it can be reflected in the whole "API".

Table 2

Lines of code comparison between Henri's work and mine

	Henri's "API"	Actual "API"	Action
TOTAL	1407	1465	
GlobalParams	16	162	Centralization parameters
ConnectToFoxyTag	158	127	Factoring code
ListenerGPS	25	41	Add functions for network state
ScanRadar	212	206	cleaning and centralisation of parameters
RTools	133	106	Factoring code

As you can see in the table above, I started correcting API problems, before discovering how every single piece of code was strongly linked to each other.

It was obvious that if we wanted to produce a high quality API from this code it would be very difficult and time consuming. And more importantly, spending a lot of time doing reverse engineering does not lead to an interesting bachelor work.

I discussed that with Michel. He had no idea what Henri's code looked like, and when I explained this to him he agreed with me that we needed a fresh start.

3. Foxy Challenge Trip

Before we started working on the new API, Michel and I defined, that it would be a good idea to create a “Framework” that would verify that my API respects the rules defined by the server API document (appendix 1). This document explains how to communicate with the server, and what rules we must respect to have access to the production database.

A test database is provided for application developers. We are able to post all kind of tags (speed cameras) that we needed in order to test our application.

In order to verify that my API is Foxytag compliant, I have written down a document that describes how to test the application. It is a kind of stress test that ensures that your application complies with all the rules.

I called this document the “Foxy challenge trip”. It describes the path you have to follow, the points of interest (different kinds of speed cams) you will meet on your way, and the expected results.

This complete document can be found in appendix 2, but I will show you here the path to take, and the speed cams I have inserted into the database.

Figure 4

Road to take for Foxy challenge trip

Figure 5

Speed cams on the way

4. The new API

4.1 How to start?

First of all, we need to define what an API (Application programming interface) is.

If we want to explain what and API is, we must understand the concept of interface.

An interface is a way to define what the actions you can do are, and what the expected results are. It can be compared to a native language: when you say hello to someone, you expect this person to do the same to you.

With this starting point, we can say that every single piece of software has its own interface, from a single method to a whole system.

So is API a good word to describe what I am building? Not completely, but it is not wrong either. In fact I am not building only an API, but a library. In fact I am simplifying Android developer's work. My API has no reason to exist if nobody uses it.

I will build a complete and complex system, and the API of this system, will be the door to access the services provided by this system/library, as you wish to call it.

This library is useless if there is no API to use it. Since only me will know the whole library, the developers will only see and use the API. So I think we can call it an API. This term is commonly accepted and used by developers.

Now we know what we are talking about. But how to create an API? Are there rules? Is there a step to step path to follow? Are there common errors to avoid?

I have read a lot of documents, and browsed websites on this subject, but much of what I found was only advice. There are no established rules or methods to create a good API.

My main inspiration point was the presentation by Joshua Bloch (Principal Software Engineer at Google) on the theme “How to design a good API, and why it matters”

Full references in webography.

He says that *«Public APIs are forever – one chance to get it right»* (page 2), which means that if you release a lousy API on the market, developers, and users will try to use it, and if it does not suit to their requirements, or it is bugged and does not provide the expected results, people will drop your API and change for another one, or build their own.

In his opinion the characteristics of a good API should be:

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to the audience

The reason your API exist is to be used. It can be used only if it suits your user’s requirements.
So firstly you need to be sure that you have defined the good requirements for your API. One good way to do that is to take them from the use cases.

4.2 *The architecture*

4.2.1 The process to define it

4.2.1.1 Identify the responsibilities

High level ones:

- Give GPS accuracy information.
- Give Data pertinence information.
- Give Data connectivity (2G/3G) information.
- Communicate with server (init, tag/section post, tag/section request).
- Display tags and section
- Give tags/section around
- Create a Tag
- Create a section
- Alert client app when GPS, data, network Status changed
- Alert client app when the position has changed
- Forward messages from server API to client app
- Alert client app when there is no data and then it can provide its own data
- Provide the possibility to store tags/section and to delay the sending to the server

Secondly, we need to detail these responsibilities in order to discover low level ones.

Explaining scheme:

- High level responsibilities
  - Low/Middle level responsibilities
    - Requirements for this responsibility

- Give GPS accuracy information:
  - Computes the GPS accuracy into different accuracy levels
    - GPS information

- Give Data pertinence information:
    - Know if the data are up to date
        - Know Position
        - Know the area covered by actual data.
        - Know the freshness of the data.

- Give Data connectivity (2G/3G/Wifi) information.
        - Connectivity information

- Communicate with server (init, tag post, tag request).
    - Know the user login and pass; the client, version, language, platform
    - Know the server address, and lab Address
    - Know the Tag/Section to post
    - Recover server's answer, analyse them, and parse the result
    - Launch requests
        - State of the data connectivity

- Display tags/sections
    - Know the tags/sections around
    - Alert user when a tag is approaching

- Give tags around
        - Know position

- Create a Tag
        - Know position
        - Know speed
        - Know heading
    - Enable tag creation only on minimum speed and accuracy

- Create a section
        - Know position
        - Know speed
        - Know heading
    - Enable section creation only on minimum speed and accuracy

- Alert client app when GPS, data, network Status changed
    - Alert client when GPS status changed
        - Be aware of GPS accuracy changes
    - Alert client when data status changed

        - Be aware of position changes
        - Be aware of data pertinence
    - Alert client when newtork status changed
        - Be aware of server API communication attempts

- Alert client app when the position has changed
        - Be aware of position changes

- Forward messages from server API to client app
    - Decode server API answers, and dissociate business answer vs inf/pub messages
        - Communication with server

- Alert client app when there is no data and then it can provide its own data
    - Update the data status according to the client's answer

- Provide the possibility to store tags/section and to delay the sending to the server
    - Tells the client that these tags are not already sent or being sent
    - Tells the client when the tags are sent.
    - Send the tags when the network is free and available
        - Know network state

4.2.1.2 Regroup functionalities and create the Class responsibilities collaboration cards

ServerProxy		
Manage server communication		
Responsibilities	**Requirements**	**Collaboration**
Know the login and pass, the client, version	Must be provided in any way by external source (client app)	
Know the prod and lab server address		
Post a Tag	Data status (Unavailable/sending/free)	AndroTag, ASyncRequest
Post a section	Data status (Unavailable/sending/free)	AndroTag, ASyncRequest
Init the server	Data status (Unavailable/sending/free)	AndroTag, ASyncRequest
Forward Server messages to client app		AndroTag
Recover the request result and parse it correctly depending on the type of the original request	Must know witch request was launched	ASyncRequest

Keep the last server's answer in memory		
Keep the last request position in memory		

AndroTag		
Main entry point for client app, and main communication bus for API		
Responsibilities	**Requirements**	**Collaboration**
Manage the event publish/subscription system		AndroTagListener
Create a Tag		Tag
Create a section		Section
Know and provide data connectivity status		ServerProxy
Know and provide GPS accuracy	Must be subscriber of GPS events	
Know and provide Data freshness/pertinence status		DataBuffer
Compute GPS accuracy into 3 different accuracy levels		
Must Authorize server communication depending on some parameters	Speed, network data status	
Manage and forward all requests for server communication		ServerProxy
Allow creation of tags only when it's authorized	Speed, GPS accuracy	

AndroTagListener		
Interface used by API to send messages to client app		
Responsibilities	**Requirements**	**Collaboration**
Defines the messages that the API can send		

Tag		
Representation of a speed cam		
Responsibilities	**Requirements**	**Collaboration**
Know when it is complete and can be sent		

Section		
Representation of an average speed cam		
Responsibilities	**Requirements**	**Collaboration**
Know when it is complete and can be sent		

DataBuffer		
Must contain speed cams around the user		
Responsibilities	**Requirements**	**Collaboration**
Provide good data corresponding to the user's position	Communication with server	AndroTag
Update data freshness status		AndroTag

SpeedCamsView		
Must display speed cams to the user		
Responsibilities	**Requirements**	**Collaboration**
Display the tags	The position of main area tags	DataBuffer
Make a sound on speed cam approaching user's position	User's Position, Tag and section position, the sound to make	Parameters

Tools		
Static class that contains useful static methods		
Responsibilities	**Requirements**	**Collaboration**

ASyncRequest (extends AsyncTask)		
Launch request, and provide answer		
Responsibilities	**Requirements**	**Collaboration**
Launch an asynchronous request and recover the result	The request as String	ServerProxy

Parameters		
Static class that contain the parameters		
Responsibilities	**Requirements**	**Collaboration**

Hearthbeat		
Know GPS information, and determine when to refresh the API		
Responsibilities	**Requirements**	**Collaboration**
provide GPS information (lat, lon, accuracy)		

4.3 Building the API

Here I will describe and explain some specific and important piece of software that are in the core of the API

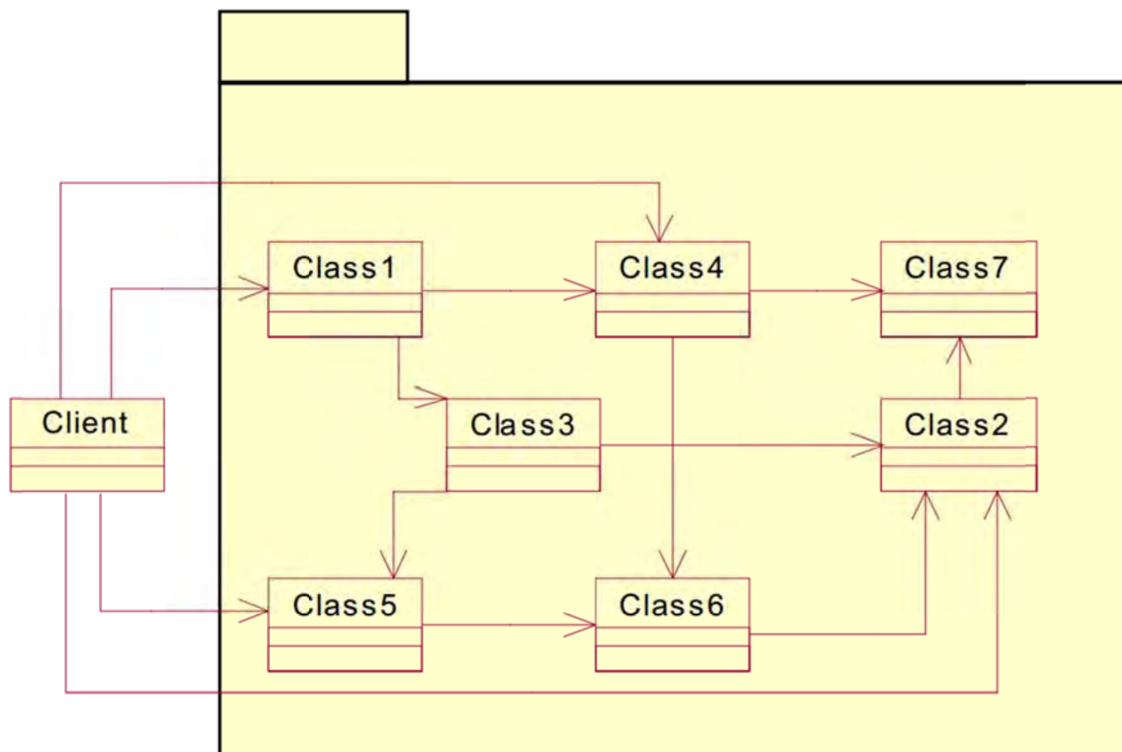
4.3.1 The implemented patterns

I have decided to implement two architectural patterns in order to simplify the code and make it more understandable.

I faced two problems that fortunately have their patterns in order to solve them.

The first problem was an access problem. If we want the API to seem simple for the client, we have to limit the number of classes it has to use. In the example below, it is a typically the problem I had. The client must access to different classes in order to obtain the desired information/service. But if we want the client to have access to this information, how can we reduce the number of classes known by the client? By placing an intermediate!

Figure 6
Multi access problem

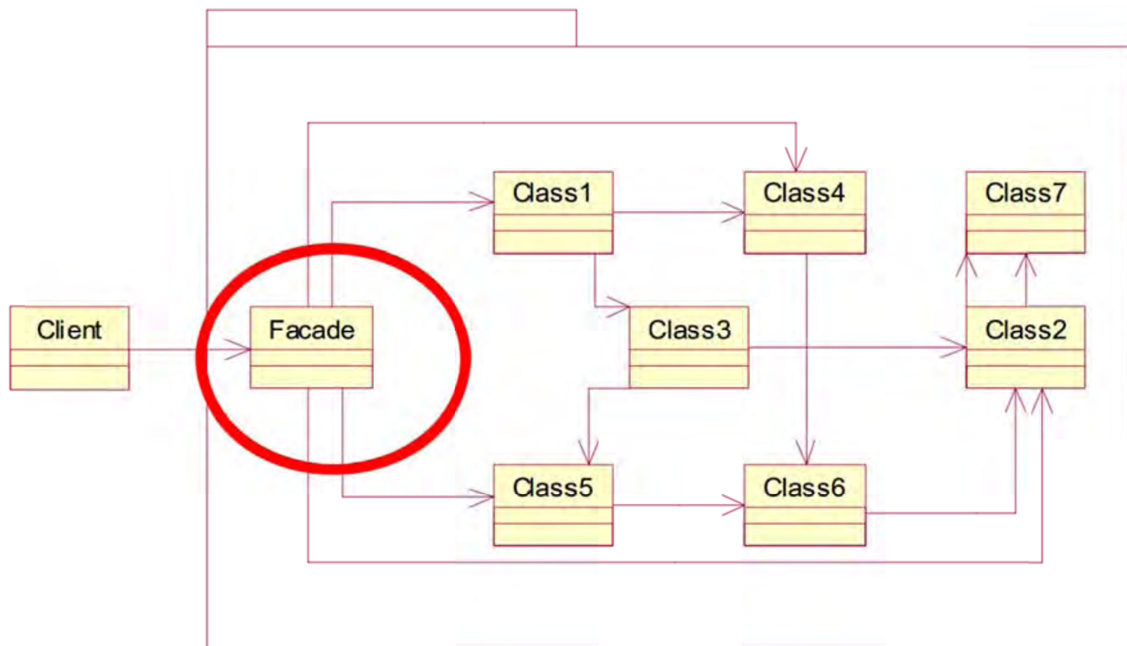


Credits: Philippe Dugerdil, Software design lesson

This intermediate is employed into the facade pattern that I chose and that perfectly solves the encountered problem. Hereunder, you can find for example the implementation of façade that solve the problem described in the image above. In our case, the façade is the AndroTag class.

Figure 7

The facade pattern



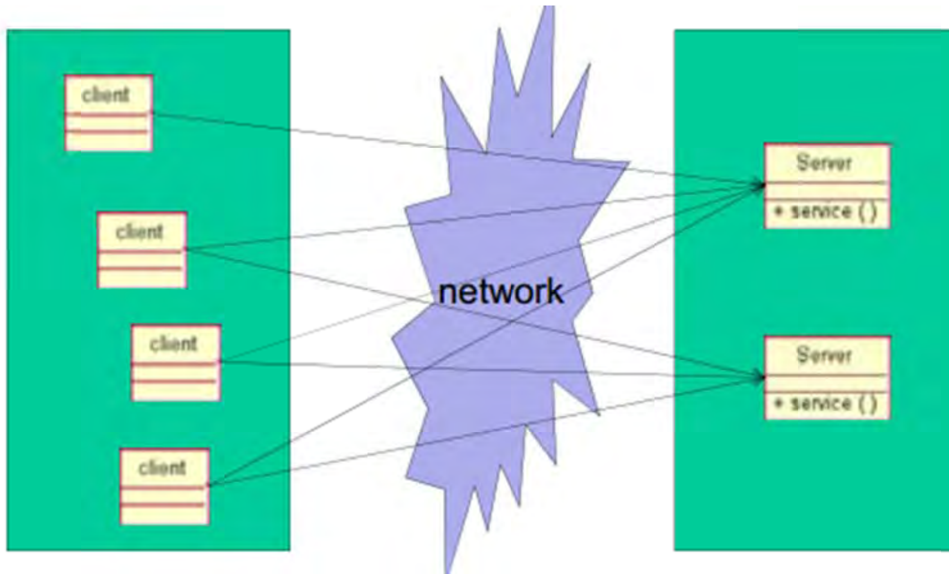
Credits: Philippe Dugerdil, Software design lesson

The other problematic I experienced was that some different classes have to communicate with the remote FoxyTag server. How to manage these connexions, and reduce them?

