UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Metode eficiente de localizare pentru dispozitivele mobile

Coordonatori ştiinţific:
Prof. Dr. Ing. Nicolae Ţăpuş
As. Drd. Ing. Alexandru Olteanu

Absolvent:
Cosmin Ştefan-Dobrin

BUCUREŞTI

2012

POLITEHNICA UNIVERSITY OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Efficient localization methods for mobile devices

Thesis supervisors:
Prof. Eng. Nicolae Ţăpuş, PhD.
As. Eng. Alexandru Olteanu

Author:
Cosmin Ştefan-Dobrin

BUCHAREST

2012

ABSTRACT

Location-aware applications have become some of the most important and used applications on mobile devices and people are in constant search of tools that make their life easier and more efficient. Yet, the main technology used for locating a mobile device is still the Global Positioning System, which has high power consumption and requires special hardware. However, there are situations, such as the case of applications for Location-Based Alarms, when an exact localization is not required. In this paper, we study four methods for localizing a device and what algorithms could be used to improve the process. We present the development of FenceIt, an Android application that allows users to define location-based alarms and which uses alternative methods to GPS positioning and different data processing algorithms in order to preserve both the battery level and the data connection, while being easy to use and attractively designed.

TABLE OF CONTENTS

NOTATIONS AND ABBREVIATIONS

CDMA – Code division multiple access
BSSID – Basic Service Set Identifier
GPS – Global Positioning System
GSM – Global System for Mobile Communication
OS – Operating System
LBR – Location Based Reminders
LBS – Location Based Service
SDK – Software Development Kit
UI – User Interface

TABLE OF FIGURES

Chapter 1: Introduction

1.1 Current situation and motivation

The society is moving faster every day and the human body and mind have to keep up with this always accelerating world, in which access to information is crucial and in which, for most of the situations, there is only one right moment to do something. This has made people start to use more and more tools to help them in their daily life.

One of the most important needs for people in today's society is the need to remember that they need to do something, sometimes, somewhere, with somebody. And for this, people have been using various methods to help them remember things: writing to-do lists, sticking post-its on the fridge or next to the door, using an alarm, setting up an email reminder, asking a friend to give a phone call and many others.

The problem with these solutions is that, most of the time, they might not be available or may not be appropriate. For instance, sticking a post-it at home in the evening to remind you to do something when you get at work in the morning has a small success rate, as you don't have the required information when and where it is needed -- at work, in the morning. Or, to give another example, setting a timed-based alarm to remind one to do something right as he/she gets home or just before leaving will most likely not work, as few people can always know exactly when such events happen.

We can easily conclude that some reminders or some actions would be more effective if they would be triggered based on various types of contextual information, rather than only time. The types of systems that use different types of information are usually called "context-aware systems". For instance, an action to automatically set one's mobile device on 'Silent' when he/she gets to school is much more helpful if it's not triggered 20 minutes before he/she reaches school or half an hour after the class has started (as it might happen if a time-based alarm would be set). Meanwhile, a solution that would take into account the person's location would fix such a problem perfectly.

In the category of the context aware systems we can include Location Based Services (LBS), which have the ability to make use of information regarding location to provide better services to users. A great example of such a LBS is a system that would allow users to set location-based alarms that would notify them when triggered. And, as in today's society almost everyone has a mobile device in possession almost at any moment, the mobile platforms are excellent deployment targets for these kinds of systems. The development and evolution of mobile devices has been exponential in the recent years and their performance is now comparable to full-scale computers, while their size and usefulness allows them to be carried by people all the time. Thus, they give the owners the capability of having permanently available solutions for defining location-based alarms.

The domain of such location-based alarms applications has recently started to get the attention of a growing number of developers, but the existing solutions are still in an incipient stage. Some of the problems that most of the developers have encountered so far include: high battery consumption due to the use of location-sensing hardware (GPS), necessity to use a data connection which increases costs for some users or is unavailable for others. Moreover, some situations have certain defining factors that should be taken into consideration, allowing applications of this type to be improved. For instance, special algorithms that increase the time interval between updates and/or location-data acquisitions in certain cases can be used. All these aspects have motivated us to further study this field and look for alternative solutions.

1.2 Project Goals and Achievements

This project is focused on studying the available methods for localizing a mobile device in an area, in an efficient way. Our goal was to design a system that is not dependent on location-sensing hardware and does not need the device to have permanent Internet connectivity, all while making use of any conditions and contextual information that might improve the algorithms' performance.

The work has led to the development of FenceIt, a mobile application for the Android operating system, thus having a huge market and being accessible to numerous people, that allows users to create, manage and use alarms based on the location -- it is a part of the Location Based Services (LBS) category of applications. A user can define locations, using various methods and utilizing different information gathered from the environment, and then configure alarms which are triggered when the device enters or leaves these locations. When an alarm is set off, the user can be notified or other pre-defined actions can be automatically executed. Multiple methods of defining locations, gathering data from the context (environment) and various algorithms have been implemented and analyzed, the results being presented in this paper.

1.3 Thesis Outline

This paper is divided in 9 chapters, as follows:

Chapter 1: Introduction
This chapter offers a brief presentation of the existing situation, the motivation for the paper, the project and the goals of the study.

Chapter 2: Related Word
This chapter presents the existing studies in the field and previously developed mobile applications.

Chapter 3: Localization Methods
This chapter presents the theoretical aspects behind localization methods for mobile devices and what optimizations can be implemented.

Chapter 4: Application Design
This chapter describes the application implemented for the study, with its core concepts, requirements and most important use-cases.

Chapter 5: System Architecture
This chapter offers a description of the application's architecture and the interaction between components.

Chapter 6: Implementation
This chapter presents the implementation of the mobile application, with details regarding its modules.

Chapter 7: Testing and Evaluation
This chapter describes the testing methods used and the results of the application's evaluation.

Chapter 8: Future Work
This chapter shortly describes improvements that can be made to the application in the future.

Chapter 9: Conclusion
This chapter presents the conclusions of the paper.

Chapter 2: Related Work

Implementation of location-based reminder applications for mobile devices is not a brand new idea, as many developers have started to develop such applications in recent years. However, as mentioned before, different problems have been encountered.

One of the first experiments in this field is the work of Dey and Abowd (2000) [1], who developed Cybre-Minder, a context-aware application that allows users to send and receive reminders, based on complex contextual information, such as time, location and other available data. The application supported various types of reminder delivery, such as simple audio & visual reminder, SMS on a mobile phone, email or printing on a local printer. For detection of the available context information, it used Context Toolkit.

Other initial papers explored this domain, for example comMotion, by Marmasse and Schmandt (2003) [2]. These incipient applications were mainly using GPS positioning. However, some studies by Barbeau et al. (2008) [3] and Gasinski (2009) [4], showed that the use of the GPS hardware interface and the continuous access to a data connection cause the battery level to decrease quickly. Thus, solutions based only on GPS hardware were proven to be less practical for large-scale use in daily life of mobile users.

Various papers started to explore alternative methods for locating a device without requiring GPS hardware. LaMarca et al. (2005) [5] developed PlaceLab, which was one of the first solutions to position a device using radio beacons. The methods used included 802.11 wireless access points and GSM beacons. The study manages to achieve a median accuracy of 20-30 meters with almost perfect coverage measured by availability in people's daily lives and is considered to be one of the first important ones in this field.

Other studies, such as Sohn et al (2005) [6], which developed Place-Its, Ludford et al. (2005) [7], which worked on PlaceMail, Meeuwissen et al. (2007) [8] and Hedin and Noren (2009) [9] explored various methods for locating a device using alternative methods, including Cell Tower Ids. Their results have proven useful for many other developers.

Regarding existing implementations of Location-Based Reminder (LBR) applications for the modern mobile operating systems, in the past couple of years there have been some developers that explored this domain.

In the autumn of 2011, in the new iOS5 [10], Apple provided a LBR application for the users. However, the biggest issue is that it works only on devices that have GPS capabilities. The fact that it is only based on GPS positioning is a huge drawback, as this system consumes a great amount of electricity, draining the battery relatively fast. Furthermore, its features were extremely simple and did not provide much customizability for users.

One of the most popular applications of this kind, for Android, is Spoty Location Reminder [11]. Initially, it started as an application that allowed the users to specify a location by positioning a marker on a map and used only GPS and Coarse positioning. However, the latest version provides the users with the possibility to define locations also using connected Wi-Fi networks or Wi-Fi/Cell Towers in Range, although the feature is still in development and isn't very reliable at this point.

Another popular LBR Android application is ePythia To Do List [12]. Although it doesn't provide advanced methods for location acquisition, mainly using GPS and Coarse positioning, it brings some new concepts to LBR applications. For instance, it allows the users to use the Google Points–Of–Interests database (Google Places) to easy find close-by public locations -- restaurants, markets, banks, etc-- and to set a task with a location reminder at these points. This is a very useful integration and is a move in the right direction, in today's IT world, which is moving more and more towards social interaction.

Many other implementations of Android applications for Location-Based Alarms exist; however, most of them rely exclusively on the GPS hardware, which, as proven many times, has very high power consumption, draining battery very fast. Furthermore, a lot of the existing implementations are not optimized at all and, for example, query the operating system for location information constantly, even though this is not necessary in most cases.

As it can be noticed, some research has been done in the field of Location-Based Reminder systems in the recent years. However, most of the existing studies and implementations lack an integration of multiple methods for locating the device and thus lack the capacity of allowing the researchers to make an active comparison of the benefits and disadvantages of the available methods.

This has motivated us to study this field. The results can be seen in the following chapters.

Chapter 3: Localization Methods

In many scenarios it is useful or it suffices to just determine if a device is or is not in a particular area and not exactly where that area is. For example, for the purpose of a location based alarm application, it would suffice to conclude that the device has reached “Home”, not being necessary to know exactly what address “Home” has or what are its geographical coordinates.

However, before we get into details, it is useful to clarify a term that will be used, starting from this point, in the paper. As defined by the Longman’s Dictionary of Contemporary English [13], a “Location” is a “particular place, especially in relation to other areas, buildings” or “the position of something”. In the context of this paper, a Location is the position a device has at a given point, as defined or determined by a subset of environmental factors that place the device there. What this means is that we can say the device is at a particular Location not only if we can accurately place it on the surface of the Earth, but if we can say it is somewhere, based on the surrounding context/ environment.

In the context of the definition presented above, defining a Location for a mobile device can make use of some technical features that such a device might have. For example, it can use its hardware sensors/interfaces to gather information from the context/environment and, later, when it is again in the same place, it can match the conditions identified before and can conclude that it is now “in the same location” as before.

When discussing the problem of efficiently localizing a mobile device in a particular area, multiple methods can be used, each based on some particularities or characteristics of that location. This chapter presents four methods of localizing a mobile device in an area and some optimization algorithms. It will be discussed from the point of view of an application that just needs to know if the device is or is not in a particular area, such as a location-based alarms application.

3.1 Locations

As mentioned before, we will describe and analyze four types of Locations. Each of them is defined by enough information about a particular place so that a device has the possibility to identify when it enters, leaves or is at that particular Location. This variety of possibilities has the advantage of offering a possible “best-choice” for most of the situations the user would encounter.

The Locations discussed in this chapter can be defined using:

- the Geographical Coordinates of the particular place (Geographical Coordinates Location)
- the Wi-Fi Network the device is Connected to (Connected WiFi Location)
- the Wi-Fi Networks in range (WiFis Detected Location)
- the Cellular Network the device is Connected to (Cell Network Location)

A diagram showing the defining components of each Location type can be seen in Figure 1. In the following sub-sections, the details of each of the four types are provided.

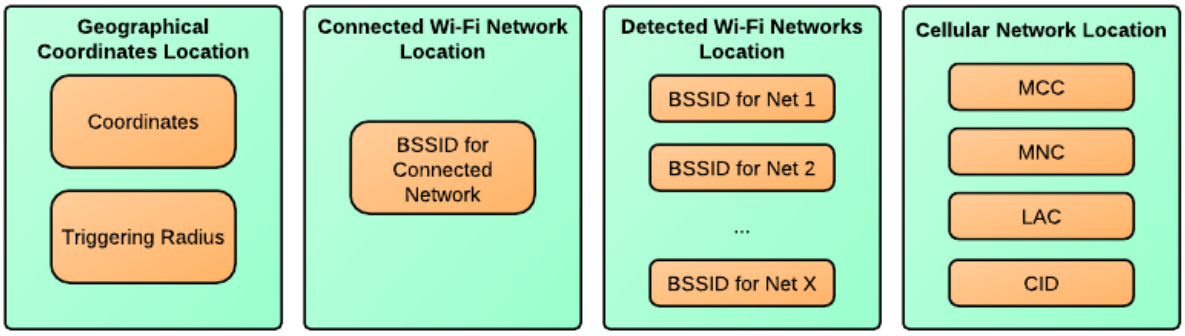

Figure 1 - Diagram of defining components for Locations

3.1.1 Geographical Coordinates Location

The first type of Location is based on the *geographical coordinates* of a given point on the surface of the Earth (GPS coordinates – Latitude and Longitude). What the application does in this case is that it queries the Location Provider of the device for the current coordinates (acquired either using the GPS hardware or using the Coarse method -- localization using the known coordinates of Wi-Fi Networks and Cell Towers – [14] pp.15) or allows the user to select a point on a map. To completely define the Location, a triggering distance from the central point also needs to be set-up. Thus, the key elements necessary to define such a type of Location are: geographical coordinates, triggering distance.

For instance, to set a Location at ones house, the application would acquire the coordinates of the house and set a radius of, to exemplify, 400m around it. Anytime the device gets closer than 400m to this point it is said that it has “entered the Location”, when it is within 400m it is “at the Location” and when it gets farther than 400m from that point it has “left the Location”.

The problem with this type of Location is that, for the Alarm to be triggered, the device has to have GPS hardware/capabilities and has to use it as often as needed, which has the drawback of big battery consumption. The alternative, of using the Coarse method, requires permanent Internet connectivity and is not always available and, when it is, it could not be precise.

Analyzing the goal of a Location-Based Alarms application, it has to be noticed that the exact coordinates of a given location are not required for the alarms to be triggered. Instead, just a method that can say whether the device is or is not in a given area is needed. The next three Location types are based exactly on this concept. The application does not know the global geographical position of that particular Location. It just uses available contextual information that can place the device inside the target area.

3.1.2 Connected Wi-Fi Network Location

This second type of Location is based on the *connected Wi-Fi network*. This method relies on the fact that every Wi-Fi Access Point has a unique or semi-unique identification number, called Basic Service Set Identifier (BSSID).

The BSSID is a 48-bit number which has the same format as an IEEE 802 MAC address. This field uniquely identifies each Basic Server Set (BSS). The value of this field in an Independent BSS (IBSS) is a locally administered IEEE MAC address formed from a 46-bit random number. More details can be found in the IEEE 802.11 Specifications [15] and in [16].

When a user wants to define a Location using the connected Wi-Fi network, the BSSID of this network is stored. When the alarm is checked if it should be triggered, the BSSID of the currently connected network is compared with the stored BSSID. Thus, the key element for this Location type is this unique identification number for a Wireless network, which accurately identifies a place.

The main advantage of this method is that it does only require the wireless interface to be activated. Besides this, it has a very high accuracy and is easy to be checked. Some of the main disadvantages are that sometimes the BSSID of the Wi-Fi network the device can get changed -- for instance, when the access point hardware malfunctions and it is manually changed by the administrator -- and that the device has to be connected to that particular network, not just have it in range. This latter disadvantage is the most important and can cause some issues on devices that do not auto-connect to (previously set-up) Wireless Networks or do not store required authentication credentials during the previous connection process. Another issue can occur when the Location is defined using one Wireless Network, but in that particular area another Wi-Fi network is visible and the device automatically connects to the second one. These types of issues can be however resolved by a proper set-up of the device and of the behavior when in range of Wi-Fi networks it has previously connected to.

3.1.3 Wi-Fi Networks Detected Location

The third possible Location type is the one that is based on the Wi-Fi networks in range of the device or, also called, Wi-Fi networks detected by the Wireless-capable device.

Similarly to the second method (Connected Wi-Fi Location), it is also based on the unique identification number of Wireless Networks (BSSID). More details regarding it can be read in the previous section. This method relies on the acquisition of the BSSIDs of all the networks in range of the device and on storing them so that, at a later moment, they can be compared with the networks in range at that particular time. Besides this, a ratio of the number of Wi-Fi networks that have to be detected can be set up. This introduces a degree of flexibility in the analysis process and allows some errors (e.g. one of the networks is temporarily down) to be overcome.

One advantage compared to the second Location type is that the device does not have to connect to a particular Wi-Fi network, which solves some of the issues presented before. However, the problem is that, depending on what ratio of the number of original networks' BSSIDs have to match, it might not always trigger if the signal from some networks is dampened -- therefore they are not detectable. Another considerable downside to using this method is that the battery consumption of a Wi-Fi Scan is usually larger compared to just getting information about the connected Wi-Fi network and takes a significantly larger amount of time.

3.1.4 Cellular Network Location

The last type of Location is the one that relies on the data provided by the network, when the device has integrated telephony capabilities and is connected to a Telephony Network (Cellular Network) [17]. At any point, a device which is connected to a cellular network is registered to a Cell Site, which can be uniquely identified using a sequence of numbers. Besides this, on some mobile phones, information regarding the surrounding Cell Sites can also be acquired.

In the case of GSM (Global System for Mobile Communication) networks, each of the Cell Sites can be uniquely identified using 4 values [18]:

- MCC - Mobile country code (fixed code and available online)
- MNC - Mobile network code (fixed code and available online)

- LAC - Local Area Code (dynamic and obtained from the device)
- CID - Cell ID (dynamic and obtained from the device)

The combination of MNC and MCC is also known as Network Operator.
In the case of CDMA (Code-Division Multiple Access) cellular networks, each of the cell sites can by uniquely identified using:

- BID - base station identification number
- SID - system identification number
- NID - network identification number

Between these numbers a correspondence can be made: CID < - > BID, LAC <-> NID, MNC & MCC <-> SID. From this point forward the paper will refer only to the GSM cellular network identification numbers for Cell Sites and, using the correspondence mentioned above, the same characteristics apply to CDMA cellular networks.

This location type is based on the fact that using the uniquely identifiable cell sites, plus information regarding the strength of the signal, a device can be located in an area around one Cell Site. The accuracy varies depending on conditions, but can range from a few tens or hundreds of meters in densely populated areas, to even kilometers in unpopulated areas, where the density of Cell Sites per surface unit is smaller. A better accuracy can be obtained if identification data can be obtained about all the detectable Cell Sites in range. In this particular case, using interpolation algorithms, a better fix on the location can be made.

When discussing disadvantages, the accuracy is not the best for this method and in certain situations it cannot be used (e.g. when trying to locate the device in two areas very close to each-other). However, the biggest advantage is that it does not require any additional interface (such as GPS or Wireless) to be activated, besides the normal telephony interface. This has a great importance on the battery consumption of the device in long-case use scenarios. Besides this, it allows Location Based Services to be implemented on devices without a Wi-Fi interface and/or without GPS hardware.

3.1.5 Other aspects regarding Locations

One more important disadvantage has to be stated regarding all the last three Location Types (Connected Network Wi-Fi Location, Wi-Fi Networks Detected Location, Cell Network Location) compared to the first one (Geographical Coordinates Location). While setting a Location by its geographical position can be done from anywhere, for example by setting a pinpoint on a map – though it might need internet connectivity –, for the last three methods the device has to be in that particular location/area, so it can collect the required contextual data. This is sometimes not possible, thus making the first method the only available one. A good example for this case is a person who is travelling by train and wants to be awakened by an alarm when he/she reaches his train destination, in a city where he/she has never been before.

Moreover, as it was stated during the description of each of the Location types, some of them cannot be applied in certain locations or require some specific hardware capabilities that could not always be present (e.g. GPS sensor or Wi-Fi interface). Thus, the decision of which Location type to use in which situation is a very important one and has to be studied in detail.

3.2 Localization Optimizations

While most of the existing applications in this field use a straightforward approach regarding when to check for updates, acquiring information from the environment at regular time intervals and

checking if any conditions have been met, this is not the optimal behavior. In some cases, more information from the context can be used to make the querying algorithm more efficient.

An important type of optimization is only applicable to the first type of alarm (using the geographic coordinates) and considers the distance of the target location to the current position. For instance, if there is only one alarm of this particular active in the application and its target location is far from the current position of the device (for example a few kilometers), querying for GPS location every second is inefficient, as it is not practically possible to arrive at that location in that short amount of time. In such a case, an approximation algorithm that takes into consideration the current speed, direction and distance to target and estimates the arrival moment at the location can be used to dynamically change the frequency of data acquisition from the GPS hardware.

Another optimization considers the inactivity of the device or the lack of movement. If it hasn't moved for a considerable amount of time, the frequency at which any type of data acquisition should be done can decrease, improving the battery consumption. This can prove highly efficient, as it considers cases when, for instance, the user is at work or at home and the mobile device is stationary on a table. If this happens for a considerable amount of time, it's inefficient to acquire location information every second and an algorithm can be used to compute a dynamic waiting time between data gatherings.

The third possible type of optimization includes pre-defined inactivity periods, when the user knows that he/she will most likely not move very often, if at all. As an example, we can consider the case of time spent at home, during the night, when the user is usually stationary and the frequency of alarms' analysis should not be as high as during the day, when the probability of movement is higher.

As detailed, there are various methods for optimizing the localization process and all of them can be combined in a complex algorithm that computes a variable interval between context information gatherings/ checks for triggered alarms. Such an algorithm would take as input the data acquired from the environment (now and/or at previous moments), the settings the user might have made and the constants/ variables/ results that the application developer has researched as the optimum for the particular situation. The output would be an optimized duration until the next check for triggered alarms should be made. Figure 2 shows a diagram describing this optimization algorithm:

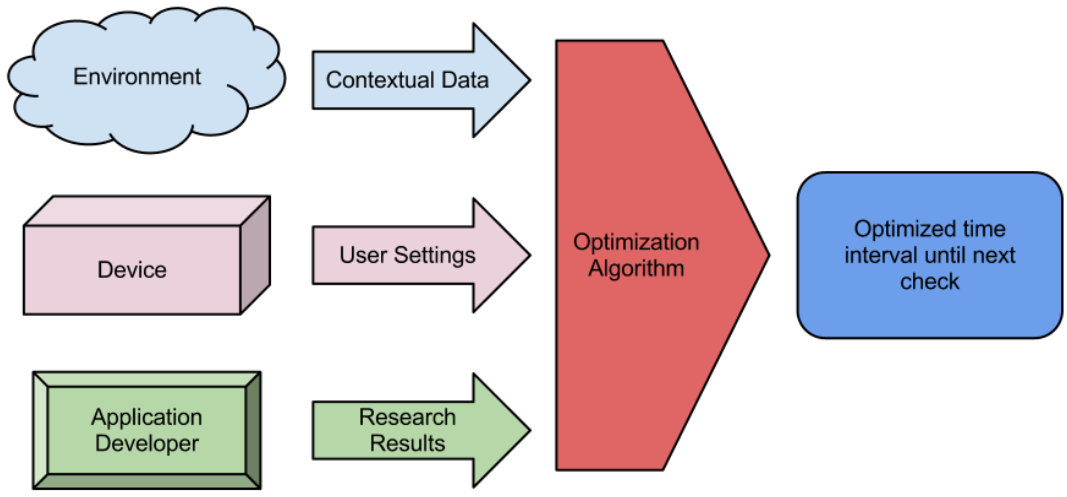

Figure 2 - Diagram of an abstract optimization algorithm during the localization process

The immediate effect of using such an algorithm is that the battery consumption decreases significantly and, in certain cases, the data connection use is reduced.

Chapter 4: Application Design and Requirements

To evaluate and study the different methods of acquiring the location data and the various algorithms that drive the process, we have started the development of FenceIt, a mobile application for the Android operating system. In this chapter we will define the core concepts of the application, what are the applications requirements and the most important use case scenarios.

4.1 Core Concepts

FenceIt is an application that allows the users to create, manage and use alarms that get triggered when the device enters or leaves some custom-defined Locations. When these alarms set off, some actions defined by the user are executed.

There are some concepts regarding the application that have to be defined and that will be used in the paper:

- Alarm – a collection of conditions (Triggers) referring to Locations (as defined in Chapter 3) and a set of actions that get executed when any of the conditions is met.
- Trigger – a condition which defines what the Location of the device should be in order for the any corresponding alarm to be set off.
- Location – as briefly defined in Chapter 3, a Location is a collection of environmental factors/ data that define when the device is in a particular area/ place. In the context of the application, four Location types will be used, as it was presented in Section 3.1.
- Action – an operation that is executed automatically when the containing Alarm is triggered.

When we talk about the relations between these concepts, the most up-level entity is an Alarm. It contains a set of Triggers, which are constantly checked for being triggered, and a set of Actions, which are all executed when the Alarm is set off. Each of the Triggers is a condition regarding a defined Location, of any of the types defined in Chapter 3. A diagram depicting the relations between these concepts can be seen in Figure 3.

FenceIt was developed so that it allows users to define any of the four types of Locations (Connected Network Wi-Fi Location, Wi-Fi Networks Detected Location, Cellular Network Location, Geographical Coordinates Location) and even use combinations of them when creating Triggers for Alarms. For instance, an user can define multiple Triggers for an Alarm such that it gets set off either when the device connects to a particular Wi-Fi Network or when it gets in range of a Cell Site. Thus, depending on the conditions and the available hardware, the application can detect if the device is in an area in various cases and scenarios.

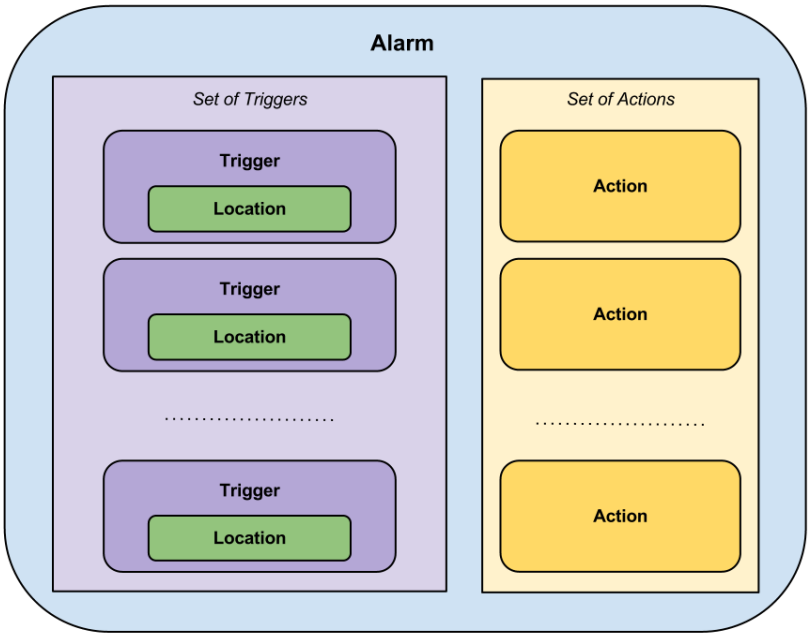

Figure 3 - Diagram of relations between core concepts in FenceIt

4.2 Requirements

FenceIt was designed to help us study the various methods for localizing a mobile device and analyze what algorithms could be used to optimize the overall process. Nonetheless, we also set the goal of developing the application as a fully functional and usable one, so that it could also be used by anyone and so that it could be added in the *Google Market* (recently rebranded into *Play Store*).

Therefore, before developing such an application, some requirements regarding the features it should have and the overall characteristics have to be set.

The architecture and the design of the system have to consider a number of factors, which are important for an application of this size:

- Operating system particularities – each operating system has a set of characteristics that impose some limitations on the process of developing applications. In the particular case of a mobile operating system (such as Android) the architecture of the application has to take the characteristics of the mobile devices into consideration and make use of the development tools created for such an OS.
- Modularity & Extensibility – any subsequent upgrade to the application has to be done with minor interference with the existing and working modules. This allows modules later developed (e.g. new Actions or Location types) to be added with minimal impact on the existing structure.
- Easy to maintain – in high connection with the previous point, an application of this size has to be easy maintainable and whenever some changes are required, they have to be done as easy as possible.
- Usability – the application has to be designed in such a way that it allows inexperienced users (with minimal technical knowledge) to use it. The application should perform as many steps and background actions as possible, without requiring user interaction.
- Intuitive – the interaction of components and the user interface have to be intuitive, allowing a new user to get acquainted with the features as easy as possible. Therefore, one of the

goals was to design the application so that the learning curve would be as smooth as possible.

- Efficiency – as it is an application that will run almost permanently, it is very important for the users to be as efficient as possible with both the battery and with the data connection; otherwise clients might stop using it.
- Attractive Design – as stated, one of our goals was to make the application accessible to the all users, thus the application interface must have an attractive and easy to use design for the UI.

4.3 Use-Case Scenarios

As we have defined what are the main concepts of the application and what are the most important features and characteristics, in this section we will get into the details of how users should interact with the application and what would be the expected outcomes. Thus, we will present some common use-case scenarios, which represent a set or flow of actions that are required to perform a complete operation. These scenarios are particularly useful to understand how an application works and to help developers in testing if the application behaves as expected. The scenarios presented are just the most important ones and are not meant to represent an exhaustive collection of use-cases.

4.3.1 Defining and managing Locations

The first use case scenario presents how a user would create, edit or manage a Location, without being connected (so far) to an Alarm.

An example consists of a user who is outside his house, for example at the university, and wants to define a Location there, gathering information from the environment, so that later he can define an Alarm using this Location even though he is not there anymore. This concept is important because, as presented in Section 3.1.5, some Locations cannot be defined remotely and the device has to be physically there to acquire necessary data.

The Locations can be defined using: Geographical Coordinates, the Wi-Fi Network the device is connected to, the Cellular Network the device is registered to, the Wi-Fi Networks in range, or a combination of these. More details regarding how these information types trigger the alarms are given in Chapter 6: Implementation.

The action flow for how a user can create a new Location can be followed in Figure 4. The approach for editing an existing one or just deleting one is extremely similar.

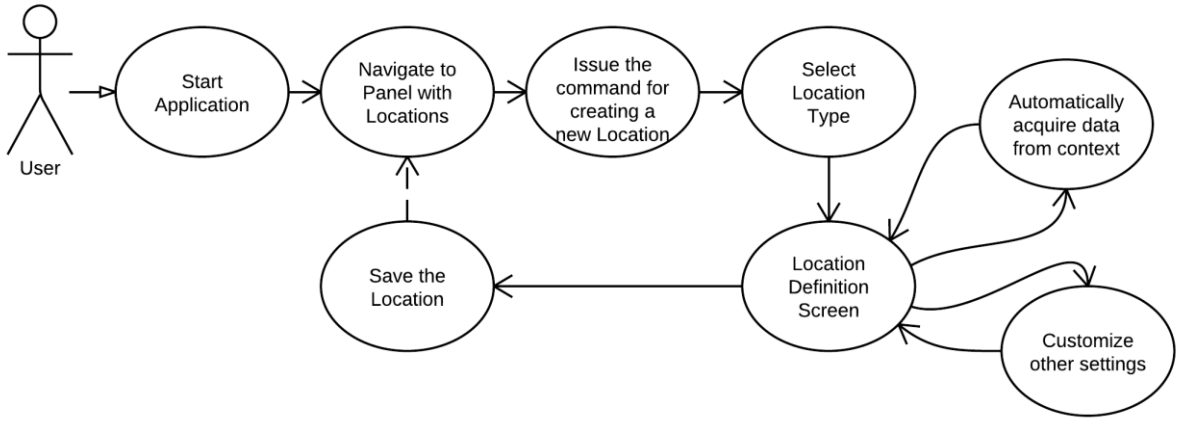

Figure 4 - Use case scenario for creating a Location

As it can be seen, the user has to start the application and navigate to a panel where he can see all existing Locations. From there, he/she can create a new Location (or, just as easy, edit an existing one) by clicking the appropriate command. The user has to first select which of the Location types he wants to create. After this step, the application automatically uses the sensors and interfaces for gathering information from the environment, if requested by the user, or simply allows him/her to customize other settings regarding that location. When everything is set up, the user has to save the Location and he is taken back to the Locations Panel.