Hereunder a graphical explanation of the problem:

Figure 8

multiple remote server access

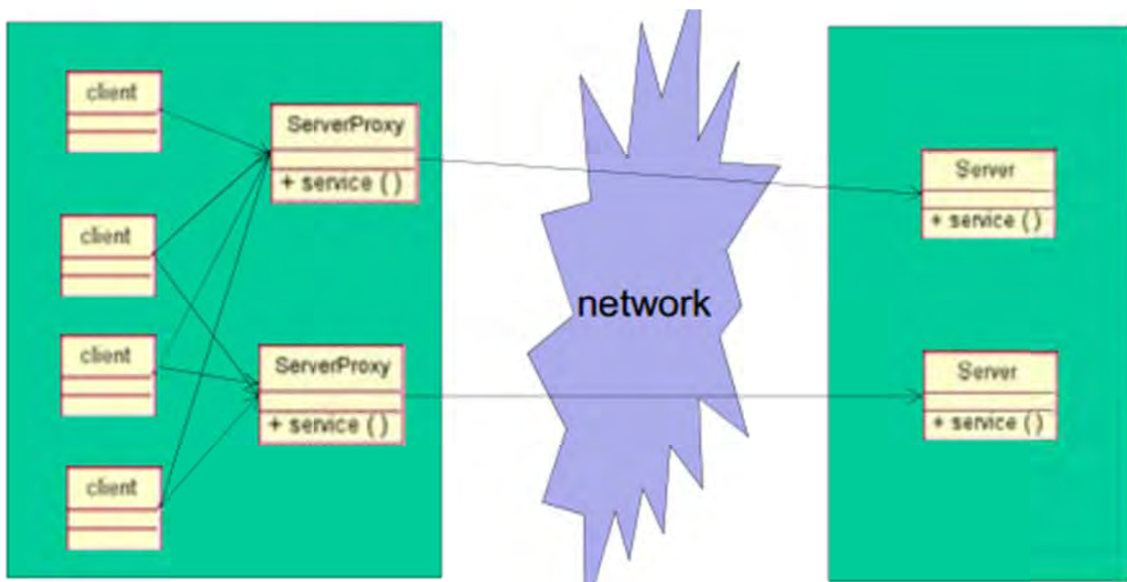


Credits: Philippe Dugerdil, Software design lesson

The solution was to create a specialized component for the communication with the server. It is the idea behind the Proxy pattern as you can see below. I implemented this pattern in the component ServerProxy of my API.

Figure 9

with proxy pattern



Credits: Philippe Dugerdil, Software design lesson

4.3.2 The event system

The old API was based on a special way of communication: Each class has a link to other classes, and they were using this in order to communicate. I introduced a new way of communication: the event system.

The idea was to provide a simple, robust and powerful system that can manage the communication between classes inside and outside the API.

The only class that have direct links to other classes is the AndroTag class, because it is the façade.

The other classes use the events in order to communicate.

There are 5 events. For now, only 4 are really useful, because one refers to the sections, and sections are not yet implemented into the API.

Here is the content of the AndroTagListener interface (the available events):

`msgServer(String cmd, String msg, String details)`
Called each time the server sends a message.

`positionChanged()`
Called each time the current position changed.

`sectionUpdated(String speed, int timer)`
Called each time the data about the current section is updated.

`serverAnswer(boolean requestEnded, String cmd)`
Called each time the server answer to a request or the request timed out.

`statusChanged()`
Called each time that the status of the data, the GPS or the network has changed.

For each of these events, there is a protected method in AndroTag called fireEVENT_NAME. These methods are protected, because logically, only the API can launch API's events. These methods are located into the AndroTag class that is in the center of the API. The classes that want to subscribe to these events must satisfy the AndroTagListener interface and must be registered using the method *addAndroTagListener* of AndroTag class.

4.3.3 The ServerProxy

The sever proxy will be the main communication path with the server. The entire API will ask the ServerProxy to communicate with the server. Normally, all the communication requests should be sent to the AndroTag class that will forward the requests to the ServerProxy one. But there are some exceptions, and some classes ask the ServerProxy directly. But these are very specific cases.

As said before, this class is responsible for the collaboration with the server. In this class, we will find the first improvement compared to Henri's work:

Asynchronous requests.

In Henri's work, the User Interface (in fact, the whole program) blocks while waiting for a reply from the server. And in Android, when the UI is blocked for some time, the system tells you that your application is not responding, and proposes you to kill it.

But now, we have introduced parallelism!

The ServerProxy is able to launch asynchronous requests (parallel tasks) and to get the result. As you can see in appendix 1 (Server API) there are different kinds of requests that can be sent to the server, and more importantly, there are different results, in different formats.

If we can launch some requests, how to know how to parse the results? How to know which answer corresponds to which request?

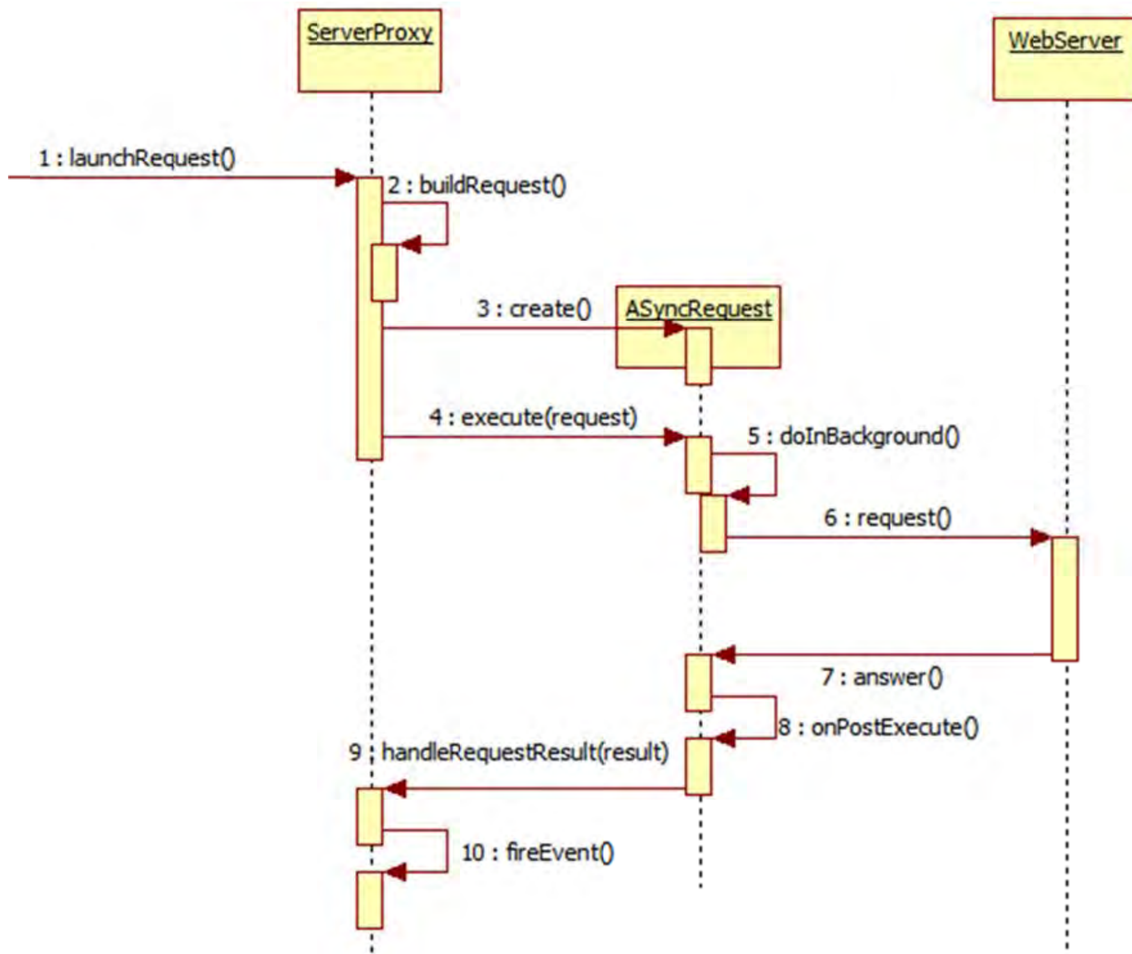
That may be possible, but it is not necessary to answer this question in this case. As the communication with the server is not heavy (1 request per minute in average), we have established a rule: One server communication at a time.

The ServerProxy will launch a request, remember the kind of request in order to correctly parse the corresponding answer; and will not allow any other request until it gets an answer from the server.

Bellow, you can find a sequence diagram that will show you graphically what I have just explained. The reality is not exactly the same, but not far from it.

Figure 10

Sequence diagram for server communication



4.3.3.1 The AsyncRequest

The AsyncRequest is a very simple component that extends android.os.ASyncTask. It is the same component that is called for every kind of request.

I will explain to you how it works: You create it, and once it is done, you call the method “execute” that takes an array as parameter. As we need to launch only one request, we just fill the first position of the array with the request we need to send. The request as we call it is the url to reach in String format.

After that, ASyncRequest will communicate with the server until it gets the answer. Directly after, it will call the HandleRequestResult method of ServerProxy and set as parameter the BufferedReader that contains the result.

How can ASyncRequest call this method? How does it know ServerProxy? In fact it is quite simple: ServerProxy is a singleton, so ASyncRequest gets the ServerProxy instance, and after that calls the method.

4.3.3.2 The Event System

As these requests are asynchronous, we cannot know the result immediately. In order to know when a result from server is received, a dedicated event exists: It is $nswer\ (boolean\ requestEnded, String\ cmd)$.

In this the AndroTag class there are two different concepts to differentiate:

- The action to request or post something
- The action to get something

For example, if you want to get new tags, two functions are at your disposal: $getTags()$ and $requestTags()$. Which one to use?

Firstly you have to call $requestTags()$. That will generate a request for the server that asks the tags around you or around a specified position. Once it is done, and when the server will answer, the $serverAnswer$ event will be launched. If the request was successful, you can now call $getTags()$ in order to have the last speed cams.

Remember that if you call $getTags()$ while $requestTags()$ was never called, you will get null.

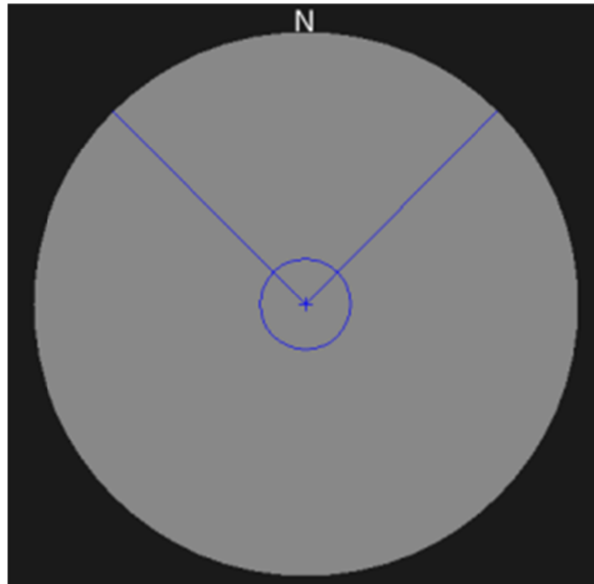
Now you have understood the concept, let me explain a few more details. When you call methods that request communication with server, you will get a boolean as answer. This boolean only tells you if the request could be launched (true) or not (false). You will only know if your answer was successful by listening the event.

4.3.4 The SpeedCamView

This component extends android.view.View, and is designed to be easily integrated, by implementing functions such as auto-resize depending on the parent's width. This component is a smart one, and functions on its own with the API.

Figure 11

Example of SpeedCamsView



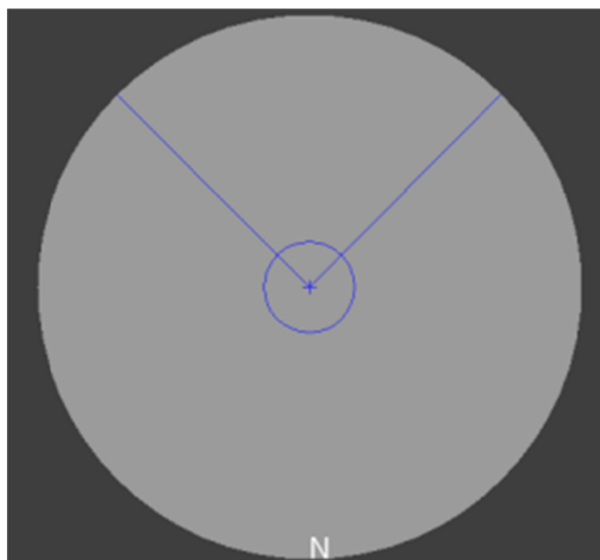
4.3.4.1 Rotation of the north sign

As you can see in the figure above, there is a north sign around the external circle. It should move around the circle in order to indicate where the north is, depending on our current direction.

It uses simple trigonometry to calculate where the sign should be depending on where we are heading. The position given by this function is expressed in terms of abscissa and ordinate. The coordinates (0;0) represent the centre of the circle. But, after implementing these functions, I detected a problem.

Figure 12

The north sign problem



As you can see above, in this case our heading is 180 degrees, direction South. The North sign is almost well positioned, but is a little on the north east of the position it should be (it should be out of the grey circle). Why?

Is the trigonometry function I have written malfunctioning?

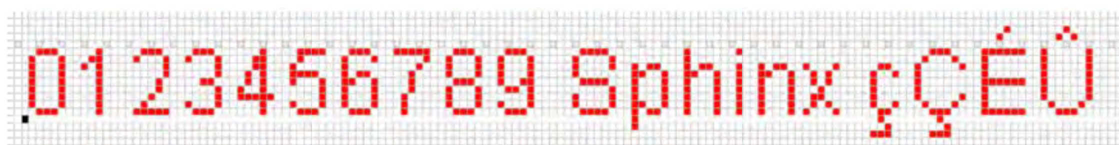
In fact the North sign position calculation is correct. To understand the problem, we should first understand how the North sign is draw/displayed on screen.

I use Canvas.drawText to draw the N sign on the screen

In this function you can specify where you want to draw the text. Here I use the coordinates provided by my trigonometric function. But these coordinates do not represent the centre of the text, but the left-bottom corner (see figure bellow to better understand how text is displayed). So knowing that, let's look again at the figure above, and look precisely at the left-bottom Corner of the "N" char. It seems to be correctly positioned. So the trigonometry function is not the cause.

Figure 13

How text is printed



Crédits : Peter Daehne, Métrique des polices

The black pixel represents the coordinates we specify while calling the Canvas.drawText function.

Now we know what the problem is, but we don't have a solution yet. And here I just showed you the problem for heading = South. But there is a problem for almost all angles. In fact, only the top-right quarter has no problems, because as I said the position computed by the trigonometric function is good, so the text will be written above and to the right of the computed point. As the circle is bellow and to the left of this point, the text will never enter in this circle (for this quarter). But for the three other quarters, think about it, and remember where the text is displayed in relation to the point calculated by the trigonometric function. You will rapidly see the problem. The problem is complex, because we need to find a solution that suits all the headings.

We need to work on the two axis, the abscissa and the ordinate.

For the ordinate, this is quite simple: When we are in the most meridional point, we need to move the N sign x pixels down (x pixels is in fact the height of the Character). We have found a solution but this works only for this most meridional point. What to do between this point and the most septentrional point?

We need to build a function that will displace the N sign for a given intensity at a given position.

Here is the little formula I managed to create:

$$posY \mathrel{+}= (textSize - (textSize / 4)) * ((radius + posY) / (2 * radius));$$

posY is the y position where the N character will be displayed.

radius is the radius of the external circle in pixels.

textSize is the height of the N character.

This is now done for the ordinates; let's work on the abscissa axis:

It is almost the same problem, so the solution is similar:

$$posX \mathrel{-}= (textSize - (textSize / 4)) * ((-radius + posX) / (-2 * radius));$$

posX is the x position where the N character will be displayed.

With these two little lines, we can take the N sign’s height and width into consideration in order to provide an almost perfect positioning.

4.3.5 The DataBuffer

4.3.5.1 Description

The DataBuffer is responsible for keeping data up to date. It uses ServerProxy to communicate with the server, and knows when it is necessary to update the tags / sections.

The purpose and responsibility of the DataBuffer is to always (as far as possible) have at its disposal the tags and sections that surround the user. The FoxyTag server gives us tags in a circle with a radius of 7.25 km around the coordinates we asked. From the moment we leave this area, the DataBuffer must necessarily obtain new tags in order to stay up to date. The DataBuffer must of course anticipate that a server connection is not immediate, and that response times may vary. Rather, it should provide a safety margin. Here I will mostly use the word “Tag”, but it is quite the same for the sections.