4.3.2 Defining and managing Alarms

This use case scenario is more complex and is the one that will be used in most cases by the basic application clients. It defines how a user can create or edit an alarm, by adding new Triggers and new Actions and by either using a pre-defined Location (e.g. created using the previous scenario) or by defining a new one.

A simple example consists of a user who is at home and wants to define an Alarm so that, when he/she goes by the local super-market, he/she is reminded to buy some cooking oil. He/she would define a Trigger that uses a Geographical Coordinates Location (as it's the only one that can be defined remotely) and which would be triggered when the mobile device is close to the super-market. As an action, he/she could choose to receive a Notification, with sound and vibration.

As the process of defining a Location (selecting type, automatically acquiring information from the environment and/or manually configuration and saving) was described in the previous scenario, it will not be repeated in this sub-section. Instead, we will just use the "Define a Location" action to represent the creation process.

The action flow for how a user can create a new Alarm can be followed in Figure 5. The approach for editing an existing one is extremely similar.

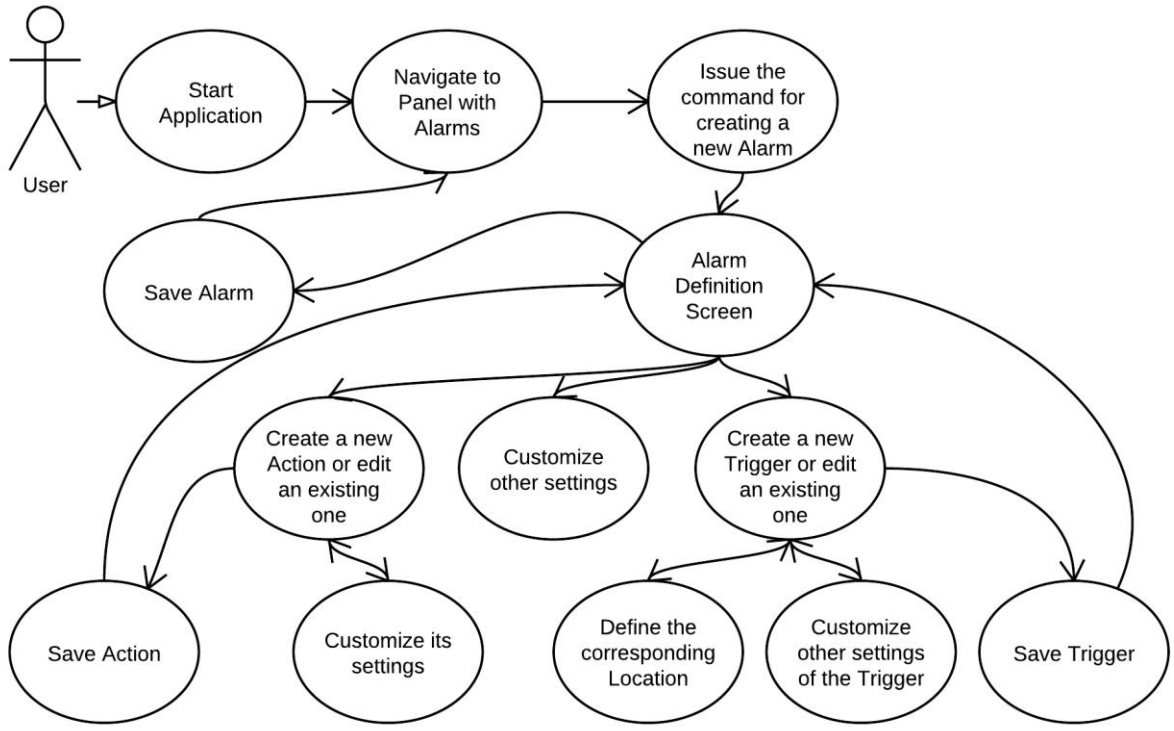

Figure 5 - Use case scenario for creating an Alarm

To quickly describe the process, the user starts defining the alarm by configuring Triggers using the already set up Locations and defining various types of Actions to be executed. Also, there is the possibility for the user to configure the Locations while creating the Triggers, instead of using an already set up Location. When all the details are defined, the user has to save the Alarm.

4.3.3 Other Scenarios

In order to better understand where such an application might be useful and in which cases the application users would benefit from it, we will now present a couple of scenarios where FenceIt could be appropriately used:

- A person is travelling by bus in an unknown country or region. The journey takes a considerable amount of time and he/she would like to go to sleep, but does not know an accurate estimation of the arrival time. A solution, using FenceIt, would be to create an Alarm that gets triggered when the user arrives within 20km of the destination and an Action that would notify the user with sound and vibration.
- Students have a mandatory rule to have their mobile phones silent during classes. However, because of this rule, most of them forget to set it back on the general profile when they leave the university, thus missing important calls. This issue would be easy solvable with FenceIt if the students set up an Alarm that gets triggered when leaving the Location at the university (using any of the four Location types) and set an Action to disable the Silent profile.
- A person is at work and needs to get reminded to make an important call as soon as he/she reaches home, but is uncertain when this will happen. An Alarm that notifies the user when he/she reaches home would be a perfect solution to the problem.
- An important business person is at home and just realizes he is out of coffee. However, he hasn't got enough time to make a trip to the supermarket just for buying coffee. In this situation, using FenceIt, he could set an Alarm that gets triggered when nearby any of the super-markets in the city. Thus, as soon as he drives by a super-market or is in the area solving other business he can get reminded to also pick up some coffee, as he is already there.

As it can be seen, FenceIt can be used in various scenarios, the ones above being just a few of them. However, as stated before, in order for such an application to be usable for long time, it has to be efficient with both the user's battery and the data connection. This is what will be studied in the following chapters.

Chapter 5: System Architecture

After we described the core concepts of FenceIt, what are its requirements and we've presented some use case scenarios, we will now focus on the architectural design of FenceIt. We will get into the details regarding the main components of the application and the communication between them.

When designing the architecture, the most important aspect was modularity. Thus, FenceIt was broken in as many independent components as it was possible, so that any modification in any of them would insignificantly affect the others. The result is that the application is divided into four components, each of them being presented in the following sections. A diagram showing the most important details of each component and the interaction between them can be seen in Figure 6.

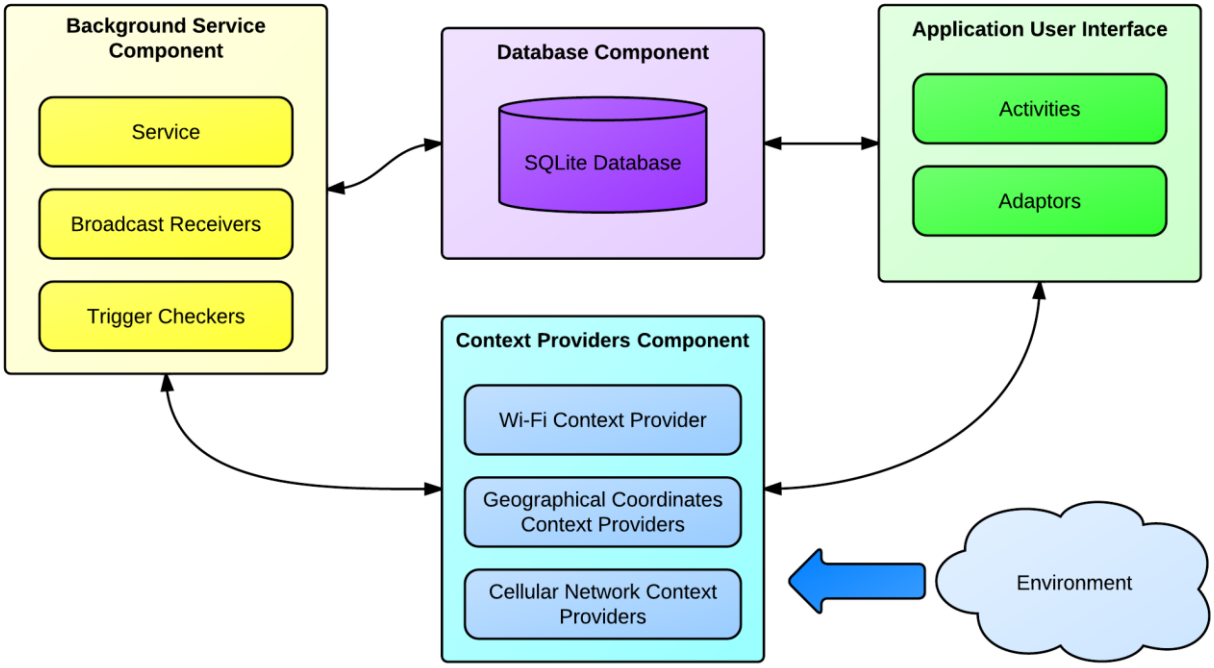

Figure 6 - Diagram of FenceIt's Architectural Design

5.1 The Database Component

This is the component that, in FenceIt, handles all interaction with the database, from any other module or component. The other application parts will not have direct access to the underlying database, all the necessary operations with the database (select, insert, update, delete) being handled by this component. What this achieves is that higher levels of control and abstractization are being enforced and that access can be synchronized through this component.

In Android, SQLite (an Open Source Database) is embedded into the operating system and supports standard relational database features like SQL syntax, transactions and prepared statements. The Database Component, however, will be the only one that is aware of this database implementation.

5.2 The Application Interface Component

This component is the one that the users will be most aware of and the one that allows them to manage their settings, alarms, triggers, locations and actions. In other words, this is the main interaction point of users with the application. The tasks that this component of the application handles are:

- Shows information to the user regarding existing alarms, triggers, locations and any other settings that have been previously made
- Offers the user the possibility to create, edit and manage new or existing alarms, triggers, locations, actions or the application settings.
- Informs the user regarding any events that show up, including when an alarm was triggered and the user has to be notified.

The application interface is composed of a multitude of modules, each handling different tasks and are presented in Figure 7.

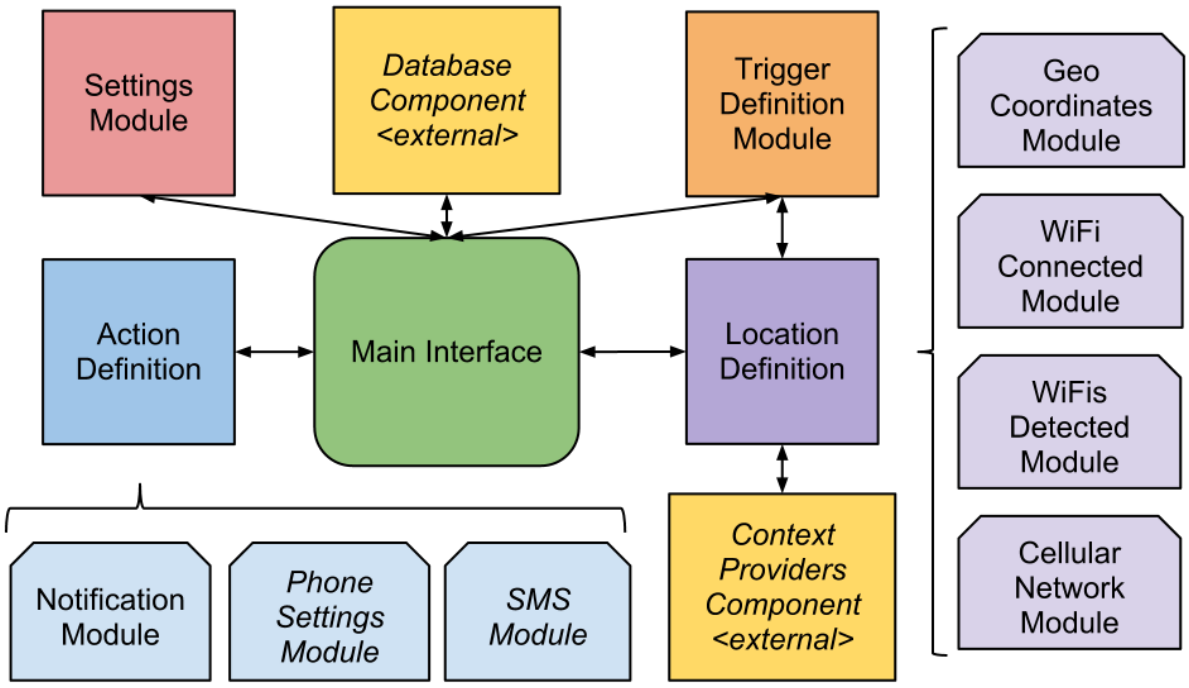

Figure 7 - Application Interface design modules

Besides the main module, which is the main interface, there are mainly four big sections, excluding the interaction with the Database Component, presented in section 5.1, and with the Context Providers Component:

- The Locations Definition, which has a module for each type of Location that the user can define. Every module requires some information from the user (e.g. pinpointing on a map for the geographical location or what is the triggering range) and gathers other contextual data from the surrounding environment (e.g. BSSID for the connected Wi-Fi network or identification details for the Cell Sites in range). All the processing done by each module is done specifically for that particular type of Location and the user interface has to be customized.
- The Triggers Definition is a module that handles the configuration of Triggers and is strongly connected with the Location Definition module.
- The Actions Definition, which has a module for each type of Action that the user can define. As above, every module allows the users to completely define the Action that should be

executed when the associated Alarm is triggered. The user interface for each of these modules is customized for the specific type of Action. As it is out of the scope of the project, not all of the Action types will be implemented.

- The Application Settings module allows the user to specify application wide settings that affect all other modules and components or the background services.

The design of the user interface must be adapted to small-screen devices and all the factors that influence interaction with an application on mobile devices.

5.3 The Background Service Component

This component is where the main logic of the application is located. It's constructed using Android Services and Broadcast Receivers, which run even when the main application activity is not started. Its purpose is to constantly scan the environment for the contextual information that is required to check if any of the enabled Alarms should be triggered and, in case any of them is set off, also handles the execution of the operations defined in Actions.

The information gathering is done on a "need-to" basis, which means that only what is needed is gathered only when it is needed. For instance, if there are no enabled alarms that use information about the Wi-Fi Networks, the wireless interface will not be queried for information. Or, as another example, the frequency at which the positioning system is queried for location data depends on the current speed, direction and distance to target and other factors. Practically, it also implements the optimization algorithms described in section 3.2.

The modules that compose the Background service are presented in Figure 8.