4.3.5.2 Get new Tags

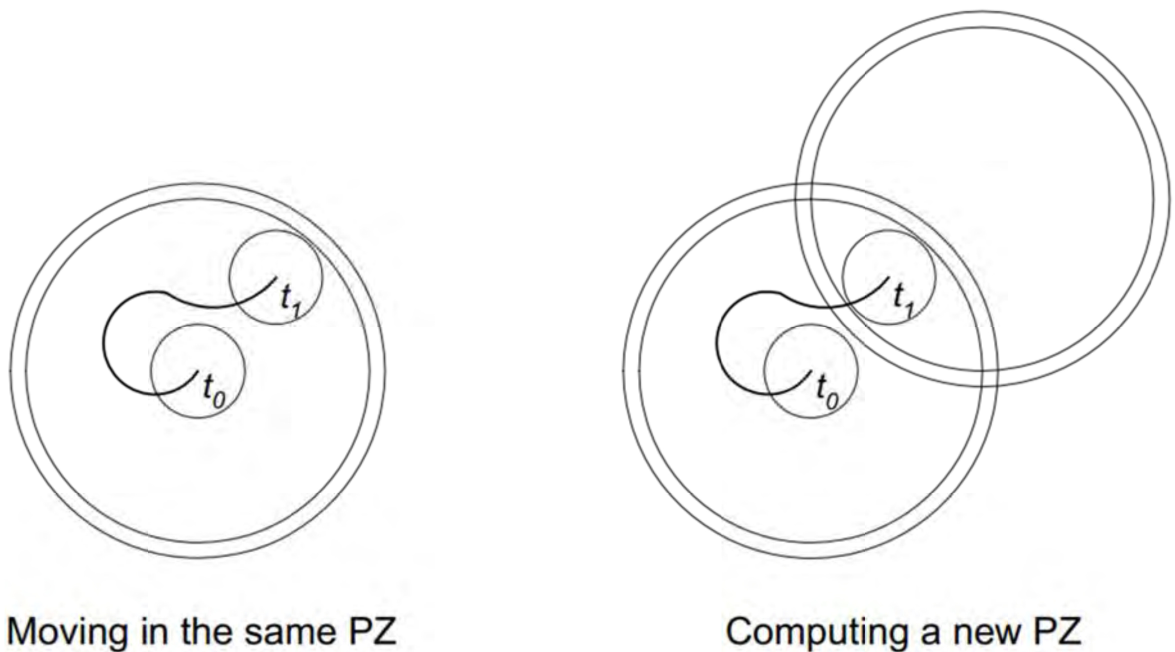
There are two ways to get tags when we approach the boundary of the circle containing the current tags.

• The "brute" method which consists in not asking questions: when we approach the boundary area, to simply ask for tags around the current position of the user. Simple and effective, but not optimal, because when doing it this way, we will receive tags for a circular area around us, and therefore tags behind us. It is unlikely that a user will suddenly go back on his path, so what is the point of requesting tags for a region where the user will not move? There is no interest, and it is a lack of optimization.
• The "smart" method that consists in guessing and anticipating the future position of the user in order to get tags in a region where it is likely that the user will go in light of his current route. It therefore does not require tags around us but in front of us. It is this possibility that I have chosen and set up.

4.3.5.3 Optimize connections

Optimizing the connections means doing less, and to do less, we must do smarter. We do this by choosing the strategy of point 2 mentioned above.

Figure 14
Optimizing connections



Credits: Michel Deriaz, Server API

This scheme should help us to see things more clearly. Here a PZ refers to "protected area". PZ represent the area within which we are currently covered by the tags contained in the DataBuffer. Let us concentrate on the left diagram. The outer circle is the circle of 7.25km radius around the user, it is the PZ. Just inside there is another circle of slightly smaller size, representing the PZ minus the safety margin (time to establish a connection to the server and retrieve the results). Then we see the points t_0 and t_1 which are different positions of the user at time t. Around these positions, we see a circle. This circle defines the area that the user sees. Specifically, within this zone, radars (tags) must absolutely be up to date, since the radars in this area are those shown to the user.

How to define when we leave the PZ, and for witch position should we request tags.

Well that's very simple. When the zone of visibility (small circle around the user) touches the large interior circle (PZ - buffer), then it becomes necessary to request new tags. You can see this scenario on the left diagram, position t_1.
On the right diagram, we see an example of an automated connection: the user reaching the limit of its area, the system then calculates the optimal location to download the new tags.

4.3.5.4 Compute / anticipate the future user position

How to define the future position of the user? How do I know if he will not suddenly turn and destroy all our plans / forecasts? The simplest would be to use a soothsayer.

Unfortunately, Android does not (yet) provides this kind of services in its API.

We cannot know precisely where the user will be at time t, but we can try to guess. Although roads are rarely linear, and they sometimes make us move away from our destination, we usually go in a specific direction. It is not possible (in fact it is, but not in our case) to know the roads in front of us, and therefore to predict more or less precisely the path of the user, but it is possible to define the overall trend of this movement.

How to proceed?

We must ask ourselves several questions:

- Is the accuracy of this calculation fundamental?

- Does this connection optimization worth/require a heavy CPU load?

- Do we need to use a complex algorithm whose implementation and adaptation would be highly time-consuming?

As the problem we are trying to resolve is just an optimization it did not seem necessary to implement highly complex solutions. We chose a solution that could not be simpler, which according to our tests, has been fruitful.

Here's how:

I will not remember the many former positions of the user. We could have done this to determine an overall trend direction, but the inherent complexity of the calculations do not worth the shot.

I adopted an approach that could not be simpler. It has two stages:

- First, when the user receives new tags from the server (logically this response answer means that we asked for these tags, and so we were about to leave the area). As we have received new tags, it means that now we are once more in a good PZ.

- A few minutes later we arrive at the limit of PZ, and it is time to seek new tags.

So we will calculate the direction of the user. This is done using the previous as well as the current position of the user. Once we have the direction, we will ask for tags around a position that we have computed x meters away from the current user position, and in the computed direction. The distance x meters is calculated using three factors: the radius of the PZ, the safety margin, and the radius within which the tags are visible to the user.

The calculation of this distance is very important because it would be pointless to ask Tags for a PZ that would be out of range and therefore generate a new request for tags using the same method. We can clearly see an infinite loop here.

4.3.5.5 Verifying that the DataBuffer works correctly

<u>Measures strategy:</u>

In order to verify the good functioning of DataBuffer, we need to collect some parameters:

The user position in crucial positions / time

The distance that separate the centre of the PZ and the user

The user position when the system starts to ask new tags to the server

The user position when the system receives the new tags

The asked PZ coordinates

The computed heading (direction)

For my measurements, I have taken the following values:

TAG_RADIUS (PZ radius): 2km

REQUEST_DELAY_SECURITY: 200m

RADAR_RANGE (radius of the area within the tags are displayed): 300m

I used System.out.println() in order to write in the logcat the desired information.

Here is a little part of primary information that was collected at the end of this measurement campaign:

> *04-25 12:56:03.710: I/System.out(3633): Demande de nouveaux tags. Distance = 1500.2227251182462*
> *04-25 12:56:03.710: I/System.out(3633): Current position lat: 46.16139 lon: 6.18052 heading calculé : 147.17852783203125 Tag request demandé: lat: 46.15005 lon: 6.19107*
>
> *04-25 12:56:06.639: I/System.out(3633): New last update position : lat :46.16125702857837 lon : 6.180217266082584*
> *04-25 12:56:06.639: I/System.out(3633): Server answer received. Reached the server : true | command : tagrequest*

This is just an example, because in reality, there are 160 lines to analyse

Representing data:

To ensure that this raw data can be understood, we must analyse them, and make a visual representation if possible.

I used Google Earth to address this problem.

The only problem with this software is that it is not possible to draw circles. Indeed, it would have been very convenient to be able to do this especially to visually display the area covered by radar, and the area outside of which it is necessary to update the data.

However, I found a solution:

The website http://www.freemaptools.com/radius-around-point.htm can draw "circles" (in fact they are highly detailed polygons that look like circles, because Google Earth

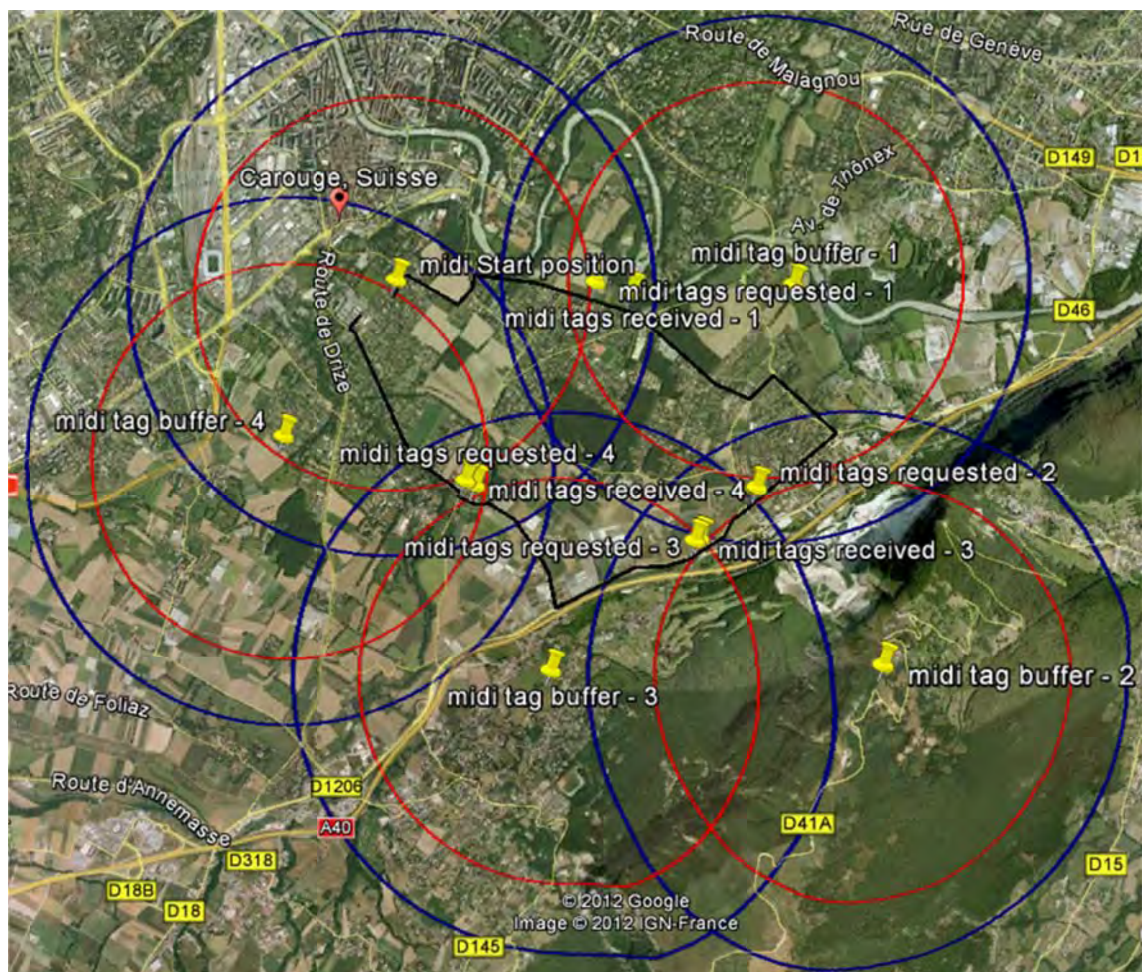
knows only polygons) of a desired radius, and a desired position. Once done, you can download a KML file containing the "circle" and which can be opened by Google Earth.

I did this for each circle.

I also reported on Google Earth the center of each of these circles, the position of the user during each tag request, and the position of the user upon receipt of new tags.

Here is the report that helped to validate the proper functioning of the DataBuffer:

Figure 15
The DataBuffer validation



Legend:

- The blue circles represent the PZ

- The red circles represent the area beyond which it is necessary to update tags (2km-(200m +300 m))

- The black line represents the route

- The yellow pins are the centres of the PZ, and the places where we asked for tags, and where they were received.

4.4 Document the API

Build correctly the API is one thing, but document it is also very important.

I have created a set of documents, explanations, guides, referential that should help the client that want of basically use the API, or want to modify it, continue its development.

4.4.1.1 The UserGuide

The user guide is dedicated to the clients that want to do a basic utilisation of the API. You can find this document in appendix 3. I decided to place it in Appendix because it is a complete document, and use different graphics standards. This document was totally written by me and should be considered as a part of this bachelor work.

4.4.1.2 The DeveloperGuide

The user guide is dedicated to the clients that want to dive more deeply into the API, and want to modify it, improve it or continue its development. I decided to place it in appendix 4 because it is a complete document, and use different graphics standards. This document was totally written by me and should be considered as a part of this bachelor work.

4.4.1.3 The javadoc

The javadoc is well known by the developers. If the code is correctly documented, you can easily generate documentation that details your functions, and constants. This is an important part of an API, because every developer has already checked a language doc. It is very easy to generate javadoc, but comparing to Android doc, the generated documentation is not very appealing. In order to improve that, it is possible to provide a Stylesheet. Here under you can see the effect of one (you can find lot of them on the net) stylesheet on the javadoc. It's clearer, and more pleasant to read.

Figure 16

Javadoc with Stylesheet

All Classes

- AndroTag
- *AndroTagListener*
- ASyncRequest
- DataBuffer
- HeartBeat
- Section
- ServerProxy
- SpeedCamsView
- Tag
- Tools

- Package
- Class
- Use
- Tree
- Deprecated
- Index
- Help

- Prev Package
- Next Package

- Frames
- No Frames

Package ch.unige.androtag

Interface Summary

Interface	Description
AndroTagListener	A way for the API to inform the client application about changes.

Class Summary

Class	Description
AndroTag	Entry point of the API.
ASyncRequest	Launches asynchronous requests and handles the result.
DataBuffer	Will keep tags and sections up do date.
HeartBeat	This class will tells the AndroTag class to react when some GPS parameters changed or at least every second.
Section	A section.
ServerProxy	This class is the request launcher.
SpeedCamsView	A scanner that shows the neighboring tags.

Conclusion and future work

I found this work very interesting, because I encountered the problems we usually find in modern software development. I had to deal with reverse engineering, I had to create a complete system, think about simplicity, usability, maintainability, and so on.

In fact I really liked that because I had to touch to every approach, every specificity of software development in order to be able to provide the best API I could do. I am really happy that this bachelor work gave me the opportunity to apply all the software design theory I had learned for years.

As this API was designed in order to be highly expendable and easy to understand and to learn, I hope someone will continue my work. There are so many things that can be added to this API. We can start with section. The basis and architecture for the section implementation is already here, and there is not a lot of work to do to implement that. We can implement offline system for the storage of tags, turn to turn navigation system. In fact the possibilities are endless. I hope I achieved to provide a good start for future work, and future developers will appreciate my work and the importance I gave to comprehensibility. As I really had a hard time with Henry's work, I tried to do my best and to avoid reproducing the same scheme.

Webography

Bloch Joshua, *How to Design a Good API and Why it Matters*,
http://lcsd05.cs.tamu.edu/slides/keynote.pdf

Appendix 1
Server API

Server API

Version: 1.0.0

1 Introduction

Welcome to the FoxyTag API. This file explains the rules that a compatible client application has to follow. Please read carefully all the points and start working on your project only if you accept all the points mentioned in this document.

2 API

Please note that all parameters must be encoded in UTF-8.

2.1 Initialization

Each time the application is restarted and each time the username, the password or the language is changed, the client sends an HTTP GET request like:

url?cmd=init&username=*username*&password=*password*&client=*client*&version=*version*&language=*language*&platform=*platform*

- ***url***: the URL of the server. (Lab server: http://www.foxytag.com/php/dogetlab.php)
- ***username***: the username. Can be empty.
- ***password***: the password. Can be empty.
- ***client***: the client application code to obtain from FoxyTag.
- ***version***: version of the client application with 3 numbers, like "1.4.63".
- ***language***: ISO 639-1 code for the language (en = English, de = German...).
- ***platform***: platform and firmware of the mobile device, for instance "GT-I9100_2.3.3".

If the username or the password is empty, the user is anonym and is therefore not allowed to post data. Note that the lab server registers automatically any new username, but the production server requests a username/password pair to obtain from the registration process at http://www.foxytag.com.

The server answers like:

MSG
msg1
msg2
...

The messages (*msg1*, *msg2*, *...*) can be:

- **OK_INIT**: Initialization OK.
- **KO**: Failed for an unknown reason.
- **KO_SERVER**: Server is not available.

- **KO_URL**: Unknown command or malformed parameters.
- **KO_VERSION**: Wrong version of the client.
- **KO_LOGIN**: Wrong username or password.
- **ERR *error_description***: An error to show to the client. A '\t' is used as separator.
- **INF *some_information***: Information to show to the client. A '\t' is used as separator.
- **PUB *some_publicity***: A publicity to show to the client. A '\t' is used as separator.

There are no '\n' chars in a message (*error_description*, *some_information* or *some_publicity*). A new line is coded "\n" (2 chars). For instance, "Hello\nFoxyTag" should print "Hello" on the first line and "FoxyTag" on the second one.

2.2 Getting tags

To get the tags around a given position, the client sends a HTTP GET request like:

url?cmd=tagrequest&username=*username*&password=*password*&client=*client*&version=*version*&lat=*lat*&lon=*lon*

- ***url***, ***username***, ***password***, ***client***, ***version***: see above.
- ***lat***: the latitude in decimal degrees (5 digits after the coma) of the center of the circle.
- ***lon***: the longitude in decimal degrees (5 digits after the coma) of the center of the circle.

This request asks all the tags in a circle centered at the given latitude and longitude with a radius of 7250 meters. A new request is expected each time the client application is going to quit this circle but at the latest five minutes after the former connection.

The server answers either like it is done in the Initialization section, with messages that can be **KO**, **KO_SERVER**, **KO_URL**, **KO_VERSION**, **KO_LOGIN**, **ERR *error_description*** or **INF *some_information***, or with a list of tags like:

TAG
request
tag1
tag2
...