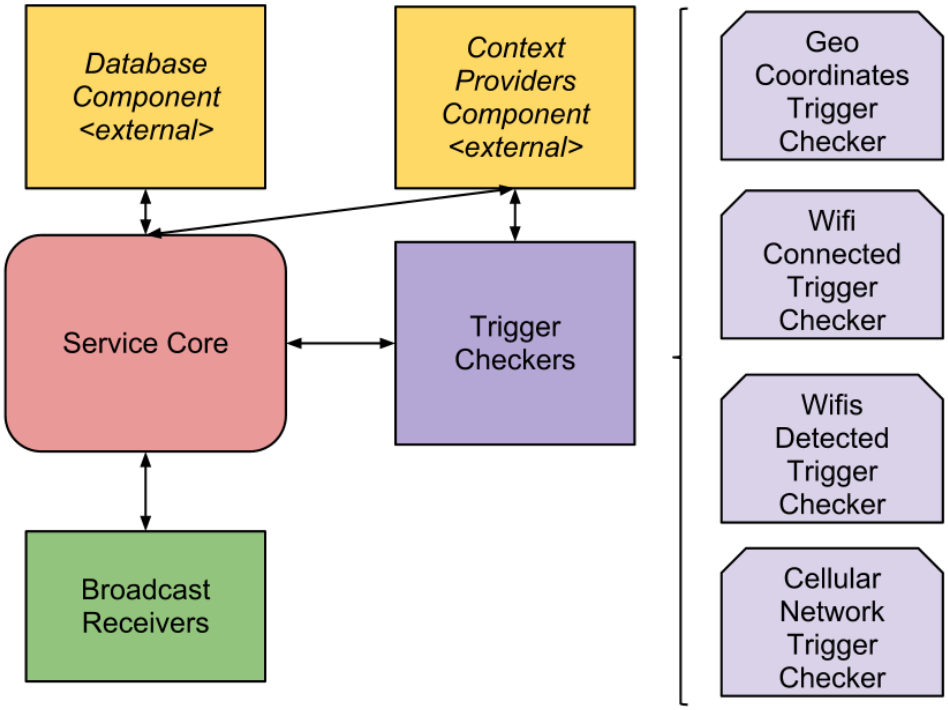

Figure 8 - Background Service design modules

Besides the main block, which contains the main logic of the service, and the interaction with the two other components, two sections can be identified:

- The Trigger Checkers section, which is composed of a module for each type of Location that the user can define. These modules, called Trigger Checkers, have the purpose of communicating with the Context Providers, of acquiring the necessary information from the environment and of checking if any of the Triggers related to Locations of their particular type should be triggered.
- The Broadcast Receivers section is composed of, as the name suggest, receivers for different events from the OS or from the application itself. These are an important part in the Background Service and we will get into more detail regarding their behavior in the next chapter.

5.4 The Context Providers Component

This component is the one that interacts with the environment and gathers the necessary contextual information. It is extensively used by the other components to acquire required information and is formed of multiple modules, one for each type of context information that can be gathered:

- The Wi-Fi module provides information gathered from the wireless interface.
- The Geographical Coordinates module uses either the GPS interface or the Coarse Network Location service provided by Android to obtain the current geographical position of the device.
- The Cellular Network module queries the operating system for information regarding the surrounding Cell Sites and the registered telephony network.

This component has to be efficient, as it will be often queried by both the application interface component and the background service.

Chapter 6: Implementation

This chapter will present the implementation of FenceIt, for the Android operating system. The development was made in order to make an accurate study and analysis of the previously presented methods for locating a mobile device and all the related aspects. Furthermore, it follows all the application requirements and features that were discussed in section 4.2.

6.1 Android concepts used in the development

Android is an open source mobile OS platform, which is based on the Linux operating system, on Apache Harmony and the Dalvik Virtual machine. Initially developed by Google it was later backed by the Open Handset Alliance and is the fastest growing operating system for mobile devices [19].

The development in Android is done using its own Java-like language based on Google-developed Java libraries. It offers the developers a very large amount of possibilities and tools when developing application for this operating system. The Android SDK is offering support for the majority of the Java SE (Java Standard Edition), with the exception of the Abstract Window Toolkit (AWT) and Swing. Instead of using AWT and Swing, the Android SDK has its own implementation for a framework for designing User Interfaces. In order to develop and run applications for Android, a developer only requires the SDK, which also includes an emulator and advanced tools that can aid the process. Add-ons where also developed for programming environments (e.g. for Eclipse) to aid the process.

Being an open-source OS for mobile devices which offers an extensive and detailed documentation, Android allows the developers to start working with ease and simplicity and the advanced facilities it provides make it an excellent target platform for an extremely wide range of application. This has made us choose Android as the development platform for FenceIt.

Before the details of the FenceIt implementation are offered, we will make a short description of some of the component types that represent the building blocks in Android. They are used throughout the implementation and their understanding is essential to the understanding of how FenceIt was developed:

- Activity - this component is used by developers to define the user interface and represents a “screen” of the application. At any one point, only one Activity can be active (and accept input from user) and Android offers very powerful tools for controlling the behavior of this component. FenceIt user interface is mainly created with the help of this component.
- Service - this component allows the application to do background service and offers the possibility to execute code even after all the UI Activities have disappeared. The user cannot interact specifically with a Service, all the required settings being made using the UI (Activities). This component is used to allow FenceIt to constantly perform trigger checks in the background.
- Broadcast Receiver - this component is used to respond to asynchronous messages received by the application either from other applications, from the operating system or even from inside the application. It is an important Android concept that is used by FenceIt in the Background Service, the Context Providers and in the Application Interface.

More details regarding how the Android OS is working and how components interact with each-other can be found in [20].

6.2 Log4J – an external logging library

Log4J is a Java-based logging framework, currently developed by the Apache Software Foundation, that allows developers to easily control the logging process in their application. Log4J is based on the concepts of Loggers (logical log names), Appenders (handle the output) and Layouts (that format the output) and is easily configurable, either from a configuration file or programmatically. More details can be found at [21].

Android-Logging-Log4J is an Open Source project that offers a wrapper of Log4J for the Android OS. Besides this, it offers some Android-specific features, such as an appender to LogCat (the Android Logging system). This project can be found at [22].

While working on FenceIt, Log4J and Android-Logging-Log4J were used to ease the development process and to offer the possibility to quickly control the logging process after the final version is released. The main purpose is to easily keep log files of the application execution and to control this process without difficulty.

6.3 Database Component – Andro-Wrapee Library

While developing FenceIt, a considerable amount of Java Objects had to be stored in Android SQLite Database and, later, to be restored into Java Objects. This has created the need to use a wrapper for the basic operations with the Database (table creation, the CRUD operations, etc.).

We initially developed a sub-component of FenceIt that would act as a wrapper between the database and the rest of the application, easing the development process. However, after a considerable amount of time and work was dedicated to this, it was decided to break this code from the FenceIt project and implement it as a separate library, which could also be used by other Android developers. This has led to the creation of Andro-Wrapee open-source project, which can now be found at [23]. As it is out of the scope of this paper, we will only make a short presentation of the facilities developed and the simplest use scenario.

The main purpose of Andro-Wrapee is to allow developers to easily store entries in the database (and restore them later), without interacting with the database directly and without writing any SQL queries. The library automatically handles the transformation of Java Objects into the appropriate parameters for a SQL query to insert, update, delete an entry in the database or parses the query results, creating a corresponding Java Object, in the case of a Get (SELECT) operation. In order to aid the library on how to make the transformations (correlations), the developer has to add some Java Annotations to the Java classes that need be stored in the database. They will be presented later in this section.

The three important classes of this implementation are DefaultDAO, DefaultDatabaseHelper and ReflectionManager, which are generic classes and offer the basic operations/features necessary for managing the database tables and creating, reading, updating and deleting entries from tables.

The *DefaultDatabaseHelper* is an extension of *SQLiteOpenHelper* and handles the automatic manipulation of SQLite tables (creation, upgrade). Besides other parameters, initialization of the *DefaultDatabaseHelper* requires the classes that should be stored in the database and the table names for each class. A basic usage example is:

```
SQLiteOpenHelper dbHelper = new DefaultDatabaseHelper(context, DATABASE_NAME,
DATABASE_VERSION,
    new Class[] { Alarm.class, ... }, //The classes to store in the database
    new String[] { "alarms", ... }); // The table names for the classes in the database
```

The *ReflectionManager* is used to handle the Java Reflection operations on the objects. It is initialized with the required class and goes through every field in a Java Class, checks whether it should be in the Database and any other conditions and speeds up the Reflection process by caching some results. In order to get an instantiated *ReflectionManager*, the following code is used:

```
ReflectionManager rm = new ReflectionManager(Car.class);
```

As mentioned before, to allow the library to handle the correspondence between the Java Objects and the corresponding columns in the SQLite Database, the library user has to annotate the classes he wants to store in the Database. For this, the following Annotations can be used:

```
@DatabaseClass  // marks a class as corresponding to a Database Table
@DatabaseField  // marks a field that will be stored in the table and which corresponds to a
column
@IdField         // marks a field that is considered to be the id (primary key) in the Table
@ReferenceField // marks a field that stores an entity which forms a relationship with this
one
```

An example of a basic class that is annotated is:

```
@DatabaseClass     //required annotation for classed stored in database
public class Car {
       @IdField
       public long id; //will be stored in the database as the primary key of the table

       @DatabaseField
       private String name; //will be stored in the database

       @DatabaseField
       protected Date producedOn; //will be stored in the database

       public ArrayList<Color> availableColors; //will not be stored in the database, as it
doesn't have the required annotation
}
```

Andro-Wrapee also adds support for inheritance and, when parsing a class annotated with *@DatabaseClass,* it also recursively explores the super-classes, as long as they also have the *@DatabaseClass* annotation. This is highly used in FenceIt, as it allows us to store a class into the database, even though some fields are members of abstract parent classes.

The DefaultDAO<T> is the class that actually handles the CRUD operations with the database. When constructed it requires a *ReflectionManager*, a table name and other required parameters. The *ReflectionManager* is not instantiated in the *DefaultDAO* implementation to allow developers to make some caching (for example singletons), as the Reflection step in the construction of a *ReflectionManager* is computational expensive. In order to instantiate and use *a DefaultDAO*, similar code can be used:

```
DefaultDao<Car> dao = new DefaultDAO<Car>(Car.class,  // the class it handles
                              dbHelper,      // a SQLiteOpenHelper to manage the database
                              rm,            // the reflection manager
                              "cars");       // the table name
Car newCar=new Car();
...
dao.open();
long id=dao.insert(newCar); newCar.id=id;
dao.close();
...
dao.open();
dao.update(newCar, newCar.id);
long carID=...
Car otherCar=dao.fetch(carID);
dao.close();
...
```

As described, the usage of Andro-Wrapee library for handling operations with the Android SQLite Database is simple and intuitive and helps speed up the development process. More details regarding the implementation and usage can be found in the source code, provided at [23].

In FenceIt, all the entities that have to be stored in the database (and which are presented in the following sections), are stored using Andro-Wrapee and, as required, are annotated accordingly. Also, in order to speed up the process, a *DatabaseManager* class has been developed, that stores (caches) some already initialized *DefaultDAO* objects. Whenever a DAO for a class type is requested, if it was already initialized, it is returned directly, skipping the computationally-intensive process of initialization.

6.4 Core Concepts

The implementation of the application was done with modularity, extensibility and ease of maintenance in mind. The project is broken into multiple modules, each of them handling a sub-section of the entire application.

In this paper sub-section, the details regarding the general aspects of the application are given, while in the following sections the paper will go into the details regarding the application graphical interface, context data providers and the background service.

As mentioned in the previous chapters, there are a couple of concepts the application is working with. A brief description is provided below, after which the paper will get into the implementation details for each of them:

- **Alarm** - the entity which contains a collection of conditions (Triggers) and operations (Actions)
- **Trigger** - stores information regarding a condition that has to be valid for an alarm to go off
- **Location/Fence** - entity that stores information regarding a specific place, so that the application can identify when the device is in that area and when it is not.
- **Action** - defines how the application should behave when an alarm is triggered.

A class diagram with the most important fields and methods and the relationships between classes is provided in Figure 9. It has to be mentioned that the diagram provides a simplified view, with getters, setters or other non-relevant fields and methods not being shown.

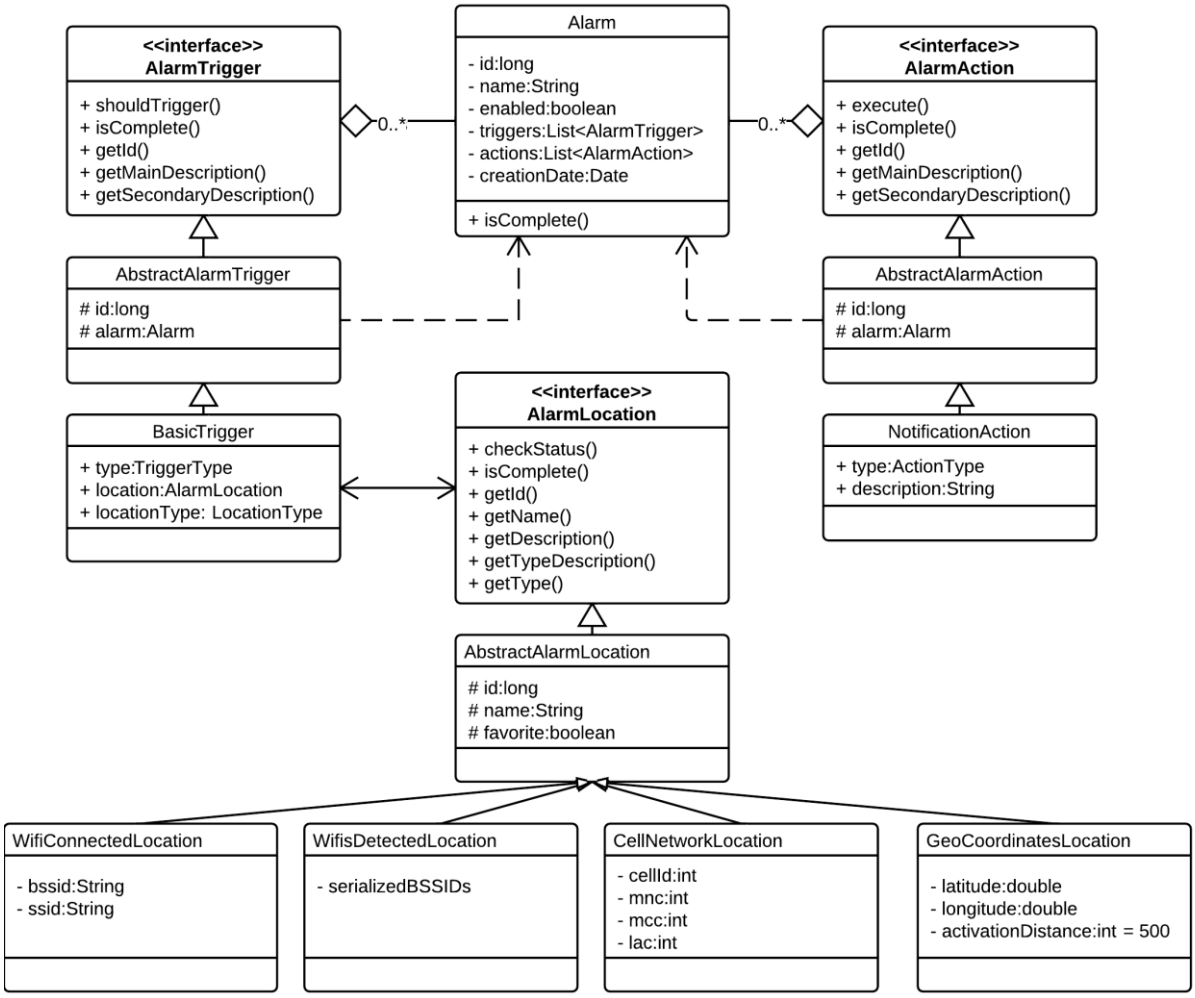

Figure 9 - Class Diagram for core concepts of FenceIt

6.4.1 Alarms

In FenceIt, an Alarm corresponds to the Alarm class (com.fenceit.alarm) and stores all the required information about how it should be shown, when it should be triggered and what happens when the alarm is triggered.

The name is added by the user and has the purpose of easing the identification process between alarms. The *enabled* field marks if the alarm is active and its triggers should be checked for matching conditions. The *triggers,* which implement the *AlarmTrigger* interface and which need to be checked with current contextual information if they should go off, are stored as a list. The *actions* list stores the actions which are executed (performed) when the alarm is triggered. The *creationDate* field only has the purpose of storing the exact moment when this alarm was created and is used in various algorithms for sorting the alarms and also for the UI.