The *request* contains the latitude, the longitude and the radius of the circle. A '\t' is used as separator.

A tag (*tag1*, *tag2*, *...*) is written like:

lat lon kind heading

A '\t' is used as separator.

- ***lat***, ***lon***: position of the tag.
- ***kind***: the kind of tag: F = Fixed speed camera, FC = Fixed speed camera that has already been confirmed (or posted) by the user, M = Mobile speed camera, G = Ghost.
- ***heading***: an integer in [-1..360] U [1000..1360]. -1 means unknown heading. A heading between 0 and 360 (0° = North, 90° = East, 180° = South and 270° = West) means that the user posted the tag for the same direction. A heading between 1000

and 1360 is equivalent to heading-1000, but means that the user posted the tag from the opposite direction and his position is therefore a little bit less precise.

The returned list contains first the fixed speed cameras, then the mobile ones and finally the ghosts. A client application draws therefore first the ghosts, so that mobile speed cameras and fixed speed cameras are painted above them.

A ghost is created by the server when a mobile speed camera disappears. If there are often mobile speed cameras at a specific place or in its neighbourhood, the ghosts stay active (and are sent to the client) in order to signal this risky zone. Ghosts communicate between them, so it is possible to see a ghost a few hundred meters before the actual position of the mobile speed camera (since the later is not always exactly at the same position).

2.3 Sending, confirming or canceling a tag

Sending a new tag or confirming an existing one is identical for the client. It is the server that decides if it is a confirmation (the position is close to an existing tag) or a new one (there is no tag with similar heading in the neighborhood).

To send/confirm/cancel a new tag, the client sends an HTTP GET request like:

url?cmd=tagpost&username=*username*&password=*password*&client=*client*&version=*version* &lat=*lat*&lon=*lon*&kind=*kind*&heading=*heading*&speed=*speed*

- ***url***, ***username***, ***password***, ***client***, ***version***: see above.
- ***lat***: the latitude in decimal degrees (5 digits after the coma).
- ***lon***: the longitude in decimal degrees (5 digits after the coma).
- ***kind***: the kind of tag, F = Fixed speed camera, M = Mobile speed camera, C = Ask to cancel this tag.
- ***heading***: an integer in [0..360] U [1000..1360]. A heading between 0 and 360 (0° = North, 90° = East, 180° = South and 270° = West) means that the user posts the tag for the same direction. A heading between 1000 and 1360 is equivalent to heading-1000, but means that the user posts the tag from the opposite direction. For instance, a driver going northwards that tags a camera for the drivers going southwards will send 1180.
- ***speed***: an integer in [20..320]. It is the current speed of the user, in km/h.

The server answers like it is done in the Initialization section with messages that can be **KO**, **KO_SERVER**, **KO_URL**, **KO_VERSION**, **KO_LOGIN**, **OK**, **ERR** ***error_description*** or **INF** ***some_information***. The new message of this list (**OK**) is returned by default.

3 Lab server

The URL http://www.foxytag.com/php/dogetlab.php connects you to a lab server. Once your application is ready, we'll give you the URL of the main server.

Some particularities of the lab server:

- If you take username=$lab1, you get at initialization an error message, an information message (2 lines) and a publicity message (3 lines with a blank one).
- If you take username=$lab2, you get at initialization a **KO_SERVER** message.

- If you take username=$lab3, you get at initialization a **KO_VERSION** message.
- If you take username=$lab4, you get at tagrequest and tagpost an error message, an information message (2 lines) and a publicity message (3 lines with a blank one).

4 Rules

In addition to the API, an application must respect the following points. Some of these rules avoid that users tagging differently (because of their application) are excluded from the system by the other users.

- Your website promotes FoxyTag (link to http://www.foxytag.com, map with all the speed cameras, ...) and mentions clearly that the data comes from FoxyTag.
- The name of your application cannot contains the word "foxy" without our authorization.
- Your application is fully compatible with the API described above and all the rules described in this document. Your application behaves in a similar way as the FoxyTag Reference Implementation. If your application needs an exception, please write us your issue so that we can find a solution that doesn't harm the global system.
- You can freely access our lab server in order to make tests. However, any new version of your software must be validated by us before being allowed to access the main server. The application sent for validation must be exactly the one that will be publicly released, which means that it must point to the main server. Note also that the version number must be increased at each validation attempt and this number must be mentioned in the email as well.
- The application must do an "init" command (cmd=init) each time the application starts and each time the username, the password or the language is changed. An "init" is always followed by a "tagrequest" even if the last "tagrequest" has been done less than 5 minutes earlier.
- The application must download new tags (cmd=tagrequest) every five minutes or earlier, and post new information (cmd=tagpost) as soon as possible in order to be compatible with the current trust engine (off-line mode is not allowed).
- By default, only cameras that are in front of the driver, plus-minus 45°, and that have the same heading than the driver, plus-minus 45°, will launch an alarm and be painted on the screen. If a setting allows to change the default value and to show the "non-flashing" cameras as well, it is recommended to paint them differently.
- During a tag post (cmd=tagpost), the precision of the position must be under 35 meters, the precision of the heading must be under 7° and the precision of the speed must be under 7 km/h. The speed and the heading must be taken from the GPS device (not computed from two previous points and not taken from the compass) and it is not allowed to tag under 20 km/h. The application must be designed so that it is easy to tag precisely and easy to see where the tags are according to the user's position.
- When a posting action (cmd=tagpost) needs several interactions (touching the screen a first time, then selecting the kind of camera, then selecting its direction), the position is recorded when the first touch is done and the highly visible message "Position memorized" is shown to the user.
- Even when FoxyTag is integrated in a bigger system (like a navigation tool), the users must still register at FoxyTag in order to obtain a username and a password.

5 Rules to show to the client

The following rules must be visible in the help screen and also be shown to the client each time he enters or changes his username or password.

*****
IMPORTANT: Respecting the following rules allows you to increase your trust links with others and to benefit therefore from more reliable information. It avoids you also to be excluded from the system by people tagging differently.

1. A speed camera must be tagged or confirmed when you are as close as possible to it (not already when you see it). Otherwise there will be a second tag for the same camera and other users will decrease your trust links.
2. A camera inside a tunnel or close to a tunnel exit (typically less than 10 seconds after the exit) must be tagged at the tunnel entry.
3. During the tagging process, the position is recorded when you touch the screen for the first time (message "Position memorized" is shown), so you have as much time as you need to finish the tagging process.
4. It is useless to confirm several times the same speed camera. In some particular situations this can even be bad for your trust links.
5. In case of a doubt (is there a camera?) it is better not to tag than tagging wrong.
6. If there can be a doubt whether a camera can flash in both directions or not, it is better to tag for both directions.
7. A traffic light camera is tagged and treated like a speed camera.

*****

An application can also show dynamic messages to help the user to better respect the rules. For instance:

- In the main screen write "Touch the screen to add/confirm a tag when you are as close as possible to the camera".
- Show "Tagging only possible when speed > 20 km/h" if the user tries to tag while driving slower than 20 km/h.
- Show "GPS precision too weak" when the user tries to tag with low precision.

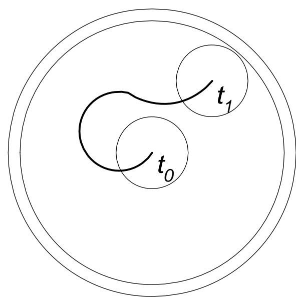
6 Notes

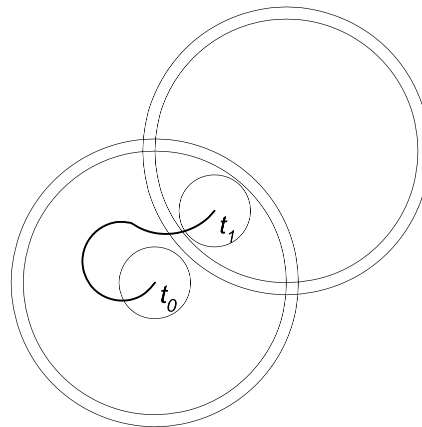
- When a user tries to cancel a tag, only the closest tag with a heading difference less than 45° will be affected, and only if this tag is closer than 75 meters.
- The heading in the tag indicates the "dangerous" direction, the direction where a driver could get a fine if he is speeding. It has nothing to do with the orientation of the camera, that can record either the front or the back number plate.

7 Optimizing connections (Optional)

The following algorithm can be used in order to reduce the number of connections to the server: Let's call protected zone (PZ) the area that contains the downloaded tags. A new connection is then made only every given delay (here it is 5 minutes) or if the user requests tags that are outside of the protected zone. A protected zone is defined by 5 main attributes:

- **pzLat** and **pzLon**: Two numbers, the latitude and the longitude of the center of the PZ. A PZ is always a circle. A user can give its current position if he wants to see the tags around him, but can also give coordinates of another point to see the tags in another place.
- **radiusOut**: This is the radius of the external circle of the PZ. It defines the area where the tags have been downloaded.
- **radiusIn**: This is the radius of the internal circle of the PZ. This value is used as a trigger to ask a new connection to the server. Since connections to a remote server are not instantaneous, this internal circle allows the application to update its protected zone before the user exits the former PZ.
- **timeout**: This is the maximum delay between two connections to the server. Even if the user doesn't quit its PZ, a connection is made time to time in order to guaranty the freshness of the information.





Moving in the same PZ | Computing a new PZ

The first time tags are requested, the algorithm will compute a PZ centered on the current position (or on the position where the tags are requested). But in order to limit the communications with the server, the next time a request is made, the PZ will be shifted according to the last moves of the user. In the left figure we see an example. The user starts his application at time t_0. The little circle represents the radius of visibility (simply called radius), or the area where the tags are visible on the mobile device. The big inner circle is the internal circle of the PZ (called radiusIn), which acts as a trigger (when the user requests tags that are outside, a new PZ is computed). The big outer circle is the external circle of the PZ (called radiusOut), which is the area where the tags are actually present in the memory of the mobile device. While the user moves close to the center of the PZ, the tags are already in memory. If the user is still at the center of the PZ when the timeout is reached, a new connection is made to the server in order to update the tags, but the PZ stays at the same position (centered on the user). But if the user moved away, then the new PZ will be shifted in the same direction. For instance, if our user reaches the inner circle of the PZ at time t_1, a new PZ is requested and the latter will be shifted like it is shown in the right figure. The distance of the shift is proportional of the distance that the user moved since last update, so if he didn't move, the shift is null (and the new PZ is centered on the current position).

Appendix 2
Foxy Challenge Trip

Foxy Challenge Trip

Version: 2012-02-24

Contents

1 Introduction....1
2 Capabilities testing....2
3 Test Area....2
4 General rules....4
5 Test Cases....5
5.1 GPS good, and data OK....5
5.2 GPS poor, and data OK....7
5.3 GPS KO, and data OK....8
5.4 GPS good, and data KO....9
5.5 GPS poor, and data KO....10
5.6 GPS KO, and data KO (for example in a long tunnel)....11
5.7 Any connectivity status....12
6 Tests Radars/Places location....13

1 Introduction

This document describe all test the application must pass in order to succeed the FoxyChallengeTrip.
The FoxyChallengeTrip is a “framework”, an environment where your application will be put and tested. If it succeed, it ensure the good quality of your application on the opposite, it means that you application does not fill the quality requirements of foxy tag, and must be modified.

2 Capabilities testing

The application must let the user log in, or redirect to the foxyTag website if he needs to register.
The application must let the user specify his language (Possibility to take the system language, but the language must be shown, and modifiable)
The user must be able to change the app metrics.

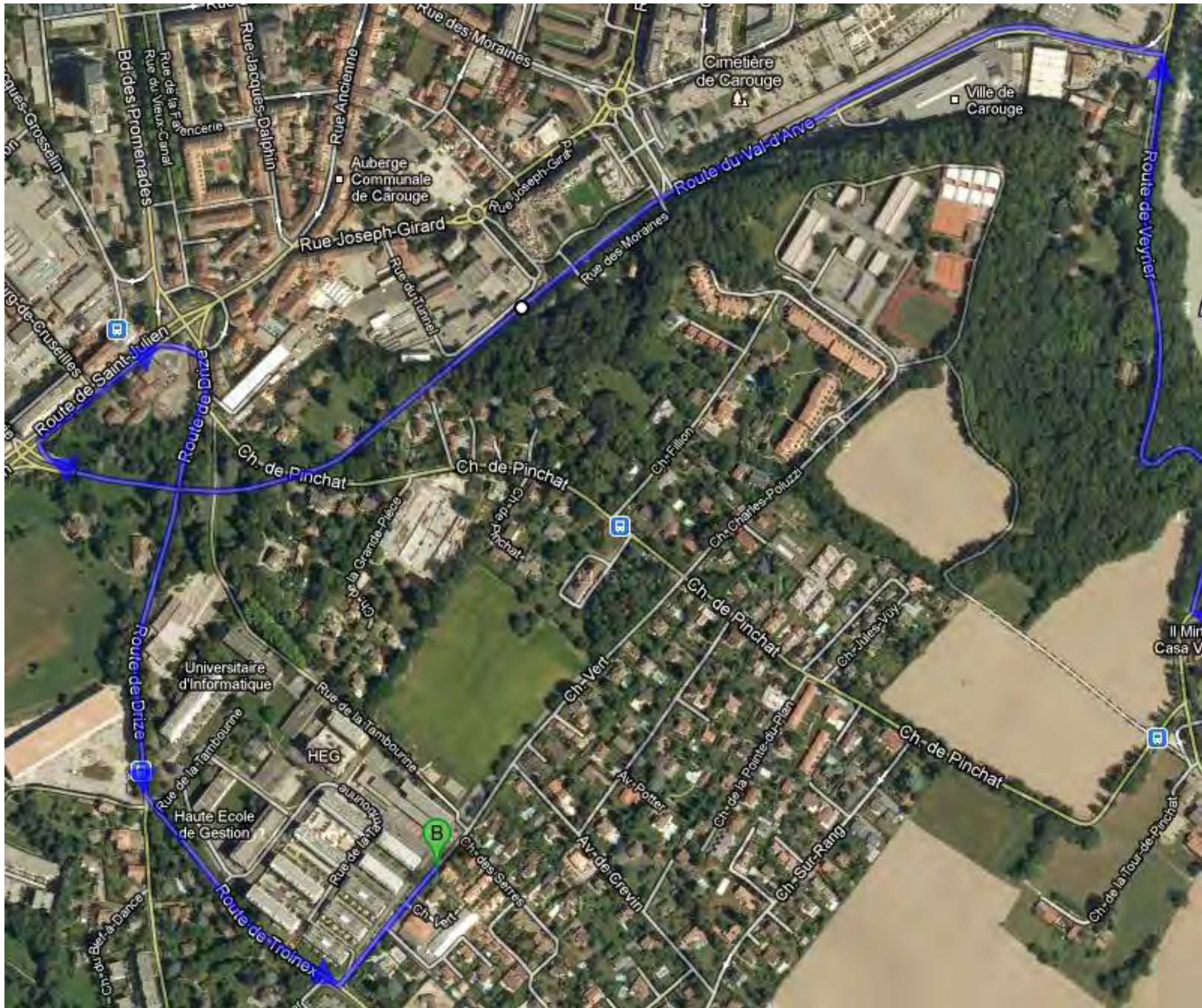
3 Test Area

In order to test the application, some radars must be strategically placed.
They will be placed along my home-work route witch is Reignier (France) to Carouge (Switzerland).
The data will be received from the 3G french network. The good point is that in Switzerland, the Network will sometimes drop, and we will be able to observe what's happening.

There is several places on the trip or around it to test critical points of the applications.

Here is the main path we will use in order to stress the application.

Zoom of the end of trip:



There is a tunnel near to the arrival. We will take it to test the reaction of the application in this case.

Here are the indications in order to fully make the foxyChallengeTrip :

Departure: Impasse des fauvettes 74930 reignier:

1. Prendre la direction **ouest** sur **Imp. des Fauvettes** vers **Rue des Érables**

100 m

2. Prendre à droite sur **Rue des Érables**

71 m

3. Tourner à droite pour rester sur **Rue des Érables**

87 m

4. Prendre à gauche sur **Route de l'Eculaz/D19**

280 m

5. Prendre la 2e à droite et rester sur **Route d'Annemasse/D2**
Continuer de suivre Route d'Annemasse
Radar automatique à 2,3 km

3,9 km

6. Continuer sur **Route de Reignier/D2**

1,9 km

7. Prendre à gauche sur **Route du Salève/D906A**

11 m

8. Tourner à droite pour rester sur **Route du Salève/D906A**

1,3 km

9. Prendre à gauche sur **Route de Saint-Julien/D1206**
Continuer de suivre D1206
Traverser le rond-point

2,8 km

10. Prendre légèrement à droite sur **Rue du 18 Août 1944/D1206**
Entrée sur le territoire : Suisse

25 m

11. Continuer sur **Route du Pas-de-l'Echelle**

600 m

12. Au rond-point, prendre la **2e** sortie sur **Route de l'Uche**

280 m

13. Au rond-point, prendre la **1ère** sortie sur **Route de Veyrier**
Traverser le rond-point

2,4 km

14. Tourner à droite pour rester sur **Route de Veyrier**

700 m

15. Prendre la 1re à gauche et rester sur **Route du Val-d'Arve**

1,4 km

16. Prendre à droite sur **Route de Saint-Julien**

190 m

17. Prendre la 1re à droite et rester sur **Route de Drize**

550 m

18. Prendre légèrement à gauche sur **Route de Troinex**

300 m

19. Prendre à gauche sur **Ch. Vert**

170 m

Arrival : Chemin Vert.