Alarms are stored in the database, in the table "*alarms"*, so the class has an *id* field and is annotated with *@DatabaseClass*. All the other fields that should be stored in the database are also, as required, annotated with either *@DatabaseField* or *@ReferenceField.* As the *triggers* and the *actions* are Lists, they will not be stored in the Database in the '*alarms'* table, but in their own separate tables and a reference will be kept to the corresponding alarm.

6.4.2 Triggers

In our application, a Trigger is an implementation of the AlarmTrigger (com.fenceit.alarm.triggers.*) interface and is the entity that stores everything that is required to check if an alarm should be triggered, given some contextual information. It has to be mentioned that, in FenceIt, the triggering mechanism is defined such that if any of the Triggers associated with an Alarm is fired, then the Alarm is set off. (OR behavior).

To ease the implementations of various types of triggers, an AbstractAlarmTrigger was implemented, which stores the trigger *id*, the corresponding *Alarm* of this trigger and also implements some methods.

The only implementation for an AlarmTrigger used in FenceIt (in the current version) is the BasicTrigger, which is based on when (e.g. enters, leaves) the alarm should go off. The possible moments when a trigger can be set off are defined in the TriggerType enumeration:

```
public enum TriggerType {
        /** The triggering occurs when entering the Location. */
        ON_ENTER,
        /** The triggering occurs when exiting the Location. */
        ON_EXIT
    };
```

Thus, a BasicTrigger holds a reference to a Location (which will be presented in the following section), in the *location* field and a type of triggering, in the *type* field. Besides these, the BasicTrigger also hold a LocationType field (*locationType*), which is used to distinguish the different types of Locations and optimize the fetching process (more details in the next section). In the *shouldTrigger* method it checks, using the provided contextual information, if it should be triggered or not.

The BasicTriggers are stored in the database, in the table "*triggers*", so the class has an *id* field and is annotated with *@DatabaseClass*. All the other fields that should be stored in the database are, as required, annotated with either *@DatabaseField* or *@ReferenceField.* As the Location is a custom object (and, as we will see, with multiple implementations), only its reference (the id) is stored in the *triggers* table, all the other details being stored in separate tables (as we will see in the following section).

6.4.3 Locations

Locations are implementations of the *AlarmLocation* (com.fenceit.alarm.locations.*) interface and, as presented before, are of four different types. An *AlarmLocation* object stores all the required data so that, when appropriate contextual data is provided, it can decide whether the device is or is not in that particular Location/Fence.

As mentioned, in FenceIt there are multiple Location types used and all the theoretical aspects regarding them was presented in previous chapters. When talking about implementation, the LocationType enumeration, containing each of the four location types (Wifi Connected Location, Wifis Detected Location, Cell Network Location and Geographical Coordinates Location), was defined:

```
public enum LocationType {
    /** The Geo Coordinates location, based on geographical coordinates. */
    GeoCoordinatesLocation(0),
    /** A location based on the connected Wifi. */
    WifiConnectedLocation(1),
    /** A location based on the detected Wifi(s) network(s). */
    WifisDetectedLocation(2),
    /** A location based on the visible Cell Network Sites and the Cell Site connected to. */
    CellNetworkLocation(3);
}
```

To ease the implementations of various types of locations, the *AbstractAlarmLocation* was implemented, which stores the *name* of the location, in case it has one, if it is a *favorite* and should be persisted even after the corresponding trigger is deleted and the *id* of the location. Besides this, it also implements some common methods.

All the classes that extend the AbstractAlarmLocation (thus implementing AlarmLocation) are stored in the database, in a corresponding table, so the abstract class has an *id* field and is annotated with *@DatabaseClass*. Each of the extending classes also is annotated with *@DatabaseClass* and all the other fields that should be stored in the database are, as required, annotated with *@DatabaseField*. Regarding the storing of AlarmLocations in the database, as mentioned in the previous sub-section, each BasicTrigger has a reference to the id of the corresponding alarm and a LocationType. The location type is used to identify in which table the AlarmLocation is stored and the reference field is used to identify the particular location in that table. This abstraction allows the BasicTrigger implementation to be completely independent of the AlarmLocation implementations, thus the addition of a new location type is easy. For fetching an AlarmLocation from the database, to fill in a BasicTrigger, the class *AlarmLocationBroker* is used, which handles the correspondence between the location id and type and the corresponding table and entity in the database.

Now, the paper will go into the implementation details for each of the four Location types, shortly describing what each AlarmLocation stores and how it checks if the device is or is not at the Location. A diagram depicting the general process, for any type of alarm, can be seen in Figure 10.

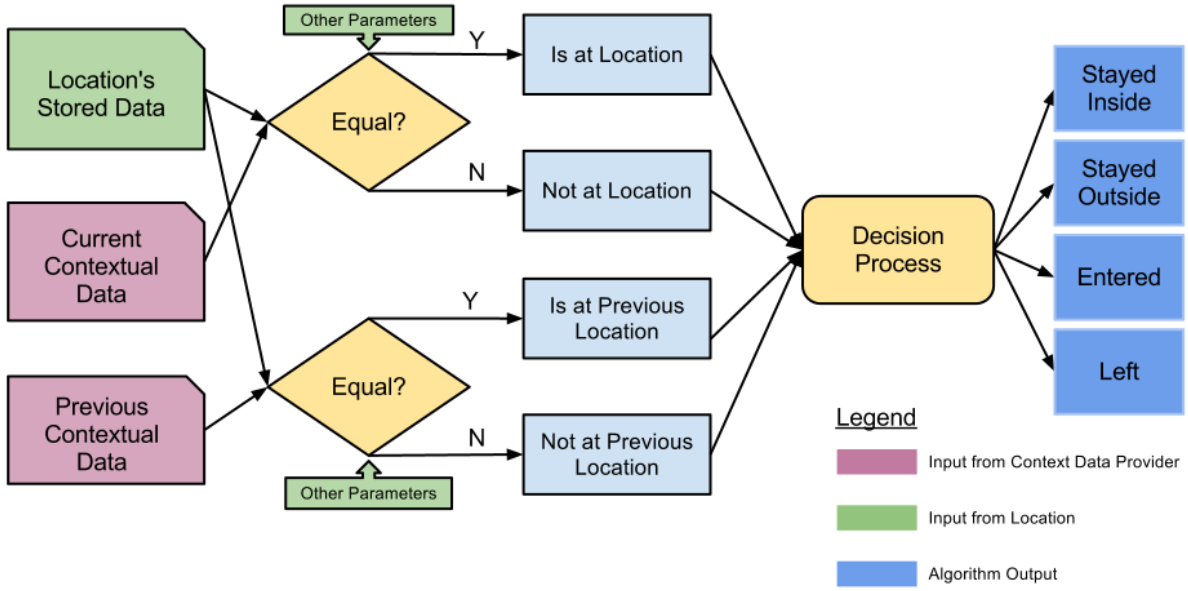

Figure 10 - Process diagram for Locations

6.4.3.1 Wifi Connected Location

This location type is implemented in the WifiConnectedLocation class and corresponds to a Location based on the Wi-Fi Network to which the device is connected. As described in Chapter 3, it is based on the unique BSSID of the connected Wi-Fi Network. Thus, it stores the BSSID of the Wi-Fi network to which it was connected when the alarm was defined and the SSID, for a human-readable name for the user. In the database, it is stored in the table *"wificonn_locations".*

When the application needs to check whether it is or it is not at the Location, it compares the stored BSSID to the BSSID provided by the Context Data Provider (now or at the previous query). There are no "Other Parameters" for this Location type.

6.4.3.2 Wifis Detected Location

This location type is implemented in the WifisDetectedLocation class and corresponds to a Location based on the Wi-Fi Networks which are detected by the Wireless interface of the device. As the paper describes in Chapter 3, it relies on the unique BSSID of all the Wi-Fi Networks that can be detected by the device using the Wireless interface. Thus, it stores the BSSIDs of all the Wi-Fi Networks that were in range when the alarm was defined. To optimize database storage, the BSSIDs are stored serialized, as a single String. In the database, it is stored in the table named *"wifisdetec_locations".*

When FenceIt checks whether it is or it's not at the Location, it compares the stored BSSIDs to the BSSIDs provided by the Context Data Provider (now or at the previous query). In order to accommodate a certain degree of errors, a MATCH_PERCENT constant is used, which defines the percent of the Wi-Fi Networks in the Location that have to match the Wi-Fi Networks provided by the Context Data Provider. This is represented in the diagram as "Other Parameters".

6.4.3.3 Cell Network Location

This location type is implemented in the CellNetworkLocation class and corresponds to a Location based on the identification details received from the Cellular Network. As it was detailed, it uses four unique values provided by the GSM network: Mobile Country Code, Mobile Network Code, Location Area Code and Cell Id (or three similar values in the case of CDMA mobile networks). Therefore, it stores these four values as they are obtained when the alarm is defined. In the database, it is stored in the table "*cell_locations".* In the case of CDMA networks, the correspondence presented before is used.

When the application needs to check whether it is or it's not at the Location, it compares the stored four values with the values provided by the Context Data Provider (now or at the previous query). There are no "Other Parameters" for this Location type.

6.4.3.4 Geographical Coordinates Location

This location type is implemented in the CoordinatesLocation class and corresponds to a Location based on the current geographical position of the device. As described in Chapter 3, it uses the latitude and the longitude obtained from the Android's Location Manager to position the device within a certain radius of a point. Therefore, it stores both the latitude and the longitude of the definition point and a radius (in meters), which is used to check if the device is or is not within range of that point. In the database, it is stored in the table *"coordinates_locations".*

In FenceIt, when it is needed to check whether the device is or is not at the Location, it computes the distance between the stored geographical point and the one provided by the Context Data Provider (now or at the previous query). The "other_parameters" parameter is represented by the triggering radius.

6.4.4 Actions

In FenceIt, an Action is an implementation of the AlarmAction (com.fenceit.alarm.actions.*) interface and is the entity that stores all the details that are required to execute an already defined of operations, when an Alarm is triggered. Each Alarm can have multiple Actions associated, each of them being executed when the alarm is set off because of one of its triggers.

To ease the implementations of various types of actions, an AbstractAlarmAction was implemented, which stores the action's *id*, the corresponding *alarm* and also implements some methods.

As the actions are out of the scope of this paper, the only implementation for an AlarmAction used in the current version of FenceIt is the NotificationAction, which, when executed, shows a new Activity to the user with a pre-set message (stored in the *description* field) and notifies the user with a specific sound.

All the classes that extend the AbstractAlarmAction (hence implementing AlarmAction) are stored in the database, in a corresponding table, so the abstract class has an *id* field and is annotated with *@DatabaseClass*. Each of the extending classes also is annotated with *@DatabaseClass* and all the other fields that should be stored in the database are, as required, annotated with *@DatabaseField* or *@ReferenceField.*

6.5 Context Data Providers

As mentioned in Chapter 5, an important component of FenceIt is the one that handles gathering of contextual information from the surrounding environment, with the purpose of checking if any of the Alarms (more precisely any of the triggers) should be set off or not. The component that handles this in FenceIt is built of Context Data Providers which are implemented in the classes of the *com.fenceit.providers* package.

One of the most important characteristic of these Context Data Providers is that they have to gather the required data as quickly as possible and with minimum battery consumption, as they will be used extensively by both the Application Interface (UI) and the Background Service which is constantly analyzing whether any of the alarms should be triggered.

To get into the details of how Context Data Providers work, we'll use the diagram in Figure 11, which depicts the general process of gathering data.

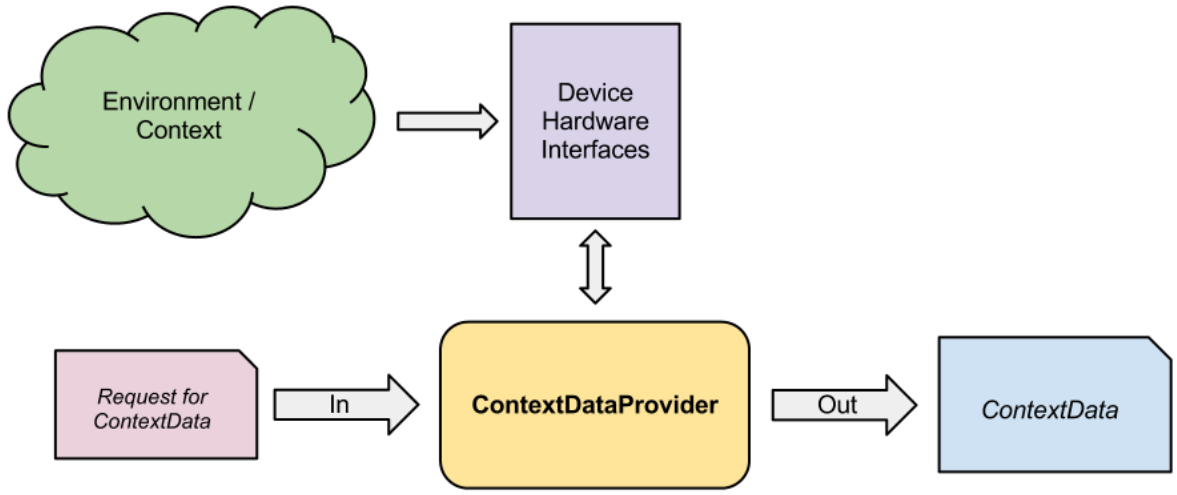

Figure 11 - Process diagram for a Context Data Provider

As it can be noticed from the diagram, the process of gathering Contextual Data from the surrounding environment starts with a request to the ContextDataProvider for information regarding the current environment (i.e. a method call). Then, the Provider queries the device Hardware and the Android OS for the necessary information from the environment. When the Interfaces/OS reply, a ContextData object is returned, which is then used by the application. In most cases, it is given to the BasicTriggers and the AlarmLocation implementations which use it to check if the device is or is not currently at a particular Location. The ContextData is, in FenceIt, an interface which is implemented by the classes that want to store contextual data.

In FenceIt, there are three Context Data Providers developed, namely WifiDataProvider, CoordinatesDataProvider and CellDataProvider and four Context Data implementations, concretely WifiConnectedContextData, WifisDetectedContextData, CoordinatesContextData, CellContextData.

6.5.1 Wifi Context Data Provider

The WifiDataProvider is used for gathering contextual information using the Wireless Interface, and is used for getting data regarding both the Connected Wifi (a WifiConnectedContextData is returned) and the Detected Wifis (a WifisDetectedContextData is returned). The Android Manager that is used is the *WifiManager.*

In the case of detected Wifi networks, the newest Wifi Scan Results are only available after a Wifi Scan is completed. Thus, in order to get the most accurate results, for a proper use of the provider, a *startScan()* must be issued before the contextual data is actually requested. After the *startScan()* method is used, the Android OS starts a scan and, when the results are available, a *WifiManager.SCAN_RESULTS_AVAILABLE_ACTION* broadcast intent is sent in the system. In order to catch and process such an intent, a BroadcastReceiver can be registered, which could be used to query the Context Data Provider for the results, using *getWifisDetectedContextData().* An example of such a receiver is:

```
private class WifiScanReceiver extends BroadcastReceiver {
	@Override
	public void onReceive(Context context, Intent intent) {
		ContextData c=WifiDataProvider.getWifisDetectedContextData();
		//use c ...
	}}
```

A method to dynamically register such a Broadcast Receiver (for example in the UI), is by using intent filters:

```
BroadcastReceiver receiver = new WifiScanReceiver();
IntentFilter filter = new IntentFilter();
filter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
registerReceiver(receiver, filter);
```

As another solution, this registration could also be done at compile time, using the Android Manifest.

6.5.2 Cell Network Data Provider

The CellDataProvider is the Context Data Provider that is used for gathering information from the environment regarding the Cell Network and the Cell Site to which the device is currently connected to. In order to do this, it uses the Android *TelephonyManager*, and, depending on the PhoneType (GSM or CDMA), uses either GsmCellLocation or CdmaCellLocation. Using the information from these, it fills in a CellContextData object that is returned to the provider requester.

6.5.3 Geographical Coordinates Data Provider

The Context Data Provider handling the geographical coordinates is the CoordinatesDataProvider. As it uses Android's Location Manager, which is asynchronous, using this provider is also done in an asynchronous mode. For this, the provider is using the concept of listeners/observers and the *CoordinatesLocationDataListener* interface is employed*:*

```
public interface CoordinatesLocationDataListener {
    public void onLocationUpdate(Location location);
}
```

The components that require data gathered from the environment have to first register themselves as a listener to the provider, using:

```
public void addCoordinatesLocationDataListener (CoordinatesLocationDataListener listener,
Context context)
```

When the first listener is added, the provider automatically starts listening for location updates and as soon as it receives a good location, is notifies all the listeners using the *onLocationUpdate()* method. When the component that started listening is no longer interested in location updates, it has to remove itself, using:

```
public void removeCoordinatesLocationDataListener (CoordinatesLocationDataListener listener)
```

To provide an accurate response to queries, the provider is using all the available Android Location Providers (if possible, both the *GPS_PROVIDER* and the *NETWORK_PROVIDER*), with dynamic values between updates for each of them. To provide the best possible solution the CoordinatesDataProvider stores the best Location found so far and, when a new Location is provided by the Android system, it compares it with the existing one, storing the more accurate one. At any request for Context Data, a CoordinatesContextData object is built using the most accurate Location found so far, if any was found at all.

6.6 Application Interface

The application interface of FenceIt was developed in a consistent way, while trying to be as intuitive and as easy-to use as possible. In this way, an inexperienced user, who is encountering the application for the first time, should have no problem in easily and quickly learning to use it.

A diagram showing all the available activities and the interactions between them can be seen in Figure 12. The diagram also shows, besides the Activities (shown in blue), the Context Data Providers (in violet), the Brokers for *LocationType* and *ActionType* (in green), the Broadcast Receivers implemented for UI (in yellow) and the Notification activity shown when a Notification action is triggered (in red).

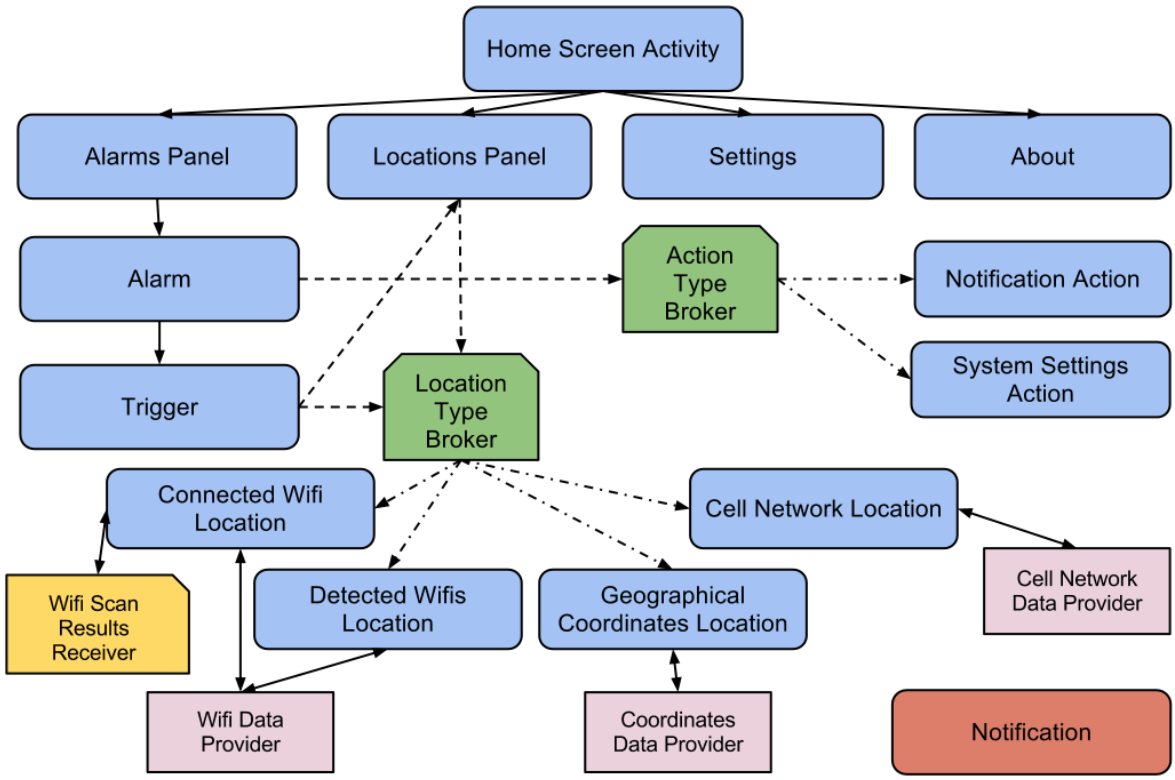

Figure 12 - Diagram of interaction in FenceIt user interface

One common element in all the interface activities is the action bar, present at the top of the screens (e.g. in Figure 13). It has the purpose of presenting the title of the current activity, a navigation button (usually to the Home Activity) and, optionally, a button (e.g. a "Save" button). This action bar was developed without the features from Android 4.0 in order to keep the compatibility of the application to a version of Android as old as possible.

Figure 13 - FenceIt action bar

Another very important aspect regarding the development of the user interface of the application is the fact that, to maintain a uniform aspect of the UI and to make any eventual change as easy as possible, wherever it was possible we used the extremely powerful Android mechanism of resource files. Thus, we used files for storing commonly used values:

Table 1 - Resource files usage

res/values/colors.xml	Store commonly used colors and color schemes
res/values/dimens.xml	Store sizes/dimensions for commonly used elements
res/values/string.xml	Some commonly used String values, that appear in the UI
res/values/styles.xml	Various styles for use for different Views

Besides this, in the *res/*drawable folder, definitions for some commonly used graphical elements were made, for example for the setting item on the Editing Activity for any entity or for the elements of the action bar.

The paper will now make a short description of the Activities developed for FenceIt and provide implementation details regarding the most interesting aspects.

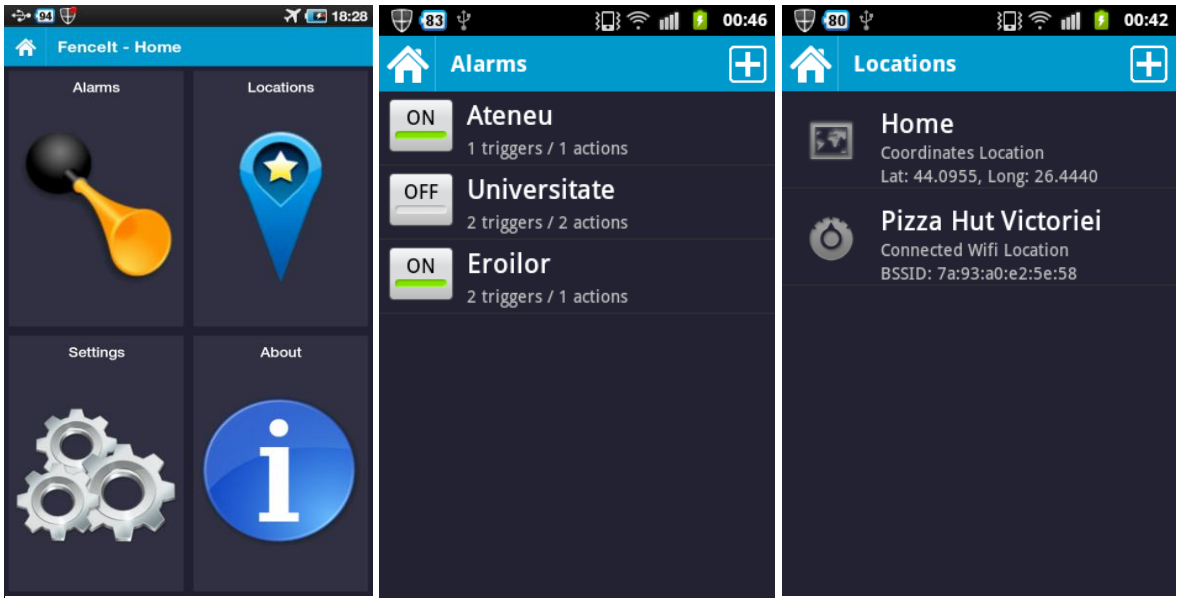

Figure 14 - FenceIt UI: Home Screen, Alarms Panel and Locations Panel

6.6.1 Home Screen

The Home Screen, implemented as the HomeActivity, is the first screen the user interacts with when the application is started. It can be seen in Figure 14. It is developed in the style of a dashboard and follows the development guidelines regarding the design of User Interface for mobile devices [24]. It provides four links to the following screens: Alarms Panel, Locations Panel, Settings and the About Screen.

6.6.2 Alarms Panel / Locations Panel

These two panels have the purpose of showing the user a list of the existing Alarms and the Locations he/she has marked as Favorite. Besides this, the user can add new entities, by clicking upper right corner button, and delete existing ones, by long press on any item. They can be observed in Figure 14.

From the implementation point of view, these activities (AlarmsPanelActivity and LocationsPanelActivity) are composed of the action bar, with the add button, and a ListView, which shows the existing alarms/locations. Custom Adapters are used for the display of the elements.

6.6.3 Editing Alarms / Triggers

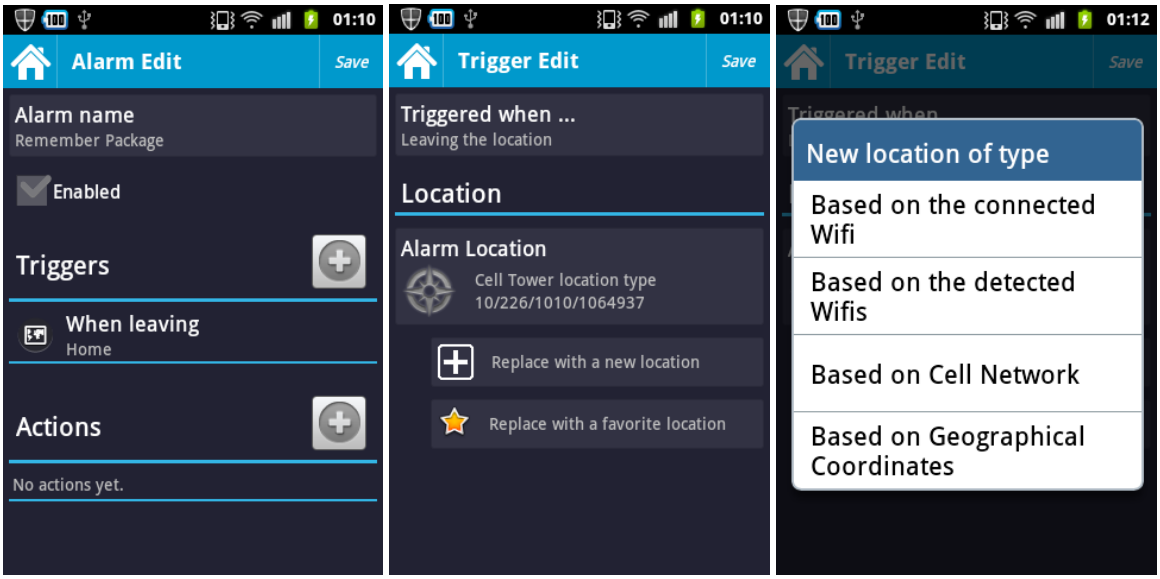

Figure 15 - FenceIt UI: Editing an Alarm, a Trigger and selecting Location type

When the user clicks on an Alarm from the AlarmsPanel or wants to create a new one, he/she is shown the Alarm screen (AlarmActivity), which shows the settings the user can make to that particular entity. He/She can edit the name of the alarm or if it is enabled and is able to add/delete/edit Triggers or Actions corresponding to the current Alarm. Some screenshots can be observed in Figure 15.

Similarly, when the user selects a Trigger from the Alarm screen or wants to create a new one, it is taken to the Trigger screen (TriggerActivity), where he/she can manipulate all the properties of that particular BasicTrigger.

As it can be seen, throughout the application, when a property needs to be set up, we made the decision to not embed the input control in the Activity. Instead, we use an approach similar to the one used by Android in the Phone Settings. When the user first enters a screen where he can set a property, he only sees a section that textually describes the property and its current value. When he/she clicks this section, a Dialog appears, where he/she can set the new value. Implementation-wise, this is done by using a simple vertical LinearLayout with two TextViews, one with the description/name of the property and one with the set value. This LinearLayout is set up as focusable and clickable and whenever the user clicks on it, a selection dialog is created. The description text is usually filled in dynamically, based on the previously set values. An example layout code is shown in the following listing (some properties were purposely eliminated):

```
<LinearLayout android:id="@+id/trigger.whenSection"
      style="@style/SettingItem"
      android:clickable="true"  android:focusable="true">

      <TextView
            android:text="Triggered when..."
            android:textAppearance="?android:attr/textAppearanceMedium" />

      <TextView android:id="@+id/trigger.whenText”
            android:text="Default value. Will be filled programatically."
            android:textAppearance="?android:attr/textAppearanceSmall" />
</LinearLayout>
```

6.6.4 Locations

As there are multiple types of Locations that can be used, a different approach was taken when setting up the Location for a Trigger.

To provide extensibility and accommodate any future implementations of other Location Type, the *LocationTypeBroker* was implemented. This class is aware of what types of Locations are implemented and generates the required graphical elements (e.g. an adapter for the user to select a location type or the required Intent to start the Activity corresponding to a partiular type). In this way, on the Trigger Activity, when the user wants to create a link to a new Location, the *LocationType*Broker is queried for a list of all the available LocationTypes and their description and a selection dialog is offered to the user. After he/she has selected the required LocationType, the *LocationType*Broker is again queried for the Intent that can start the Activity corresponding to that type.

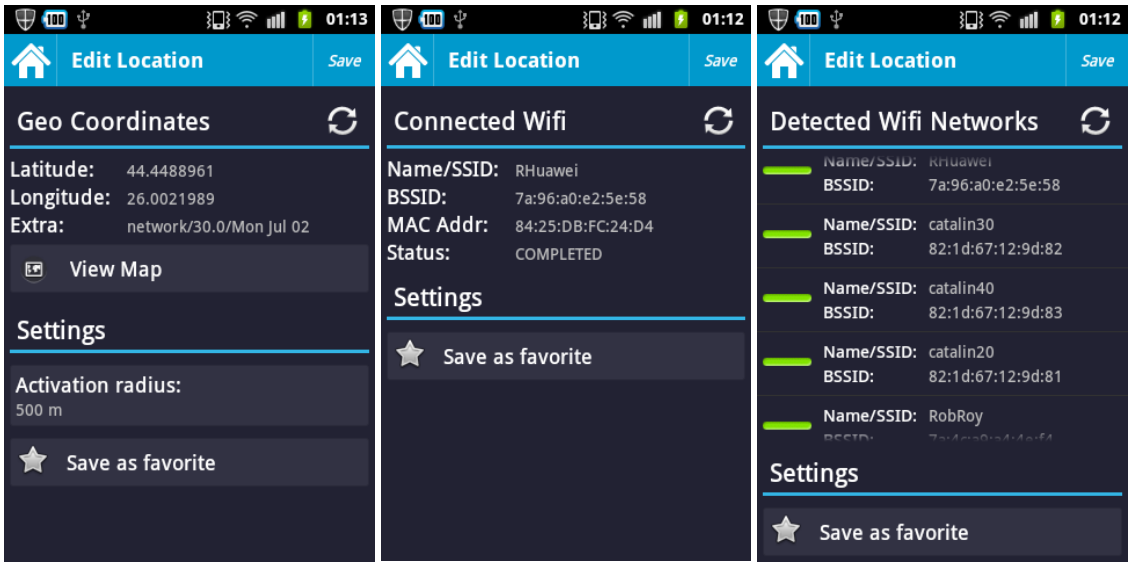

Figure 16 - FenceIt UI: Coordinates Location, Wifi Connected Location and Wifis Detected Location

The corresponding Activities to each Location Type can be seen in **Table 2** and some screenshots for each type can be seen in Figure 16.