4 General rules

The application must show every radar that should/could be in the user's field of vision (+45° and -45° of the heading of the car), and at least 1 kilometres away. The application must make some sound as a radar get closer (depending of speed -> 7 seconds before reaching it). If a radar have already make a sound, then it should not bip again (even if it quit the field of vision and enter into again. For example highway's exits that often do circles).
Ghosts and Mobile camera must make a sound if we are close to them, even if it does not aim into our direction

5 Test Cases

Here is the list of tests that must be successful for the application to pass the FoxyChallengeTrip.

5.1 GPS good, and data OK

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The accuracy of the application	Drive on a route parallel to one with radar	The application should not warn you (if you have good gps signal)	Refers radar number 1	
2	The accuracy of the application	Drive on a route, and come across a radar with a heading plus or less 35% of your heading	??	Refers to radar number 2	
3	The good functionality of the on-board storage of the radars	Drive into an area that should be stocked into your phone	The application should show you the radars as you meet some		
4	The average speed camera function	The app must indicate you if must decrease your speed cuise, and indicate you when you quit this average speed area	If you quit the road, the app should not warn you exept if you're approaching to the last camera using an other route than the one was tagged(if your average speed is excesive, and if you pass the first camera)	The app should also explain you how to use the average speed camera function on first use. Refers to radars #4	

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The good storage of taggings	Tag some radars while you are offline	Once you're online, the app should tag on the server the locally stored radars		
3	The good integration of average speed cameras	Post the first camera as average speed	You should return to radar view once you press a second time the screen in order to entrer the last speed camera	Try for average speed camera that contains different speed sections	

5.2 GPS poor, and data OK

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The accuracy of the application	Drive on a route parallel to one with radar	The application should not warn you	See radar number	

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The respect of the Foxy Rules	While driving, post a radar on a known place	The radar must be placed at the place you were when you taped the screen for the first time		
2	The good programming of an application				

5.3 GPS KO, and data OK

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The accuracy of the application	Drive on a route parallel to one with radar	The application should not warn you	See radar number	

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The respect of the Foxy Rules	While driving, post a radar on a known place	The radar must be placed at the place you were when you taped the screen for the first time		
2	The good programming of an application				

5.4 GPS good, and data KO

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The accuracy of the application	Drive on a route parallel to one with radar	The application should not warn you	See radar number	

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The respect of the Foxy Rules	While driving, post a radar on a known place	The radar must be placed at the place you were when you taped the screen for the first time		
2	The good programming of an application				

5.5 GPS poor, and data KO

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The accuracy of the application	Drive on a route parallel to one with radar	The application should not warn you	Refers to radar number 1	
2	The good programing of the application	Drive on a route and approach to a 2 side radar (shown as 2 differents radar by the API)	The application must show only one radar	Refers to radar number	

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The respect of the Foxy Rules	While driving, try to post a radar	You should not be able to post a radar when your GPS accuracy is poor.		

5.6 GPS KO, and data KO (for example in a long tunnel)

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The reaction of the app	Drive until you lost the signal	The application should zoom out and show you the radars (in a radius of 7 kilometers) around your last know position, and your last known heading		

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The good programming of an application	While driving ,try to post a radar	You should be asked if you are in a tunnel.	If you answer that you are in a tunnel, the tag process will continue, and the tag must be placed where you lost good gps Signal. Otherwise, it will not tag .	

5.7 Any connectivity status

Showing Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	The user interface	Drive, and verify that there is different representations of confirmed or not confirmed radars	The speed camera confirmed and unconfirmed must be displayed differently		

Tagging Radars :

Test number	What will be tested	Test instructions	Expected result	Remarks	Effective result
1	Good communication with server	While driving ,try to post a radar	The radar should be correctly posted into the sever at the position of the first touch	.	

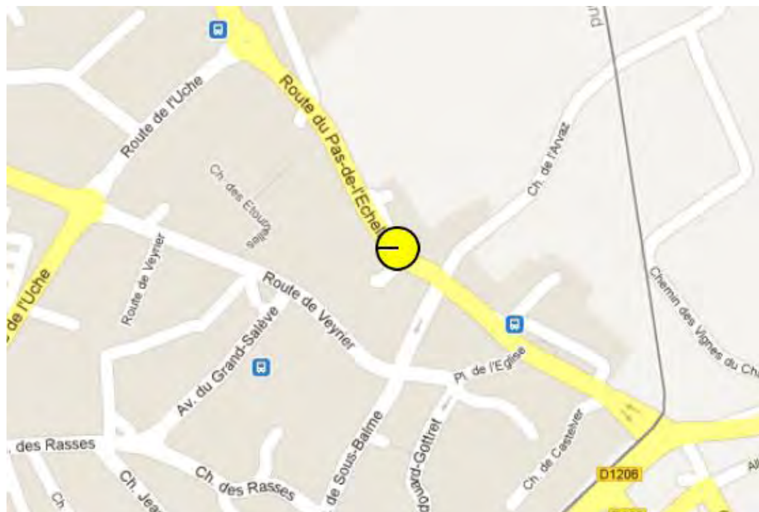
6 Tests Radars/Places location

Radar 1 :

Latitude : 46.17033
Longitude : 6.20797
Heanding : 235
Kind : Fixed

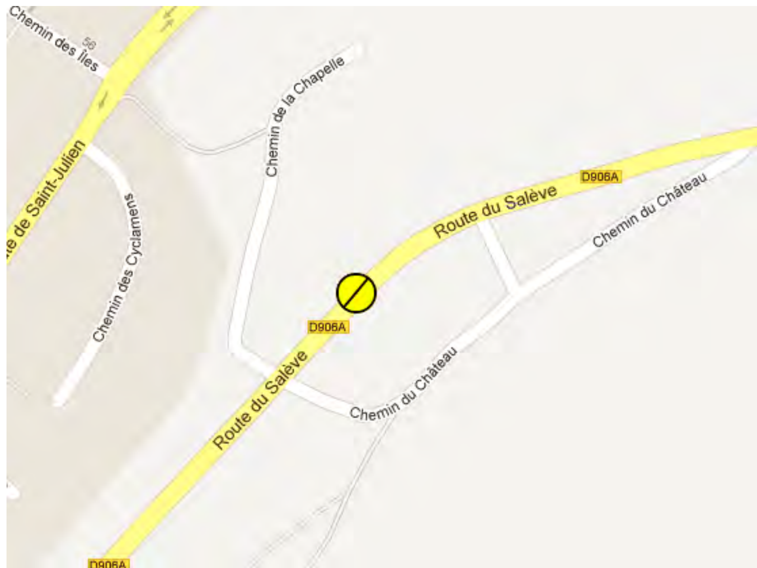
Radar 2 :

Latitude : 46.16816
Longitude : 6.18487
Heading : 270
Kind : Fixed
Speed : 80



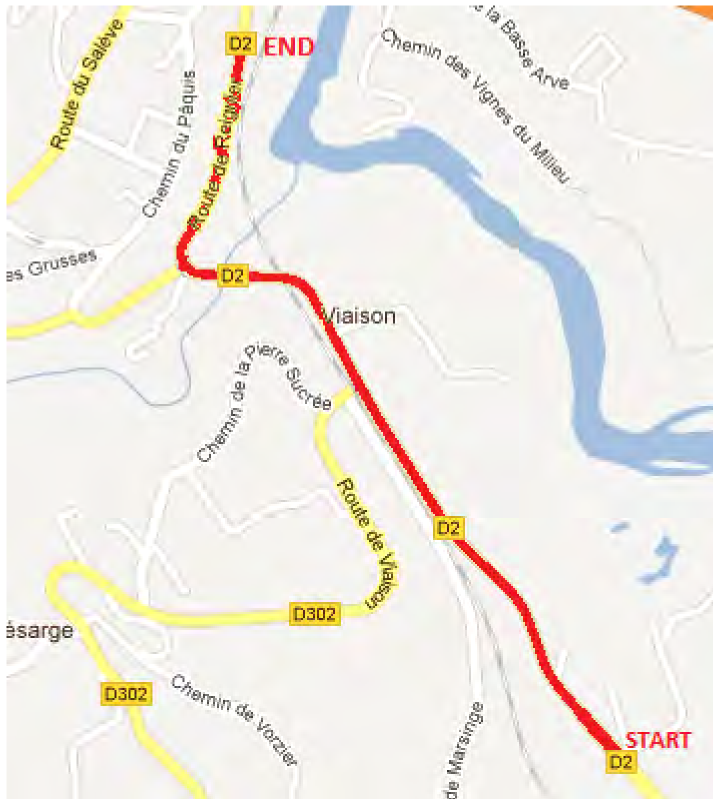
Radar 3 :

Latitude : 46.17743
Longitude : 6.22807
Heading : 40 & 220
Kind : Fixed
Speed : 80



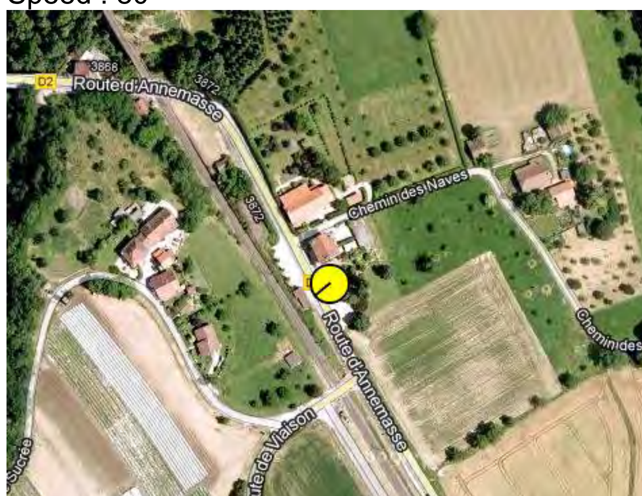
Radar 4 : (average speed)

Latitude_start : 46.1546
Longitude_start : 6.24871
Latitude_end : 46.16527
Longitude_end : 6.2406
Kind : Fixed



Radar 5 :

Latitude : 46.1607
Longitude : 6.24286
Heading : 230
Kind : Fixed Confirmed
Speed : 80



Radar 6 :

Latitude : 46.15153
Longitude : 6.25355
Heading : 325
Kind : Mobile
Speed : 80

Radar 7 :

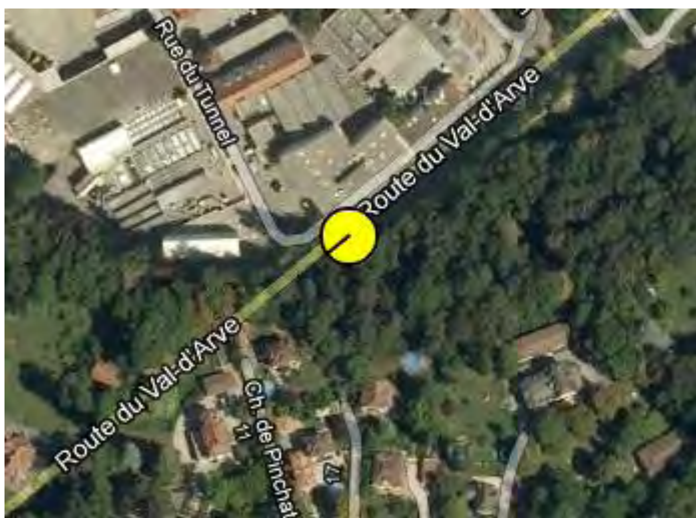
Latitude : 46.14547
Longitude : 6.26554
Heading : 280
Kind : Mobile
Speed : 80

Radar 8 :

Latitude : 46.14547
Longitude : 6.26554
Heading : 280
Kind : Ghost
Speed : 80

Radar 9 (at the enter of a tunnel):

Latitude : 46.17964
Longitude : 6.1429
Heading : 230
Kind : Fixed
Speed : 80



These are some of the most important radars to test.

There is also a lot of other radars that can be tested as shown in the following image:

Le Bout-du-Monde
Sierne
Étrembi
Vétraz-Monthoux
Pinchat
Pinchat
la Tour
Route de Veyrier
Veyrier
Route de Troinex
Marsillon
Ch. des Marais
Troinex
Cloche
Monnetier-Mornex
Arthaz-Pont-Notre-Dame
Sous
Bossey
Autoroute Blanche
Bossey
Evordes
Collonges-sous-Salève
Route de Bonneville
Route du Salève
Route des Bornes
D1206
D41A
D145
D48
D302
D15
D2
D202
D185
D1205
E25
E712
A40

Appendix 3
User Guide

AndroTag API User Guide

Version: 2012-06-08

Contents

1 Introduction 1
2 Quick start Guide 2
2.1 Requirements 2
2.2 Few things to know 2
2.3 The first line to use the API 2
2.4 Display the SpeedCamsView 3
2.5 Initiate the API. 4
2.6 Subscribe to the events (Optional) 5
2.7 Conclusion 5
3 Understand the API 6
3.1 General concept 6
3.2 Events system 8
3.3 The server communication 8
4 Classes description 10
4.1 SpeedCamsView 10
4.2 AndroTag 11
5 Conclusion 12

1 Introduction

This document will help you using the FoxyTag API. It contains some descriptions and examples about how to use the API. If you plan to use the API without modifying it, this guide if for you! If you want to modify the API, it is highly recommended that you also read the Developer Guide.

The API currently contains the architecture in order to support sections (average speed cameras), but it is not yet implemented. If you try to use API methods that refer to Sections, you will obtain no results.

In this document I will use terms such as tag, radar, and speed camera. This all refers to the same concept: a speed camera.

2 Quick start Guide

2.1 Requirements

It is highly recommended that you read the Server API document before starting to read this one.
Firstly you must have created a new Android application project.
After that, you have to copy the sources of the provided API into the sources of your project.
All the API sources should be into the package “ch.unige.androtag”.
You also have to copy into your resources folder the provided default alarm sound (do not rename it!) this sound must be located under res/raw/. If necessary, you can create the raw folder in your IDE or in your file explorer.
A minimal knowledge of Android is required. If you haven’t yet developed any applications, it would be a good idea to do some Android development tutorials.

2.2 Few things to know

Now you are ready to start. Yes that’s so simple.
With a few lines of code you will be able to see the speed cams around you if there are some.

Before we start to code, you have to understand that the API is a very autonomous system. It knows itself when it is necessary to update tags, it manages the communication with the server and a lot of more complicated stuff that is simplified for you.
The API has only one possibility to communicate with you: the events.
The API has his own event system and launches some specific events.
One last thing before we start: The API provides you with a graphical component: “SpeedCamView”. This component needs the API in order to function, but the API can function without this component. That means that if the view provided does not suit your requirements, you can just drop it and develop your own view.

2.3 The first line to use the API

Now you know enough, and we can start coding.

In this section, we will explain you how to instantiate the API in order to be able to use it.
In fact this is very simple. Just type this line:

```
AndroTag api = AndroTag.getInstance(this);
```

This is done. With this single line, your API is ready to be used.
With this line, the API will automatically subscribe to the GPS and the GPS activity icon in the Android notification bar should appear. Once the GPS fix is done, you can access GPS information using the API. Refer to “classes description->AndroTag” section or javadoc.

"Why do we need to pass our Activity as parameter?"
It is because the API takes care or the GPS for you. But in order to subscribe to GPS updates, an Activity is needed. This is the only use of this parameter.

It is your responsibility to make sure that the GPS is enabled. The API does not provide this functionality.

2.4 Display the SpeedCamsView

Here are two solutions:

- You define your component in the layout
- You add your component to the layout during runtime

We will explain only the first solution because if you understand the second one and want to do it that way, means that your developer's skills are sufficient and you don't need explanation.

Firstly, place this xml code into your layout code:

```
<ch.unige.androtag.SpeedCamsView
   android:id="@+id/scannerView"
   android:layout_width="wrap_content"
   android:layout_height="fill_parent">
</ch.unige.androtag.SpeedCamsView>
```

Secondly, add this code to the onCreate() method of your activity. It will give you a reference to the SpeedCamsView instance.

```
SpeedCamsView scannerView = (SpeedCamsView)findViewById(R.id.scannerView);
```

Thirdly, you have to register the scannerView to the event system:

```
api.addAndroTagListener(scannerView);
```

"Why must we register the scannerView to the event system?"
As the API can function without the SpeedCamsView, it has no reference to it, and cannot do it for you. This is important, because this view use events such as the "positionChanged" in order to update the tags position on the screen.

2.5 Initiate the API.

As you have read in the Server API document, prior to any other communication with the server, we need to do an initialization. If you do not do that, the API will not be able to do anything else than providing you GPS updates.