Table 2- Correspondence between Location Type and Activity

Geographical Coordinates Location	CoordinatesActivity
Wifi Connected Location	WifiConnectedAcivity
Wifis Detected Location	WifisDetectedActivity
Cell Network Location	CellNetworkActivity

As it can be noticed from the screenshots, on each Activity the user can initially view the already existing information defining that particular Location, if any. If he/she clicks the refresh button, the corresponding Context Data Provider will be queried for information and, depending on the type of data requested, the response will be available immediately or in a couple of seconds. As soon as new data is available, the Activity is refreshed with the new information. However, the data is only saved in the Location object (and in the database) when the user clicks the Save button.

Another aspect that has to be mentioned is regarding the option of saving a Location as a Favorite. When the user marks a Location as Favorite (by clicking the corresponding button), a name has to be provided for the Location and it is then saved in the Database for future use. If a Location is not saved as a Favorite it is not available when defining other Alarms/Triggers and as soon as the corresponding Trigger is deleted, it is also deleted from the Database. Implementation-wise, this favorite functionality is implemented with the help of the *favorite* field in the AbstractAlarmLocation class. This field is also used in the queries necessary to fill in the Locations Panel (only Locations marked as Favorite appear on that screen).

6.6.5 Actions

The design of the user interface for the Actions related Activities is similar, in conception, to the one for the Locations. In this case a broker was also implemented, *ActionTypeBroker,* which has the same purpose as in the previous case: it handles the connection between the implemented Action Types and the corresponding descriptions, adaptor for user selection and Activities.

Also related to the Actions is the NotificationActivity, which is shown to the user when an Alarm containing a NotificationAction is triggered. Completely non-related to the other components of the FenceIt user interface, this Activity shows the user which Alarm was triggered and the description message saved by the user when creating the Action. If the user clicks anywhere on the screen, the NotificationActivity is closed and the user is taken back to its previous Activity.

6.6.6 Settings and About

Starting from the Home Screen, the user also has access to two more Activities: SettingsActivity and AboutActivity. The Settings screen, implemented as a class extending PreferenceActivity, shows the user various application wide settings. The About screen is a very simple Activity that shows the user details about the version of the application, the developers of the activity and contact information.

6.7 Background Service

FenceIt's Background Service is a very important component of the application and has the role of constantly checking the environment for conditions that could trigger any of the enabled Alarms. As it is a component that is constantly running, two of the most important characteristics it should possess are: should be very efficient with both the memory used and the battery consumption and should have the ability to restart even when killed by the Android OS, so as not to miss any Alarms set by the user.

In order to address the second requirement, the service was implemented as an unbounded Android service, running in the Foreground. This has the advantages of allowing the service to permanently show a notification in the Android top bar and, according to [20], makes the service highly unlikely to be killed by the operating system.

The main concepts the Background Service is relying on are Broadcast Receivers and Android's System Alarm Manager. Android's Alarm Manager [20] is a service provided by the OS that allows the developers to schedule an Intent to be delivered to the application at a specified time in the future. These Intents are then captured and processed by registered Broadcast Receivers. The main advantage of using the Alarm Manager in the Application is that it allows scheduling at fixed times and that the Intents can be delivered even if the device is asleep.

Android's Alarm Manager Service and the Broadcast Receivers allow FenceIt's Background service to check if at a given time any of the alarms should be triggered and then schedule a new check after a

dynamically computed time period. In the meanwhile, the application (unless the UI is active) is not doing any processing and the device can, for example, go to sleep or handle other applications. What also has to be mentioned is the fact that the service schedules different alarm for the various Location Types, thus allowing some Triggers to be checked more often than others and providing flexibility.

As with other FenceIt components, the service was designed so that it allows the addition of new Actions/Locations without any major code changes. The functionality is broken into three modules that communicate with each other:

- The main module which coordinates the Background Service.
- The Broadcast Receivers, that listen for any intents requested by the application or for any other broadcast intents relevant to the application.
- The Checker Threads, which, when run, checks if any of a particular set of Triggers should be set off.

The paper will now describe the details regarding the functionality of the modules. A diagram of how the service modules interact with each other is shown in Figure 17.

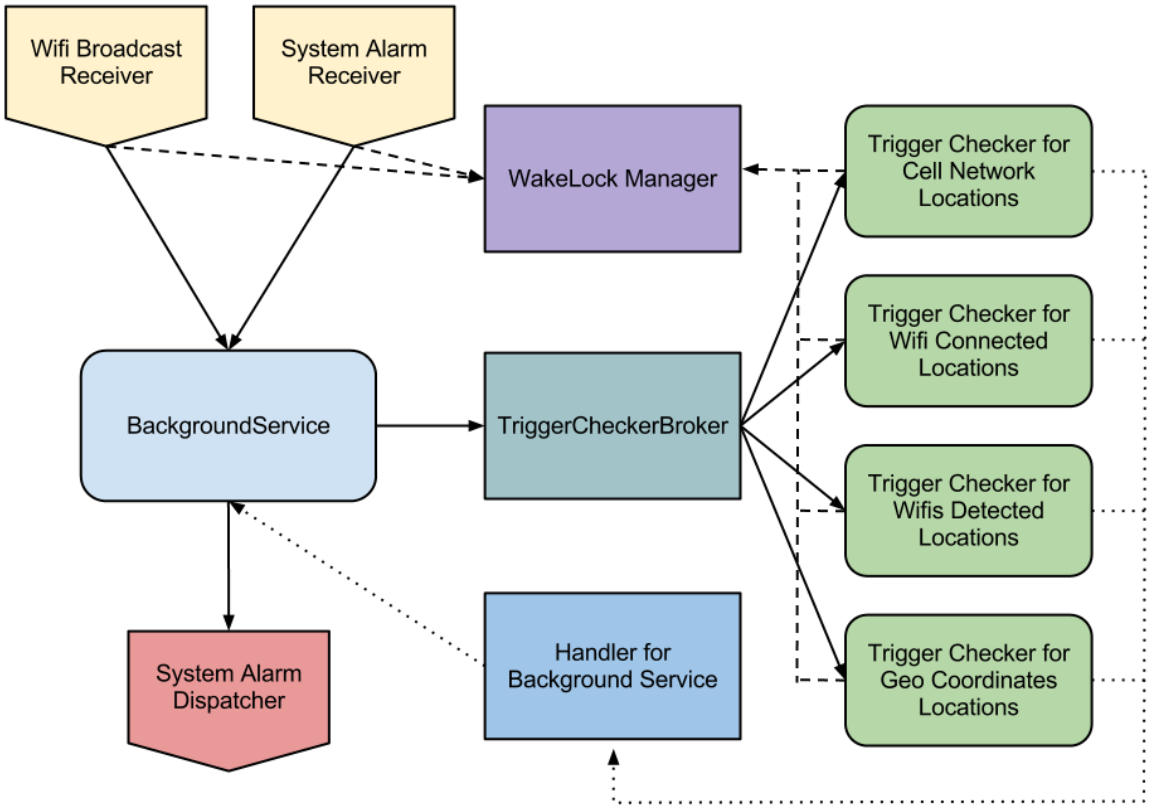

Figure 17- Process diagram for the background service

6.7.1 Broadcast Receivers

As briefly presented before, the service relies on the Android's Alarm Manager to schedule the delivery of some Intents to the application. When those Intents arrive, they are processed by the *SystemAlarmReceiver* which, in turn, notifies the service that a new analysis of triggers containing Locations of a specific type should start. This also happens on the arrival of other system specific Intents, such as the Intent delivered by the OS when a Wifi Scan is completed ("android.net.wifi.SCAN_RESULTS").

When scheduling an Intent to be delivered to the application, using the Alarm Manager, the developer can set up data in the *Extra* field. This mechanism is used in FenceIt to add an "Event Type" to the Intent. When the *SystemAlarmReceiver* gets the Intent, it parses the extra field searching for that type and sends it to the Background Service (inside a new Intent), which, in turn, starts the appropriate check for triggering conditions. To communicate with the Service, the Broadcast Receivers use one of the Android's built-in systems for communicating with unbound services: *startService()*. This command, also used to start the service for the first time, can be used at any other moment, in which case it only passes the Intent parameter to the onStartCommand() of the service, which processes it.

One more important concept related to services and broadcast receivers is the Wake Lock. Wake Locks are a feature of the Android operating system that allows developers to control when the device can go to sleep. For example, if an application does not have any active wake locks, after a certain amount of time the screen will turn off and, eventually, the processor will also go to sleep, stopping any processing the application was doing. By acquiring a lock, the application can tell the OS that it needs some power settings (e.g. do not turn off screen or do not stop CPU processing). More details regarding the WakeLocks can be read in Android's user manuals [25].

The concept of Wake Locks is important in FenceIt because our application should run in the background, even when the screen is off. Thus, at some points, the application should notify the OS to not stop the CPU, because there is some processing it needs to do. And this is done by requesting a PARTIAL_WAKE_LOCK. Broadcast Receivers can receive Intents even when the device is asleep. In this case, the OS gives the receivers a temporary PARTIAL_WAKE_LOCK, but only for the duration of the processing done in the *onReceive*() method of the BroadcastReceivers. Because the application needs to also run after that method is finished (i.e. in the Background Service and in the Checker Threads), we implemented a WakeLock Manager, which has the role of requesting a WakeLock from the Android OS, when the *onReceive*() method is run and release it when all the Checker Threads have finished their processing.

6.7.2 The main module

This module is mainly represented by the class *BackgroundService (com.fenceit.service)* which extends Android *Service* class. It is the actual Service started either by the application or by any Broadcast Receiver with the help of the following command:
Context.startService();

As any service, after the previous command is issued, if the Service was not already running, the *onCreate()* method is run, which handles the initializations of the service and schedules an immediate check for all Location Types.

If the service was already started, the *startService()* method was probably started from a Broadcast Receiver that notifies the service that a scan for triggering conditions of a given type should be done. This type, if any, is taken from the extras of the Intent received and the check for that particular Location Type is started. There are a couple of values, defined in the BackgroundService class, that can be used and they are presented in the following table:

Table 3 - Events in Background Service

SERVICE_EVENT_WIFIS_DETECTED	used for defining the event related to starting a check for the type: WifisDetectedLocation

SERVICE_EVENT_WIFIS_CELL_NETWORK	used for defining the event related to starting a check for the type: CellNetworkLocation
SERVICE_EVENT_WIFI_CONNECTED	used for defining the event related to starting a check for the type: WifiConnectedLocation
SERVICE_EVENT_GEO_COORDINATES	used for defining the event related to starting a check for the type: GeoCoordinatesLocation
SERVICE_EVENT_RESET_ALARMS	used for starting a check for all location types, as a change in the alarms has been made and previous computed check times might not be reliable anymore.

After the event has been identified, the service has to start an analysis of the triggers containing Locations of that particular type. As this process can be time consuming and the methods of the Service are run on the main thread, a new thread that makes the scan is started. This thread is called a TriggerCheckerThread and there is a full implementation for every Location Type. To add flexibility and accommodate any future addition of new types, a TriggerCheckerBroker was implemented, which has the purpose of making the correspondence between the event (which type should be checked) and the Trigger Checker Thread that should be started.

As, during their run, the checkers might find an Alarm that should be triggered and the corresponding actions should be executed on the main thread (for example it might be needed to interact with the UI), the threads need a method to communicate with the Background Service and run code on the main application thread. This is done with the help of the Handlers and Messages from the Android OS. When it is created, the Background Service creates a handler to itself that allows any other thread to send a message which is then processed by the main thread. When a Checker Thread is started, the handler is given to it so, if any time a communication should be done between the main thread and the checker thread, the handler can be used. Besides this, the handler is used to communicate with the main thread when the next check of this type should be scheduled, as the interaction with the Alarm Manager in Android cannot be done from other threads.

6.7.3 Trigger Checker Threads

The main purpose of Trigger Checker Threads is to check the current contextual (environmental) conditions) to see if any of the Triggers should be set off. They are implemented as Threads, so they can be started by the BackgroundService and can run the check without interfering with the main thread of the application.

In order to ease the implementation, an abstract checker thread was implemented, *TriggerCheckerThread (com.fenceit.service.checkers)*, with the main processing already build in. For each location type, a class has to be written that extends the *TriggerCheckerThread* and fully implements the methods that are particular to each kind of Location Type. The checking algorithm used by the Trigger Checker Thread is shown in Figure 18.

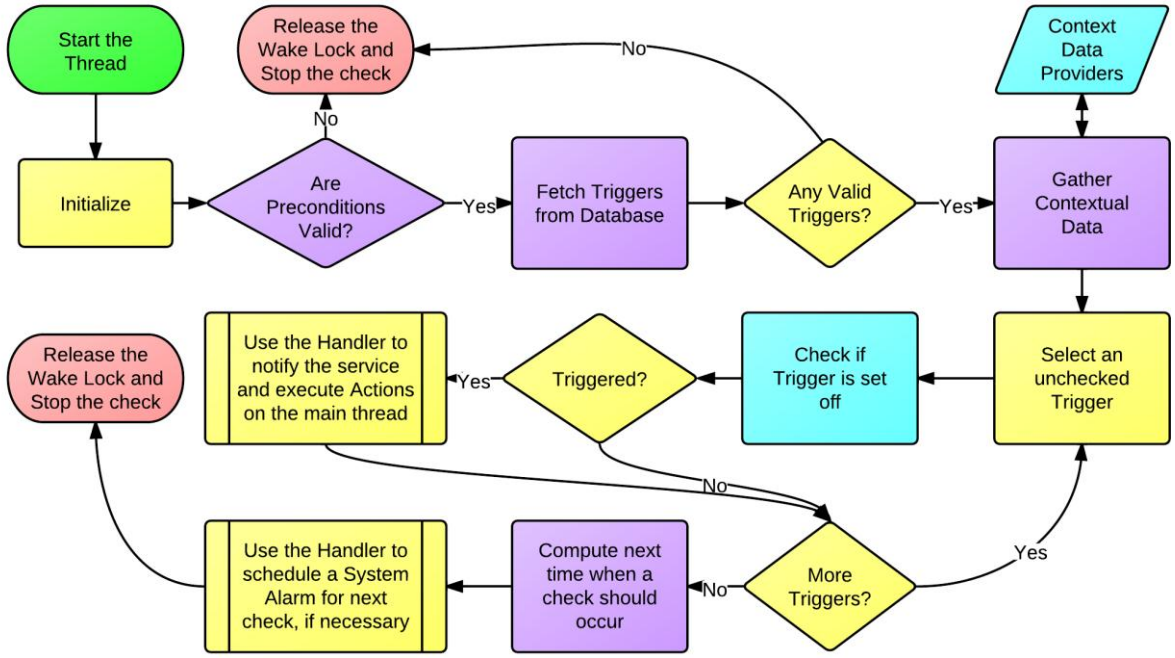

Figure 18 - Process diagram for the Trigger Checker Threads

In the diagram all the steps of the checking algorithm are presented. With yellow, the figure shows the steps that are implemented in the abstract *TriggerCheckerThread* class, steps which are common to any check, independent of the Location Type. With violet, we can see the steps that are implemented by the particular checker threads and are dependent on the Location Type. In teal, we can also see the components with which the checker threads interact, but are implemented outside the service. These are the Context Data Providers and the check if the Trigger should be set off, done in the *BasicTrigger* class.