Let see how you can initialize the communication with the server by using the API.
This is done with a single line:

```
api.init("test","test","androtag","0.0.1","fr","android",true);
```

You can find full description of this function in the provided javadoc, but here is a summary of the parameters:

1st parameter: username
2nd parameter: password
3rd parameter: application name
4th parameter: application version
5th parameter: language
6th parameter: platform
7th parameter: lab server

The last parameter is used to specify which server you want to use: the production server or the lab server.

This function returns a boolean: true if the initialization is successful or false if the initialization failed.

If the initialization is successful, the API will automatically (without any action needed) download tags around you. You can drive where you want, the API will automatically take care of the whole system, and keep a buffer of tags around you. These tags are displayed around you. By default the SpeedCamsView display tags around you in a radius of 1000 meters. If you want to change this, you can, but refers on how to do in the Classes description below, and more precisely in the part that explain this component (SpeedCamsView).

Really, isn't that beautiful? With a few lines of code, you already see the speed cameras around you.
You will see below that the API is highly customizable. It is highly recommended that you instantiate the API, create and reference the SpeedCamsView, customize what you want, and only afterwards do the initialization.

It will work if you make the customization after the initialization, but the changes will be applied with a little delay (less than 1 second I promise you).

"My initialization always fails. How to know what is wrong?"
The server provides explanation about what is wrong. You can basically subscribe to the "serverAnswer" event (Refers to "Subscribe to the events section"). If you do not receive any "serverAnswer" event, please verify your mobile internet connection.

It is your responsibility to take care that the initialization is successful. For example, if there is no network you should do a loop that calls the init method while the result of the initialization is false. Remember that once the initialization is successful, the API will do nothing else than GPS stuff. According to the Server API document, you must also call this method each time the username, password or language is changed.

2.6 Subscribe to the events (Optional)

In order to be able to receive news from the API, you must subscribe to the event system. Each subscriber of the event system must satisfy some requirements; this is why an interface is provided. Your activity must implement the interface `AndroTagListener` and its five methods. You can leave them empty, but if you want to do some special actions when an event is received, place your code in the corresponding method.
For example if you want to write in the console "New GPS location" each time a new position is received, just do that :

```
public void positionChanged() {
    System.out.println("New GPS location");
}
```

Now you are ready to receive the events, but we must subscribe to them.
This is done by typing this line (for example in you *onCreate* function):

```
api.addAndroTagListener(this);
```

Perfect! You should now receive all the updates from the API.

2.7 Conclusion

To conclude, I hope you have figured out that it is very simple to use the API, but to simplify, I will write all the lines you must place in your application in order to obtain a first minimalistic speed camera warning application:

To place in your layout file:

```
<ch.unige.androtag.SpeedCamsView
   android:id="@+id/scannerView"
   android:layout_width="wrap_content"
   android:layout_height="fill_parent">
</ch.unige.androtag.SpeedCamsView>
```

In the onCreate method of your application:

```
AndroTag api = AndroTag.getInstance(this);
SpeedCamsView scannerView = (SpeedCamsView)findViewById(R.id.scannerView);
api.addAndroTagListener(scannerView);
api.init("test","test","androtag","0.0.1","fr","android",true);
```

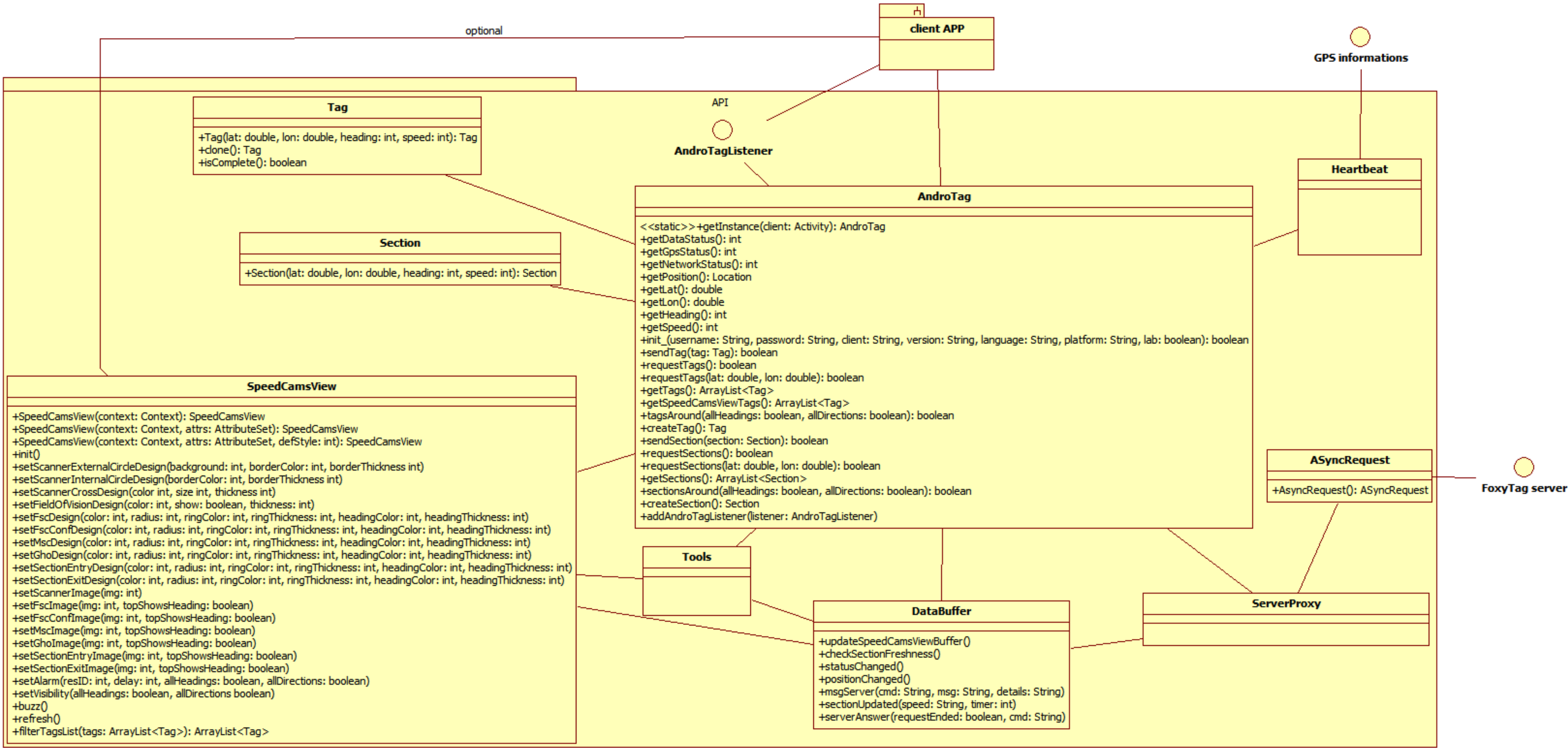
Save, Build, Run, and enjoy!

3 Understand the API

In this part you will be aware of what you can do with the API and how to configure it.

3.1 General concept

This API was designed to be very simple to use. You have two main entry points: The speedCamsView and the AndroTag class.
You should not have to access other classes of the package API, and anyway, most of these classes' methods are protected, so you do not have access.
Take a look at the simplified class diagram in the next page:


client APP
optional
GPS informations
API
AndroTagListener
Tag
+Tag(lat: double, lon: double, heading: int, speed: int): Tag
+clone(): Tag
+isComplete(): boolean
Section
+Section(lat: double, lon: double, heading: int, speed: int): Section
AndroTag
<<static>>+getInstance(client: Activity): AndroTag
+getDataStatus(): int
+getGpsStatus(): int
+getNetworkStatus(): int
+getPosition(): Location
+getLat(): double
+getLon(): double
+getHeading(): int
+getSpeed(): int
+init_(username: String, password: String, client: String, version: String, language: String, platform: String, lab: boolean): boolean
+sendTag(tag: Tag): boolean
+requestTags(): boolean
+requestTags(lat: double, lon: double): boolean
+getTags(): ArrayList<Tag>
+getSpeedCamsViewTags(): ArrayList<Tag>
+tagsAround(allHeadings: boolean, allDirections: boolean): boolean
+createTag(): Tag
+sendSection(section: Section): boolean
+requestSections(): boolean
+requestSections(lat: double, lon: double): boolean
+getSections(): ArrayList<Section>
+sectionsAround(allHeadings: boolean, allDirections: boolean): boolean
+createSection(): Section
+addAndroTagListener(listener: AndroTagListener)
Heartbeat
SpeedCamsView
+SpeedCamsView(context: Context): SpeedCamsView
+SpeedCamsView(context: Context, attrs: AttributeSet): SpeedCamsView
+SpeedCamsView(context: Context, attrs: AttributeSet, defStyle: int): SpeedCamsView
+init()
+setScannerExternalCircleDesign(background: int, borderColor: int, borderThickness int)
+setScannerInternalCircleDesign(borderColor: int, borderThickness int)
+setScannerCrossDesign(color int, size int, thickness int)
+setFieldOfVisionDesign(color: int, show: boolean, thickness: int)
+setFscDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setFscConfDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setMscDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setGhoDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setSectionEntryDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setSectionExitDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setScannerImage(img: int)
+setFscImage(img: int, topShowsHeading: boolean)
+setFscConfImage(img: int, topShowsHeading: boolean)
+setMscImage(img: int, topShowsHeading: boolean)
+setGhoImage(img: int, topShowsHeading: boolean)
+setSectionEntryImage(img: int, topShowsHeading: boolean)
+setSectionExitImage(img: int, topShowsHeading: boolean)
+setAlarm(resID: int, delay: int, allHeadings: boolean, allDirections: boolean)
+setVisibility(allHeadings: boolean, allDirections boolean)
+buzz()
+refresh()
+filterTagsList(tags: ArrayList<Tag>): ArrayList<Tag>
Tools
DataBuffer
+updateSpeedCamsViewBuffer()
+checkSectionFreshness()
+statusChanged()
+positionChanged()
+msgServer(cmd: String, msg: String, details: String)
+sectionUpdated(speed: String, timer: int)
+serverAnswer(requestEnded: boolean, cmd: String)
AsyncRequest
+AsyncRequest(): AsyncRequest
ServerProxy
FoxyTag server

As you can see, all classes are connected to Androtag class, because it is the Façade of the API. For example when you do an init, like $api.init(\dots)$, the AndroTag class will only forward this command to ServerProxy. Another example: When you ask androTag for DataStatus. Such as $api.getDataStatus()$, the class will only forward the command to DataBuffer. In fact you can see the AndroTag class as a router. It forwards your questions to the classes that have the answer.

3.2 Events system

Because the AndroTag class is in the center of the API, it is its responsibility to take care of the subscription and dispatch of events.

All kind of object you produce can subscribe to these events. In order to be able to do that, they must implement the AndroTagListener interface. After that, you can subscribe by doing this:

$api.addAndroTagListener(\boldsymbol{YOUR_OBJECT});$

Here is the content of the AndroTagListener interface:

`msgServer(String cmd, String msg, String details)`
Called each time the server sends a message.

`positionChanged()`
Called each time the current position changed.

`sectionUpdated(String speed, int timer)`
Called each time the data about the current section is updated.

`serverAnswer(boolean requestEnded, String cmd)`
Called each time the server answer to a request or the request timed out.

`statusChanged()`
Called each time that the status of the data, the GPS or the network changed.

3.3 The server communication

There are two different behaviors for the server communication:

- The init is a blocking method. That means that when you call it, your UI will be blocked until the server answer or the request times out. So take care when you use it. It might be necessary to call this method on a different thread. We decided to do it that way, because when we receive the Boolean result of the init method, we know instantly if it was a success or not. As you know, the init must be correctly done in order to automatically launch the API mechanic. So it is a critical call.
- The other communication methods are parallels ones. For example: the method $requestTags()$.You can call these methods that usually start with “request” or “send”. You will receive a boolean as answer, but unlike init method, this boolean does not means that your request was successful. In this case, it only means that tha API was able to launch the request. Only one request cans be send at a time, and as long as this request has not ended or timed out, you will not be able to launch another request. For example, if you call two times $requestTags()$, the first one will return you true, but the second one will return you false. That means the first request has not ended.

"How can I know when my request ended?"
The API provides you an event for that:
$serverAnswer\ (boolean\ requestEnded, String\ cmd)$
The Boolean tells you if we got an answer form the server if true, or if the request timed out if false. The cmd parameter tells you what command was launched to the server. For example "tagpost" or "tagrequest".

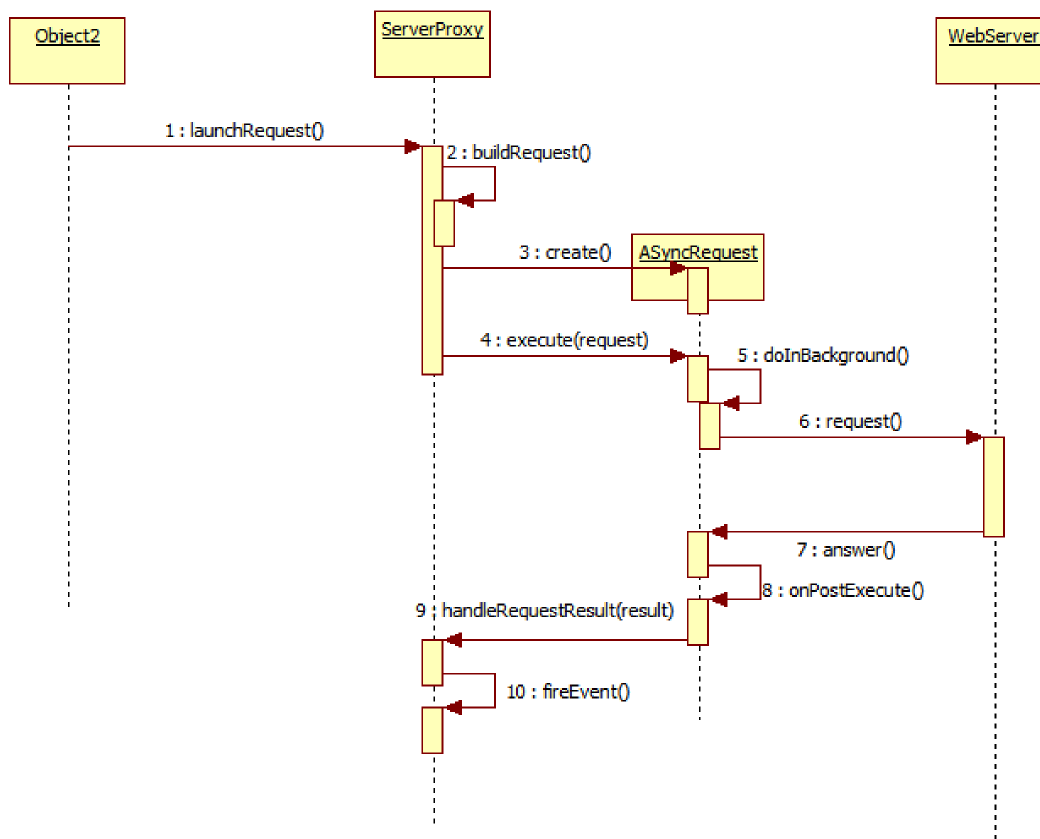
The API already use this system in order to know when we have received new tags (when we receive this event with requestEnded at true, and cmd at "tagrequest").
Try to never mix up the two concepts :

- Ask for new tags
- Get the new tags

For example for tags: If you want to obtain new tags, call $requestTags()$, and once you receive the good requestEnded event, you can call $getTags()$.
If you call $getTags()$ before having called $requestTags()$, you will just get null.

If you want for any reason to update the tags of the tagBuffer, you just have to call $requestTags()$ because when a "tagrequest" answer from server is received, the tagBuffer will automatically get the new tags just received.

Here is a simplified sequence diagram that explains how the parallel requests are done:

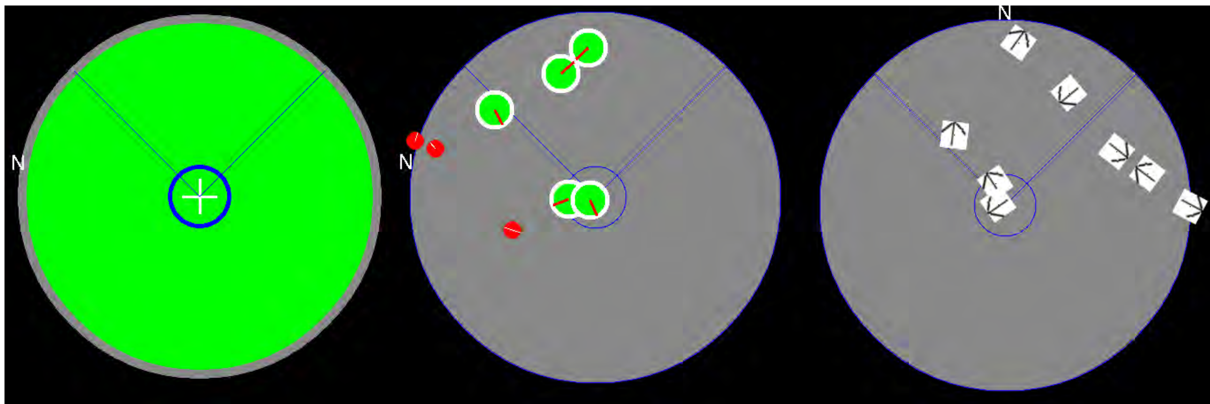


4 Classes description

Here you will find a brief description of the main classes of the API that you have access to. For more precise details, refer to the provided javadoc.