An important aspect in the algorithm of the Trigger Checker Threads is the scheduling of the next scan. As it was stated before, by using a different checker for each Location type, we have the flexibility of making checks for various triggers at different time intervals. The time between checks is dynamic and is computed based on the optimization considerations and algorithms presented in Chapter 3. Each implemented Trigger Checker uses as much available information to estimate what is the optimal time for the next scan for a particular Location type.

Chapter 7: Testing and Evaluation

In order to test and evaluate the developed application, we have used three Android powered devices, their specification being the following:

- Samsung Galaxy Gio S5660, 320 x 480 pixels/ 3.2 inches display, 278 MB RAM, 800Mhz CPU
- Samsung Galaxy Tab P1000, 600 x 1024 pixels/ 7.0 inches display, 512 MB RAM, 1000Mhz CPU
- HTC ChaCha, 480 x 320 pixels/ 2.6 inches display, 512 MB RAM, 800Mhz CPU

7.1 Testing

The tests performed covered both unit testing, where individual components of the application were tested as separate units, acceptance and functional tests, that verify if the application meets the requirements set in Chapter 4 and follows all the use case scenarios.

Unit testing was performed on the Database Component, every section of the Andro-Wrapee library having associated unit tests and being verified for proper functioning. The resulted behavior was as expected, all features of the library functioning correctly.

In the case of the acceptance and functional tests, we broke the testing process into two parts. First we tested the application interface and its proper behavior and then we tested the background service and if the alarms are triggered as specified.

Testing the application interface, we took each scenario into consideration and navigated through the application screens, analyzing the possible outcomes of each command the user might issue. On the Location Definition screens, we tested the proper interaction with the Context Data Providers and that the contextual information that appeared on screen was accurate. Also we tested whether the user interface allows the users to create alarms, triggers, actions or locations without a proper structure or with incomplete information, introducing inconsistencies in the database. The results for all the tests were highly satisfactory, all the application interface screens behaving as expected.

While testing the background service and the proper behavior for triggers configured with all the four location types, the approach was to both do an emulator test, where we provided values from the Android emulator, trying to trigger alarms, and also a real-life test, where using all three mobile devices we set up various locations in different places (over an urban area wide of around 5 kilometers) and checked if they would trigger as expected. From the implementation point of view, the results were as expected and the application managed to trigger the alarms when supposed to.

7.2 Evaluation

The process of evaluating FenceIt was divided in two and was composed of the analysis of the performance of each of the four localization methods (which is better in which situation) and the evaluation of the power consumption.

7.2.1 Performance Evaluation

In order to properly evaluate which of the four localization methods (based on geographical coordinates, connected Wi-Fi network, Wi-Fi networks in range or Cellular Network), we set up

various Locations of different types in multiple places, over an urban area of around 5 kilometers wide (in and around the university campus). The test was done with the first two devices presented at the beginning of the chapter and the results were similar.

The following table makes a summary of the findings:

Table 4 - Location types evaluation

Location Type	Evaluation
GeoCoordinatesLocation	The accuracy of the alarms triggering was very good (under 100m) and the facility of being able to define the Location from a distance proved very useful. The biggest issues found were related to the loss of signal for the GPS hardware inside the buildings and, sometimes, the very long time it takes to get an accurate fix on the position. Localization without the GPS interface was also very satisfactory, but the accuracy dropped significantly (to around 200-300m).
WifiConnectedLocation	For this test, the device was set to auto-connect to the Wi-Fi networks used in the Locations. The results were extremely good in areas where we had access to a Wi-Fi network, the application triggering the Alarms extremely accurate. However, the fact that certain areas lack Wi-Fi networks (or the networks are inaccessible) brings an important downside to this method and makes it applicable only in some situations.
WifisDetectedLocation	The results for using the detected Wi-Fi networks were also very good, the accuracy of the alarm triggering being excellent. As before, we were not able to define any alarms in areas without Wi-Fi networks (e.g. parks). However, by not having the need to connect to the networks, this method was applicable in some cases the WifiConnectedLocation type was not.
CellNetworkLocation	This method proved to be the least accurate, as in some situations the mobile device was being connected to the same Cell Site on very large areas or, in other cases, the Cell Site the device was connected to switched a few times on a radius of 10-15 meters. The biggest advantage was the fact that this method can be used practically everywhere, provided that the triggering accuracy does not have to be very good.

We can conclude that each method is applicable in certain situations and the way they are used depends on the cases and on the user's needs.

7.2.2 Battery Consumption

For evaluation purposes, from the point of view of battery consumption, we analyzed two aspects of the application: how each method compares with others and what advantages the optimization algorithms bring, if computing a variable checking frequency compared to using a fixed interval.

When testing, to have an adequate testing base, we used 8 alarms, each with 2 triggers of the same type, in total having 4 triggers corresponding to each Location type. When needed to evaluate only a Location type, all alarms containing other types were disabled, but still present in the database.

In order to analyze the power usage, we used a third party application available in Android: Power Tutor 1.4. It allows users to profile power consumption and evaluate how much battery is drained by running application.

To start with evaluating consumption for the four Location types, the results were as expected, with the Geographical Coordinates localization being most power consuming, while Cell Network localization being the most battery friendly. A small description of the results found is:

- GeoCoordinatesLocation - Using the GPS hardware has a huge effect on the power used and should be used only when strictly necessary. Using the Coarse method only, the effect is more moderate, but still above all other three methods.
- WifisDetectedLocation - This is the second most power-consuming, as it requests a Wi-Fi scan from the Wireless interface.
- WifiConnectedLocation - With the device only gathering information about the Wi-Fi network to which it is registered, the power consumption is low compared to the others.
- CellNetworkLocation - As the device is always registered to the Cellular Network, the necessary data is already prepared, thus making this type of Location the most power efficient.

The battery consumed in each of the cases can be seen in the screenshots in Figure 19 (by Location Type, from left to right: GeoCoordinatesLocation, WifisDetectedLocation, WifiConnectedLocation, CellNetworkLocation), where the average power used by the app in the previous 5 minutes is shown. In these 5 minutes, the application had only a Location type active and the optimization algorithm was not enabled, all the checks being made at the same fixed frequency.

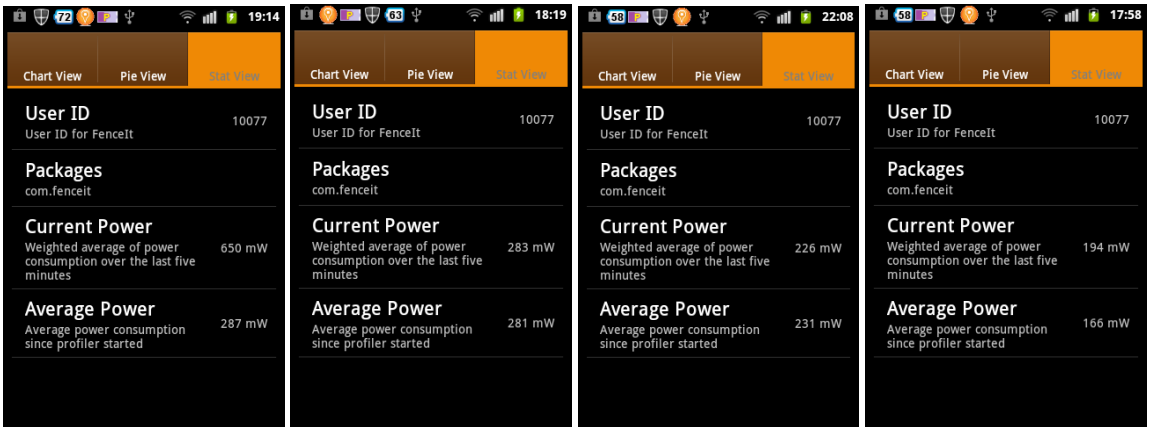

Figure 19 - Battery consumption by Location Type

When analyzing the optimization algorithm that computes a variable time interval between updates, the power consumption differences could not be evaluated extremely accurately, as the time frame of a couple of minutes is relatively short compared to a full-day use of the application. Nonetheless, a difference between the application version with the optimization enabled and the one without it was noticed (around 50mW for the 5 minutes interval). In Figure 20, we can see the spikes when the battery was used in the case where the algorithm was enabled (left), and where it was not enabled (right). On the X axis we can notice the time passed (in seconds), and on the Y axis we can see the power used (in mW).

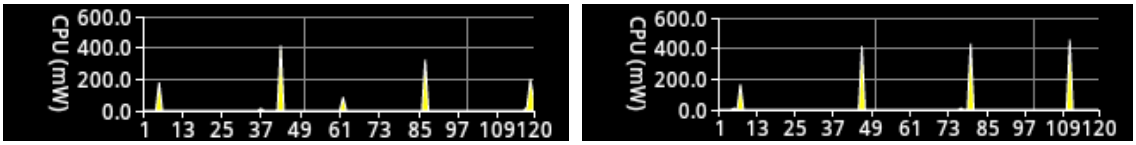

Figure 20 - Battery consumption without optimization algorithm (left) and with (right)

As we could see from the evaluation of the power consumption, we can order the location types by power used (from most efficient to least): CellNetworkLocation, WifiConnectedLocation, WifisDetectedLocation, GeoCoordinatesLocation. And we can also conclude that using the adaptive algorithm is efficient on the long term.

Chapter 8: Future Work

The research in the field of mobile device localization is still in its incipient stages and there are still solutions insufficiently explored. Regarding localization methods, future research can explore other mechanisms and can use other hardware interfaces present on mobile devices.

Regarding FenceIt, the complexity of the application makes it a good candidate for improvement and for adding new features. Some of them can be:

- Take other contextual information into consideration, for example signal strength in the case of Wi-Fi networks or the Cell Sites. Other examples could include using all the Cell Sites in range of the device to pin-point the devices location; however this is highly dependent on the hardware device.
- Implement other Location types. The architecture of the application was designed so that adding a new localization method can be done extremely easy and without a major impact on the rest of the components.
- Implement other Action Types. As before, the architecture of FenceIt makes adding new Actions a simple process. Examples can include: sending Short Text Messages (SMS) when an alarm is triggered, changing the active sound profile (switch Silent on or off) or changing the state of interfaces (Wi-Fi, Bluetooth, and GPS).
- Add a web component to the application that allows users to easily share Locations among each other or allow clients to be notified when a friend reaches a particular location. This can be implemented with the help of a web server (e.g. Google AppEngine) that is used for synchronization of Alarms, Locations or other necessary data.

Chapter 9: Conclusions

This paper discussed the main considerations that have to be taken into account when trying to locate a mobile device in a particular area. We have shown that using only the geographical position of the device is not always the most appropriate solution for an efficient mobile location-based application and that there are more methods which can be used to place a device in an area, namely using the Wi-Fi Network to which the device is connected, the Wi-Fi Networks in range or the information provided by the cellular network to which the mobile phone is registered.

Using any of the four methods has advantages and disadvantages and each one should be used when it is most appropriate. We have also shown that other contextual information, such as distance to target or inactivity period of the device, can be used by the algorithms to improve the battery consumption and data connection usage.

The paper also described the implementation of FenceIt, a mobile application for the Android operating system that allows the users to define alarms that get triggered when the device leaves or enters some areas (Locations). Besides this, the clients can define some actions which are automatically executed when any of the alarms is triggered. The application implements all the four previously mentioned methods for localizing the device and the optimization algorithms.

The application is designed especially with modularity, extensibility and efficiency in mind and is broken into four modules, which interact with each other. The Application User Interface is the component the user is mostly aware of, allowing him/her to configure the existing alarms, locations and actions, and to create new ones, as necessary. The development of the Database Component, which handles all the interaction of the other modules with the database, resulted in the creation of an open source library for Android that can be used by other developers. The Context Data Providers Component incorporates modules which interact with the environment, gathering required contextual information, with the help of the Wireless Interface, Cellular Network interface and positioning hardware. The Background Service is the component that is constantly running, processing the existing alarms and scanning for triggering conditions.

FenceIt was developed using efficient algorithms and has evolved into an application that can be used not only for studying the previously mentioned localization methods, but also by non-technical users in their daily lives.

For mobile devices, battery consumption of an application, especially one that is running permanently, is of utmost importance. This has determined us to put the most of our work in researching efficient localization methods and the results are highly satisfactory as long-term usage of FenceIt does not considerably affect the battery levels. Algorithms that take different factors into consideration when computing the frequency of computation performed in the background of the application have decreased the battery consumption to an even lower level.

To conclude, after the evaluation of the application, we confirmed that, even though there are a lot of choices when it comes to localizing a mobile device in a particular area, no method is optimal for all the usage scenarios and each of them is more appropriate for a particular situation. Furthermore, advanced algorithms can be used to optimize the process and take into consideration different information gathered from the environment. Regarding FenceIt, we consider the application to be one with a very high potential and one that can be easily used by people in their daily lives to improve their tasks.

BIBLIOGRAPHY

1. **Dey, A. and Abowd, G.** *CybreMinder: A Context-Aware System for Supporting Reminders.* 2000.
2. **Marmasse, N. and Schmandt, C.** *Location-Aware Information Delivery with ComMotion.* 2003.
3. **Barbeau, S. J. et al.** *Dynamic Management of Real-Time Location Data on GPS-enabled Mobile Phones.* 2008.
4. **Gasinski, D.** *Location-Based Services inom undervisningen på Broängsskolan.* 2009.
5. **La Marca, A. et al.** *Place lab: Device positioning using radio beacons in the wild.* 2005.
6. **Sohn, T. et al.** *Place-Its: A Study of Location-Based Reminders on Mobile Phones.* 2005.
7. **Ludford, J. et al.** *Because I carry my cell phone anyway: functional location-based reminder applications.* 2006.
8. **Meeuwissen, E. et al.** *Inferring and Predicting Context of Mobile Users.* 2007.
9. **Hedin, B. and Noren, J.** *Mobile Location-Based Learning Reminders using GSM Cell Identification.* 2009.
10. **Inc, Apple.** iOS5, Location Reminders. [Online] 2012. http://www.apple.com/ios/features.html.
11. **Incorporate Apps.** Spoty Location Reminder. [Online] 2012. http://www.incorporateapps.com/.
12. **Corporation, ePythia.** ePythia To Do List. [Online] 2012. http://www.epythia.com.
13. **Pearson Education Limited.** *Longman Dictionary of Contemporary English.* 2003.
14. **Millete, G. and Stroud, A.** *Professional Android Sensor Programming.* 2010.
15. **Society, IEEE Computer.** IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. 2007.
16. **Melville, A.** *IEEE P802.11 Wireless LANs. Virtual Access Point Definition.* 2004.
17. **Flood, J. E.** *Telecommunication Networks. Institution of Electrical Engineers.* London : s.n., 1997.
18. **Sohn, T. et al.** *Mobility Detection Using Everyday GSM Traces.* 2006.
19. *A study on present and future of Google's Android.* **Dot Com Infoway.** 2012, Position Paper.
20. **Komatineni, S. and MacLean, D.** *Pro Android 4.* 2011.
21. **Apache Software Foundation .** *Apache log4j™ 1.2.* [Online] 2012. http://logging.apache.org/log4j/1.2/index.html.
22. **Open Source Project.** *Android-logging-4j.* [Online] 2012. http://code.google.com/p/android-logging-log4j/.
23. **Stefan-Dobrin, Cosmin.** *Andro-Wrapee.* [Online] 2012. http://code.google.com/p/andro-wrapee/.
24. **Google Corporation.** *Android UI Guidelines.* [Online] 2012. http://developer.android.com/guide/practices/ui_guidelines/index.html.
25. —. *Android Developer References.* [Online] 2012. http://developer.android.com/reference/packages.html.