4.1 SpeedCamsView

The SpeedCamsView component is a highly customizable one. In the figure below, you can see some customization examples:



As explained before, you can customize almost all the user interface of this component. If instead of these circles (the radar without tags such as the left image above) you want to provide your images, you can do so, but you must satisfy these requirements:

When specifying an image for the scanner, the image must be a square. The center of the square will be the current user position, and tags will be displayed inside a circle with a radius of 45% of the image height/width. The center of the image will be the center of the circle.
The North sign will move around and outside this circle. The API do not require a specific size for this square such as 400*400px or 300*300px because all the images provided will be automatically resized in order to perfectly fit the screen, so all sizes are accepted.

As of today, the SpeedCamsView can only display normal speed camera and not sections (average speed camera), because this functionality is not yet supported by the API.
These are the four speed camera types:

- Fixed
- Confirmed fixed
- Mobile
- Ghost

For each kind of speed camera you can specify your own design, change color, thickness, display heading or not, and much more!
Like the radar, you can also specify images. But in this case, it is not necessary that the image is a square, and the image is not resized, so provide the good sizes (there is no default “good size” because all screens have different resolution, but just try and find the best size).

The other customizable thing is that you can specify which kind of radar should be painted on screen, and which ones should buzz (raise the alarm). These two settings are independent, and a speed camera that is not displayed on screen can buzz.
You can chose to display only tags that have same heading as you (plus minus 45 degrees), or/and tags that are in front of you (plus minus 45 degrees) or all the tags. When a tag is painted on the screen, it will stay on the screen even if it does not satisfy anymore the specifications (heading, direction …).

This component is also responsible for making a sound when a speed camera is close. The delay to make the sound before we reach the speed camera is customizable like the sound itself. A tag can buzz only one time. If the tag quit the screen, and come back, then it can buzz again.

The API will make a sound, but will not verify if the sound is enabled, or will not check the sound volume. You have to verify the sound yourself. It would be a good idea to be able to set the volume for the alarm, and to change the global volume to this configured volume level during the alarm, and to set the volume to the previous level when the alarm is done. This functionality can only be implemented in the API, because the API has currently no way to inform you that it will make a sound. Currently this functionality is not implemented. Maybe in a future revision of the API. but as you have the sources, you can add this functionality by yourself.

4.2 AndroTag

The AndroTag is the main entry point for the API. With this class, you can have access to lot of information: your current location, the data, GPS, network status, the tags around you, and much more! Take a look at this class representation:

```
AndroTag
--------------------------------------------------------------
<<static final>>+DATA_STATUS_GREEN: int = 1
<<static final>>+DATA_STATUS_ORANGE: int = 2
<<static final>>+DATA_STATUS_RED: int = 3
<<static final>>+GPS_STATUS_GREEN: int = 1
<<static final>>+GPS_STATUS_ORANGE: int = 2
<<static final>>+GPS_STATUS_RED: int = 3
<<static final>>+NETWORK_STATUS_FREE: int = 1
<<static final>>+NETWORK_STATUS_BUSY: int = 2
--------------------------------------------------------------
<<static>>+getInstance(client: Activity): AndroTag
+getDataStatus(): int
+getGpsStatus(): int
+getNetworkStatus(): int
+getPosition(): Location
+getLat(): double
+getLon(): double
+getHeading(): int
+getSpeed(): int
+init_(username: String, password: String, client: String, version: String, language: String, platform: String, lab: boolean): boolean
+sendTag(tag: Tag): boolean
+requestTags(): boolean
+requestTags(lat: double, lon: double): boolean
+getTags(): ArrayList<Tag>
+getSpeedCamsViewTags(): ArrayList<Tag>
+tagsAround(allHeadings: boolean, allDirections: boolean): boolean
+createTag(): Tag
+sendSection(section: Section): boolean
+requestSections(): boolean
+requestSections(lat: double, lon: double): boolean
+getSections(): ArrayList<Section>
+sectionsAround(allHeadings: boolean, allDirections: boolean): boolean
+createSection(): Section
+addAndroTagListener(listener: AndroTagListener)
```

In the class representation above, only public methods are represented, because you only have access to them.
For a more complete description of this class and its methods, refer to the javadoc.

5 Conclusion

You have reached the end of this document, and if you read it all, you should have a good comprehension of the API behavior, and what you can do with it. This API is designed in order to simplify your development of a FoxyTag client. I truly hope it reach its goal, and hope you will have pleasure to use it.

Appendix 4
Developer Guide

AndroTag API Developer Guide

Version: 2012-06-08

Contents

1 Introduction....2
2 Class diagram 2
3 Classes description 4
3.1 AndroTag 4
3.1.1 Objective 4
3.1.2 Responsibilities 4
3.1.3 Graphical representation 5
3.2 AndroTagListener 5
3.2.1 Objective 5
3.2.2 Graphical representation 6
3.3 ASyncRequest 6
3.3.1 Objective 6
3.3.2 Responsibilities 6
3.3.3 Graphical representation 6
3.4 DataBuffer....6
3.4.1 Objective 7
3.4.2 Responsibilities 7
3.4.3 Graphical representation 7
3.5 HeartBeat....8
3.5.1 Objective 8
3.5.2 Responsibilities 8
3.5.3 Graphical representation 8
3.6 Section....8
3.6.1 Objective 8
3.6.2 Responsibilities 9
3.6.3 Graphical representation 9
3.7 ServerProxy 9
3.7.1 Objective 9
3.7.2 Responsibilities 9
3.7.3 Graphical representation 10
3.8 SpeedCamsView 10
3.8.1 Objective 10
3.8.2 Responsibilities 10
3.8.3 Graphical representation 11

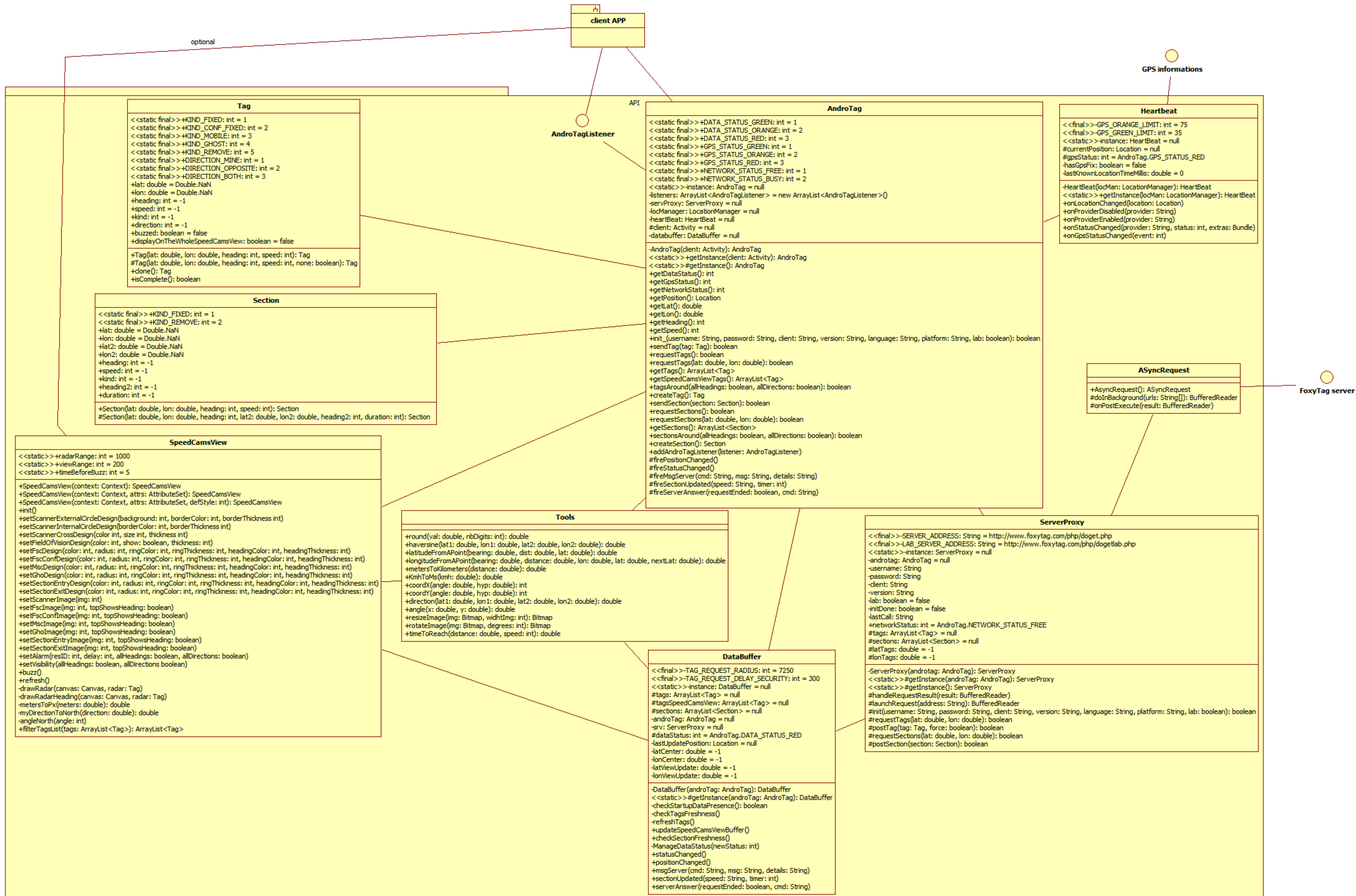
3.9 Tag....11
3.9.1 Objective....11
3.9.2 Responsibilities....11
3.9.3 Graphical representation....12
3.10 Tools....12
3.10.1 Objective....12
3.10.2 Graphical representation....13
4 General remarks....13

1 Introduction

If you plan to modify the API or to extend its functionality, continue its development, this document is for you. In this document I will explain you how I build the API, what were the choices I made, and why. Before reading this document, you should read the AndroTag API User guide as well as the Server API.

2 Class diagram

In the next page you will find an almost complete class diagram of the API. Almost, because I didn't write all the variables of the SpeedCamsView (there are 56 variables which would make the diagram unreadable). This diagram contains method signatures, Constants, variables, and methods and variable visibility. I hope this will help you to have a global visibility and understanding of the system.



client APP
optional
GPS informations
API
AndroTagListener
Tag
<<static final>>+KIND_FIXED: int = 1
<<static final>>+KIND_CONF_FIXED: int = 2
<<static final>>+KIND_MOBILE: int = 3
<<static final>>+KIND_GHOST: int = 4
<<static final>>+KIND_REMOVE: int = 5
<<static final>>+DIRECTION_MINE: int = 1
<<static final>>+DIRECTION_OPPOSITE: int = 2
<<static final>>+DIRECTION_BOTH: int = 3
+lat: double = Double.NaN
+lon: double = Double.NaN
+heading: int = -1
+speed: int = -1
+kind: int = -1
+direction: int = -1
+buzzed: boolean = false
+displayOnTheWholeSpeedCamsView: boolean = false
+Tag(lat: double, lon: double, heading: int, speed: int): Tag
#Tag(lat: double, lon: double, heading: int, speed: int, none: boolean): Tag
+clone(): Tag
+isComplete(): boolean
AndroTag
<<static final>>+DATA_STATUS_GREEN: int = 1
<<static final>>+DATA_STATUS_ORANGE: int = 2
<<static final>>+DATA_STATUS_RED: int = 3
<<static final>>+GPS_STATUS_GREEN: int = 1
<<static final>>+GPS_STATUS_ORANGE: int = 2
<<static final>>+GPS_STATUS_RED: int = 3
<<static final>>+NETWORK_STATUS_FREE: int = 1
<<static final>>+NETWORK_STATUS_BUSY: int = 2
<<static>>-instance: AndroTag = null
-listeners: ArrayList<AndroTagListener> = new ArrayList<AndroTagListener>()
-servProxy: ServerProxy = null
-locManager: LocationManager = null
-heartBeat: HeartBeat = null
#client: Activity = null
-databuffer: DataBuffer = null
-AndroTag(client: Activity): AndroTag
<<static>>+getInstance(client: Activity): AndroTag
<<static>>#getInstance(): AndroTag
+getDataStatus(): int
+getGpsStatus(): int
+getNetworkStatus(): int
+getPosition(): Location
+getLat(): double
+getLon(): double
+getHeading(): int
+getSpeed(): int
+init_(username: String, password: String, client: String, version: String, language: String, platform: String, lab: boolean): boolean
+sendTag(tag: Tag): boolean
+requestTags(): boolean
+requestTags(lat: double, lon: double): boolean
+getTags(): ArrayList<Tag>
+getSpeedCamsViewTags(): ArrayList<Tag>
+tagsAround(allHeadings: boolean, allDirections: boolean): boolean
+createTag(): Tag
+sendSection(section: Section): boolean
+requestSections(): boolean
+requestSections(lat: double, lon: double): boolean
+getSections(): ArrayList<Section>
+sectionsAround(allHeadings: boolean, allDirections: boolean): boolean
+createSection(): Section
+addAndroTagListener(listener: AndroTagListener)
#firePositionChanged()
#fireStatusChanged()
#fireMsgServer(cmd: String, msg: String, details: String)
#fireSectionUpdated(speed: String, timer: int)
#fireServerAnswer(requestEnded: boolean, cmd: String)
Heartbeat
<<final>>-GPS_ORANGE_LIMIT: int = 75
<<final>>-GPS_GREEN_LIMIT: int = 35
<<static>>-instance: HeartBeat = null
#currentPosition: Location = null
#gpsStatus: int = AndroTag.GPS_STATUS_RED
-hasGpsFix: boolean = false
-lastKnownLocationTimeMillis: double = 0
-HeartBeat(locMan: LocationManager): HeartBeat
<<static>>+getInstance(locMan: LocationManager): HeartBeat
+onLocationChanged(location: Location)
+onProviderDisabled(provider: String)
+onProviderEnabled(provider: String)
+onStatusChanged(provider: String, status: int, extras: Bundle)
+onGpsStatusChanged(event: int)
Section
<<static final>>+KIND_FIXED: int = 1
<<static final>>+KIND_REMOVE: int = 2
+lat: double = Double.NaN
+lon: double = Double.NaN
+lat2: double = Double.NaN
+lon2: double = Double.NaN
+heading: int = -1
+speed: int = -1
+kind: int = -1
+heading2: int = -1
+duration: int = -1
+Section(lat: double, lon: double, heading: int, speed: int): Section
#Section(lat: double, lon: double, heading: int, lat2: double, lon2: double, heading2: int, duration: int): Section
ASyncRequest
+AsyncRequest(): ASyncRequest
#doInBackground(urls: String[]): BufferedReader
#onPostExecute(result: BufferedReader)
FoxyTag server
SpeedCamsView
<<static>>+radarRange: int = 1000
<<static>>+viewRange: int = 200
<<static>>+timeBeforeBuzz: int = 5
+SpeedCamsView(context: Context): SpeedCamsView
+SpeedCamsView(context: Context, attrs: AttributeSet): SpeedCamsView
+SpeedCamsView(context: Context, attrs: AttributeSet, defStyle: int): SpeedCamsView
+init()
+setScannerExternalCircleDesign(background: int, borderColor: int, borderThickness int)
+setScannerInternalCircleDesign(borderColor: int, borderThickness int)
+setScannerCrossDesign(color int, size int, thickness int)
+setFieldOfVisionDesign(color: int, show: boolean, thickness: int)
+setFscDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setFscConfDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setMscDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setGhoDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setSectionEntryDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setSectionExitDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setScannerImage(img: int)
+setFscImage(img: int, topShowsHeading: boolean)
+setFscConfImage(img: int, topShowsHeading: boolean)
+setMscImage(img: int, topShowsHeading: boolean)
+setGhoImage(img: int, topShowsHeading: boolean)
+setSectionEntryImage(img: int, topShowsHeading: boolean)
+setSectionExitImage(img: int, topShowsHeading: boolean)
+setAlarm(resID: int, delay: int, allHeadings: boolean, allDirections: boolean)
+setVisibility(allHeadings: boolean, allDirections boolean)
+buzz()
+refresh()
-drawRadar(canvas: Canvas, radar: Tag)
-drawRadarHeading(canvas: Canvas, radar: Tag)
-metersToPx(meters: double): double
-myDirectionToNorth(direction: double): double
-angleNorth(angle: int)
+filterTagsList(tags: ArrayList<Tag>): ArrayList<Tag>
Tools
+round(val: double, nbDigits: int): double
+haversine(lat1: double, lon1: double, lat2: double, lon2: double): double
+latitudeFromAPoint(bearing: double, dist: double, lat: double): double
+longitudeFromAPoint(bearing: double, distance: double, lon: double, lat: double, nextLat: double): double
+metersToKilometers(distance: double): double
+KmhToMs(kmh: double): double
+coordX(angle: double, hyp: double): int
+coordY(angle: double, hyp: double): int
+direction(lat1: double, lon1: double, lat2: double, lon2: double): double
+angle(x: double, y: double): double
+resizeImage(img: Bitmap, widhtImg: int): Bitmap
+rotateImage(img: Bitmap, degrees: int): Bitmap
+timeToReach(distance: double, speed: int): double
ServerProxy
<<final>>-SERVER_ADDRESS: String = http://www.foxytag.com/php/doget.php
<<final>>-LAB_SERVER_ADDRESS: String = http://www.foxytag.com/php/dogetlab.php
<<static>>-instance: ServerProxy = null
-androtag: AndroTag = null
-username: String
-password: String
-client: String
-version: String
-lab: boolean = false
-initDone: boolean = false
-lastCall: String
+networkStatus: int = AndroTag.NETWORK_STATUS_FREE
#tags: ArrayList<Tag> = null
#sections: ArrayList<Section> = null
#latTags: double = -1
#lonTags: double = -1
-ServerProxy(androtag: AndroTag): ServerProxy
<<static>>#getInstance(androTag: AndroTag): ServerProxy
<<static>>#getInstance(): ServerProxy
#handleRequestResult(result: BufferedReader)
#launchRequest(address: String): BufferedReader
#init(username: String, password: String, client: String, version: String, language: String, platform: String, lab: boolean): boolean
#requestTags(lat: double, lon: double): boolean
#postTag(tag: Tag, force: boolean): boolean
#requestSections(lat: double, lon: double): boolean
#postSection(section: Section): boolean
DataBuffer
<<final>>-TAG_REQUEST_RADIUS: int = 7250
<<final>>-TAG_REQUEST_DELAY_SECURITY: int = 300
<<static>>-instance: DataBuffer = null
#tags: ArrayList<Tag> = null
#tagsSpeedCamsView: ArrayList<Tag> = null
#sections: ArrayList<Section> = null
-androTag: AndroTag = null
-srv: ServerProxy = null
#dataStatus: int = AndroTag.DATA_STATUS_RED
-lastUpdatePosition: Location = null
-latCenter: double = -1
-lonCenter: double = -1
-latViewUpdate: double = -1
-lonViewUpdate: double = -1
-DataBuffer(androTag: AndroTag): DataBuffer
<<static>>#getInstance(androTag: AndroTag): DataBuffer
-checkStartupDataPresence(): boolean
-checkTagsFreshness()
-refreshTags()
+updateSpeedCamsViewBuffer()
+checkSectionFreshness()
-ManageDataStatus(newStatus: int)
+statusChanged()
+positionChanged()
+msgServer(cmd: String, msg: String, details: String)
+sectionUpdated(speed: String, timer: int)
+serverAnswer(requestEnded: boolean, cmd: String)

3 Classes description

In this section, I will explain you the responsibilities of each class, what they do, why and how.

3.1 AndroTag

This class is the main entry point for client application. They should use only this class to access API services. This class is the result of the implementation of the Facade architectural pattern, and is also the privileged path of communication through the API. If you develop a new component for the API, and want to access to other component services like for example position, you should always use a function from AndroTag that can provide you what you want. And if you develop new services, make them available from AndroTag

3.1.1 Objective

This class has for objective to simplify the usage of the API for clients. It also has for objective to reduce communication paths by becoming an obligatory path to obtain the services.

3.1.2 Responsibilities

The main responsibility of this class is to build the API. As almost all classes are singleton, and only $AndroTag.getInstance$ is called, it is the responsibility of AndroTag to instantiate and refer each API components. As AndroTag is in the center of the API, It also has the responsibility to manage the Event system. If you want to subscribe to API's events, there is a method in AndroTag that allows that. This method is called $addAndroTagListener$ and is public, because client's apps should want to listen these events. In opposite, the action to launch events should only be available to the API. API classes that want to launch events, can call $fireEVENT_NAME$ methods of AndroTag. These methods are protected, because only the API can launch API's events.

3.1.3 Graphical representation

```
AndroTag
<<static final>>+DATA_STATUS_GREEN: int = 1
<<static final>>+DATA_STATUS_ORANGE: int = 2
<<static final>>+DATA_STATUS_RED: int = 3
<<static final>>+GPS_STATUS_GREEN: int = 1
<<static final>>+GPS_STATUS_ORANGE: int = 2
<<static final>>+GPS_STATUS_RED: int = 3
<<static final>>+NETWORK_STATUS_FREE: int = 1
<<static final>>+NETWORK_STATUS_BUSY: int = 2
<<static>>-instance: AndroTag = null
-listeners: ArrayList<AndroTagListener> = new ArrayList<AndroTagListener>()
-servProxy: ServerProxy = null
-locManager: LocationManager = null
-heartBeat: HeartBeat = null
#client: Activity = null
-databuffer: DataBuffer = null
-AndroTag(client: Activity): AndroTag
<<static>>+getInstance(client: Activity): AndroTag
<<static>>#getInstance(): AndroTag
+getDataStatus(): int
+getGpsStatus(): int
+getNetworkStatus(): int
+getPosition(): Location
+getLat(): double
+getLon(): double
+getHeading(): int
+getSpeed(): int
+init_(username: String, password: String, client: String, version: String, language: String, platform: String, lab: boolean): boolean
+sendTag(tag: Tag): boolean
+requestTags(): boolean
+requestTags(lat: double, lon: double): boolean
+getTags(): ArrayList<Tag>
+getSpeedCamsViewTags(): ArrayList<Tag>
+tagsAround(allHeadings: boolean, allDirections: boolean): boolean
+createTag(): Tag
+sendSection(section: Section): boolean
+requestSections(): boolean
+requestSections(lat: double, lon: double): boolean
+getSections(): ArrayList<Section>
+sectionsAround(allHeadings: boolean, allDirections: boolean): boolean
+createSection(): Section
+addAndroTagListener(listener: AndroTagListener)
#firePositionChanged()
#fireStatusChanged()
#fireMsgServer(cmd: String, msg: String, details: String)
#fireSectionUpdated(speed: String, timer: int)
#fireServerAnswer(requestEnded: boolean, cmd: String)
```

3.2 AndroTagListener

This interface define the requirements classes must satisfy in order to be able to subscribe API's events

3.2.1 Objective

Define the methods that any class that want to subscribe to API's events must satisfy.

3.2.2 Graphical representation

<<interface>>
AndroTagListener

+statusChanged()
+positionChanged()
+msgServer(cmd: String, msg: String, details: String)
+sectionUpdated(speed: String, timer: int)
+serverAnswer(requestEnded: boolean, cmd: String)

3.3 ASyncRequest

This class extends AsyncTask and is a parallel task.

3.3.1 Objective

This class objective is to launch requests to the server on another thread

3.3.2 Responsibilities

The responsibility of this class is to build and launch a request provided as a String and to receive the result. It also has the responsibility to forward the result to ServerProxy class.

3.3.3 Graphical representation

ASyncRequest

+AsyncRequest(): ASyncRequest
#doInBackground(urls: String[]): BufferedReader
#onPostExecute(result: BufferedReader)

3.4 DataBuffer

This class keeps in memory the tags around the user and tries to always have up to date tags

3.4.1 Objective

The main objective of this class is to automatically take in consideration the user's moves in order to always have good tags around the user. This class implements algorithms in order to optimize connections. For more info, refer to the dedicated section in the ServerAPI document.

3.4.2 Responsibilities

This class has the responsibility to keep up to date tags in memory. These tags should be around the user, and tags that are displayed on screen should always be up to date.
This class has the responsibility to provide the tags that should be displayed on screen.
It has the responsibility to request new tags when we are leaving the protected zone (see Server API if you don't understand this word)

3.4.3 Graphical representation

DataBuffer
<<final>>-TAG_REQUEST_RADIUS: int = 7250 <<final>>-TAG_REQUEST_DELAY_SECURITY: int = 300 <<static>>-instance: DataBuffer = null #tags: ArrayList<Tag> = null #tagsSpeedCamsView: ArrayList<Tag> = null #sections: ArrayList<Section> = null -androTag: AndroTag = null -srv: ServerProxy = null #dataStatus: int = AndroTag.DATA_STATUS_RED -lastUpdatePosition: Location = null -latCenter: double = -1 -lonCenter: double = -1 -latViewUpdate: double = -1 -lonViewUpdate: double = -1
-DataBuffer(androTag: AndroTag): DataBuffer <<static>>#getInstance(androTag: AndroTag): DataBuffer -checkStartupDataPresence(): boolean -checkTagsFreshness() -refreshTags() +updateSpeedCamsViewBuffer() +checkSectionFreshness() -ManageDataStatus(newStatus: int) +statusChanged() +positionChanged() +msgServer(cmd: String, msg: String, details: String) +sectionUpdated(speed: String, timer: int) +serverAnswer(requestEnded: boolean, cmd: String)

3.5 HeartBeat

This class manages the GPS and like its name indicates is the heart of the system: it gives the impulsions.

3.5.1 Objective

Abstraction layer between the GPS and the API. It provides GPS information.

3.5.2 Responsibilities

Provide GPS information, launch GPS API events such as $positionChanged()$. It also has the responsibility to verify that the last tag request was done less than 5 minutes ago. If it is not the case, it will request new tags in order to have compliance with Server API recommendations.

3.5.3 Graphical representation

Heartbeat
<<final>>-GPS_ORANGE_LIMIT: int = 75 <<final>>-GPS_GREEN_LIMIT: int = 35 <<static>>-instance: HeartBeat = null #currentPosition: Location = null #gpsStatus: int = AndroTag.GPS_STATUS_RED -hasGpsFix: boolean = false -lastKnownLocationTimeMillis: double = 0
-HeartBeat(locMan: LocationManager): HeartBeat <<static>>+getInstance(locMan: LocationManager): HeartBeat +onLocationChanged(location: Location) +onProviderDisabled(provider: String) +onProviderEnabled(provider: String) +onStatusChanged(provider: String, status: int, extras: Bundle) +onGpsStatusChanged(event: int)

3.6 Section

This class is the representation of an average speed camera. As of today it is useless, because the average speed camera functionality is not completely implemented in the API.

3.6.1 Objective

Numerical representation of a section

3.6.2 Responsibilities

Not implemented yet

3.6.3 Graphical representation

Section
<<static final>>+KIND_FIXED: int = 1 <<static final>>+KIND_REMOVE: int = 2 +lat: double = Double.NaN +lon: double = Double.NaN +lat2: double = Double.NaN +lon2: double = Double.NaN +heading: int = -1 +speed: int = -1 +kind: int = -1 +heading2: int = -1 +duration: int = -1
+Section(lat: double, lon: double, heading: int, speed: int): Section #Section(lat: double, lon: double, heading: int, lat2: double, lon2: double, heading2: int, duration: int): Section

3.7 ServerProxy

This class manages global communication with server and parses the results. This class reflects the implementation of the Proxy pattern. All the communication with the server should pass through this class.

3.7.1 Objective

Restrict communication path, have a control of the communication with the API, and allow only one request at a time. It also has the responsibility to keep in memory the username, password, client version and server used. These parameters are specified by the client when it calls the init method.

3.7.2 Responsibilities

Launch ASyncRequests, handle and parse the results, know the network state (busy or free), launch corresponding events when an answer or message is received from the server.

3.7.3 Graphical representation

ServerProxy
<<final>>-SERVER_ADDRESS: String = http://www.foxytag.com/php/doget.php <<final>>-LAB_SERVER_ADDRESS: String = http://www.foxytag.com/php/dogetlab.php <<static>>-instance: ServerProxy = null -androtag: AndroTag = null -username: String -password: String -client: String -version: String -lab: boolean = false -initDone: boolean = false -lastCall: String +networkStatus: int = AndroTag.NETWORK_STATUS_FREE #tags: ArrayList<Tag> = null #sections: ArrayList<Section> = null #latTags: double = -1 #lonTags: double = -1
-ServerProxy(androtag: AndroTag): ServerProxy <<static>>#getInstance(androTag: AndroTag): ServerProxy <<static>>#getInstance(): ServerProxy #handleRequestResult(result: BufferedReader) #launchRequest(address: String): BufferedReader #init(username: String, password: String, client: String, version: String, language: String, platform: String, lab: boolean): boolean #requestTags(lat: double, lon: double): boolean #postTag(tag: Tag, force: boolean): boolean #requestSections(lat: double, lon: double): boolean #postSection(section: Section): boolean

3.8 SpeedCamsView

This class displays the tags on screen. It is highly customizable.

3.8.1 Objective

Provide to the client a component that, when used with the API, can automatically display tags around you during your trip.

3.8.2 Responsibilities

Make a sound (buzz) when approaching a tag. Restrict the tags displayed depending on some parameters.

3.8.3 Graphical representation

```
SpeedCamsView
---
<<static>>+radarRange: int = 1000
<<static>>+viewRange: int = 200
<<static>>+timeBeforeBuzz: int = 5
---
+SpeedCamsView(context: Context): SpeedCamsView
+SpeedCamsView(context: Context, attrs: AttributeSet): SpeedCamsView
+SpeedCamsView(context: Context, attrs: AttributeSet, defStyle: int): SpeedCamsView
+init()
+setScannerExternalCircleDesign(background: int, borderColor: int, borderThickness int)
+setScannerInternalCircleDesign(borderColor: int, borderThickness int)
+setScannerCrossDesign(color int, size int, thickness int)
+setFieldOfVisionDesign(color: int, show: boolean, thickness: int)
+setFscDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setFscConfDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setMscDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setGhoDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setSectionEntryDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setSectionExitDesign(color: int, radius: int, ringColor: int, ringThickness: int, headingColor: int, headingThickness: int)
+setScannerImage(img: int)
+setFscImage(img: int, topShowsHeading: boolean)
+setFscConfImage(img: int, topShowsHeading: boolean)
+setMscImage(img: int, topShowsHeading: boolean)
+setGhoImage(img: int, topShowsHeading: boolean)
+setSectionEntryImage(img: int, topShowsHeading: boolean)
+setSectionExitImage(img: int, topShowsHeading: boolean)
+setAlarm(resID: int, delay: int, allHeadings: boolean, allDirections: boolean)
+setVisibility(allHeadings: boolean, allDirections boolean)
+buzz()
+refresh()
-drawRadar(canvas: Canvas, radar: Tag)
-drawRadarHeading(canvas: Canvas, radar: Tag)
-metersToPx(meters: double): double
-myDirectionToNorth(direction: double): double
-angleNorth(angle: int)
+filterTagsList(tags: ArrayList<Tag>): ArrayList<Tag>
```

As explained at the top of this document, I didn't write any of the variables (there are 56 variables that would make the diagram unreadable).

3.9 Tag

This class is the representation of a speed camera.

3.9.1 Objective

Numerical representation of a speed camera

3.9.2 Responsibilities

Know the kind of speed camera it is, know its latitude, longitude, heading. It also has to know if it already has buzzed, and if it is on the screen.

3.9.3 Graphical representation

Tag

<<static final>>+KIND_FIXED: int = 1
<<static final>>+KIND_CONF_FIXED: int = 2
<<static final>>+KIND_MOBILE: int = 3
<<static final>>+KIND_GHOST: int = 4
<<static final>>+KIND_REMOVE: int = 5
<<static final>>+DIRECTION_MINE: int = 1
<<static final>>+DIRECTION_OPPOSITE: int = 2
<<static final>>+DIRECTION_BOTH: int = 3
+lat: double = Double.NaN
+lon: double = Double.NaN
+heading: int = -1
+speed: int = -1
+kind: int = -1
+direction: int = -1
+buzzed: boolean = false
+displayOnTheWholeSpeedCamsView: boolean = false

+Tag(lat: double, lon: double, heading: int, speed: int): Tag
#Tag(lat: double, lon: double, heading: int, speed: int, none: boolean): Tag
+clone(): Tag
+isComplete(): boolean

3.10 Tools

This class is a bunch of tools.

3.10.1 Objective

Provide useful tools for use inside and outside the API

3.10.2 Graphical representation

```
Tools
---
+round(val: double, nbDigits: int): double
+haversine(lat1: double, lon1: double, lat2: double, lon2: double): double
+latitudeFromAPoint(bearing: double, dist: double, lat: double): double
+longitudeFromAPoint(bearing: double, distance: double, lon: double, lat: double, nextLat: double): double
+metersToKilometers(distance: double): double
+KmhToMs(kmh: double): double
+coordX(angle: double, hyp: double): int
+coordY(angle: double, hyp: double): int
+direction(lat1: double, lon1: double, lat2: double, lon2: double): double
+angle(x: double, y: double): double
+resizeImage(img: Bitmap, widhtImg: int): Bitmap
+rotateImage(img: Bitmap, degrees: int): Bitmap
+timeToReach(distance: double, speed: int): double
```

4 General remarks

If you want to continue the development, try to respect the two architectural patterns I implemented, and try to always think about responsibility. Responsibilities should not be shared between classes/component. This would make the code unreadable, and the reverse engineering would become very difficult.

Good luck !