netatmo

MASTER THESIS REPORT

Presented by

Santiago Ramirez Aretio

For the Master of Science *Computer Science for Communication Networks (CCN)*

Internship did from 1 March to 29 August at

Netatmo

Android Native implementation of Netatmo Weathermap application.

Director of the internship: **Sergey Vakulenko**

Academic Supervisor: **Mr Paul Gibson**

Table of content

1 Introduction

The aim of this section is to introduce Netatmo, the company where this project has been carried out. This project is related to a particular product called Netatmo Weather Station. For understanding the context of the project, this particular product is further explained. Finally the project is introduced.

1.1 About Netatmo

Netatmo is a revolutionary smart home company, developing groundbreaking, intuitive and beautifully-designed connected consumer electronics. Truly smart, Netatmo's innovative products provide a seamless experience that helps users create a safer, healthier and more comfortable home. Netatmo carefully designs the mechanics, electronics and embedded software of all its products to the highest standards. Netatmo also creates the mobile and web applications that fully realise their potential. Since 2012 Netatmo has released four devices, all of them infused with intelligence and delivering state-of-the-art features:

- The Netatmo Weather Station for Smartphone allows users to keep track of what is happening in their indoor and outdoor environments in more than 170 countries. It is the world's largest collaborative weather station network.

- The Netatmo Thermostat for Smartphone, designed by Philippe Starck, allows users to control their heating remotely from a smartphone. With analysis of their daily routine, the Thermostat for Smartphone allows users to save 37% on energy heat their home.

- Welcome, the indoor security camera with face recognition technology, puts names to the faces it sees. The camera notifies the user exactly who's at home, their loved ones or a stranger.

- Presence, Netatmo's outdoor security camera, detects and reports on people, cars and animals. The camera understands what it sees and lets the user know exactly what is happening outside his home.

Netatmo is a key player in the smart home industry, with products available through various distribution networks worldwide, from both major retailers and BtoB channels. In November 2015 Netatmo completed a series B funding round of €30 million. The company previously raised €4.5 million in 2013.

1.2 About Netatmo Weather Station

The Netatmo Weather Station is a personal weather station with air quality measurements that that can be consulted via an iOS or Android application called Netatmo Weather Station. This product contains a set of sensors to monitor living environment (temperature, barometric pressure, humidity, CO2 concentration, noise pollution...).

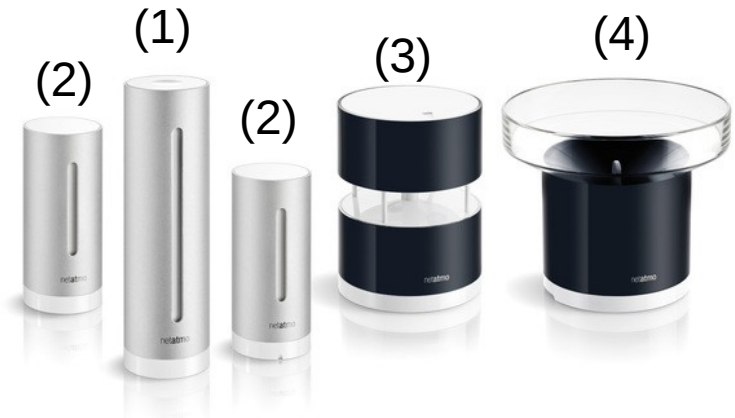


Fig 1. Weather Station Modules

Weather Stations owners provide Wi-Fi connection to their stations and measurements are transmitted wirelessly to a Netatmo private server. Measurements are indefinitely stored in a database where they are always available. Weather Station owners can access their measurements using a mobile application that interacts with Netatmo private servers. Figure 1 shows all Netatmo Weather Station modules:

(1) Indoor Module: Measures temperature, humidity, CO2, pressure and sound intensity.

(2) Outdoor Module: Measures temperature, humidity and pressure.

(3) Wind Gauge Module: Measures wind speed.

(4) Rain Gauge Module: Measures amount of rain which falls.

All modules communicate with Indoor module using a RF proprietary protocol. Indoor module send all measurements to Netatmo server using its Wi-Fi interface.

Netatmo Weather Station users can make their station public. The location and measures of a public station can be accessed by everybody using Netatmo public API. For privacy reasons indoor measures are not shared by public stations.

1.3 Android Native Weathermap

Netatmo Weathermap is a web application where public stations are displayed on an interactive map. Users can view measurements from stations along the world.

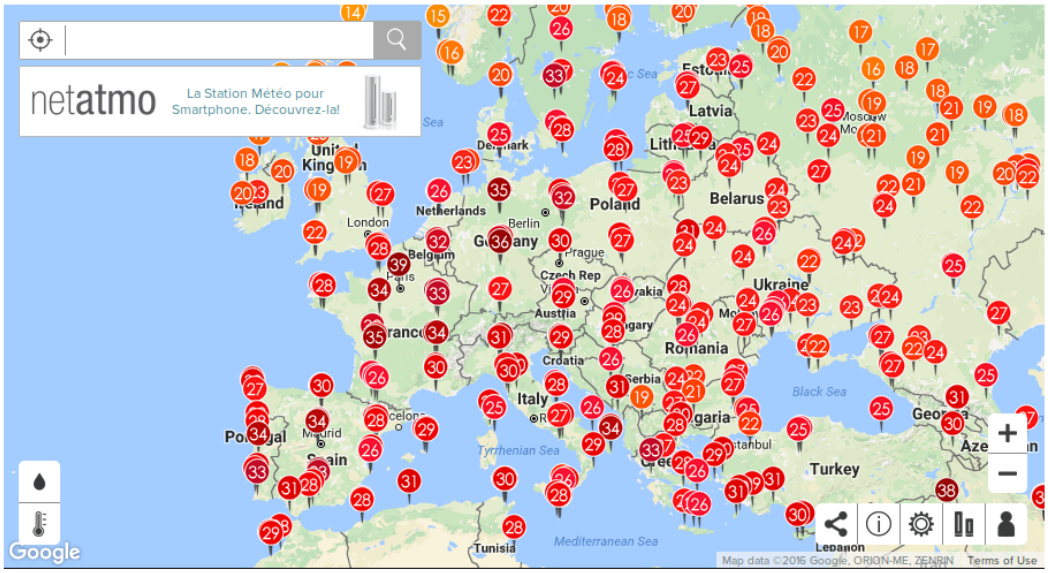


Figure 2: Weathermap web application screenshot

Netatmo Weather Station includes the Weathermap web application as an independent module using a WebView. A WebView can be seen as a web browser that is attached to the screen in an Android application. This solution introduces an extra software layer that limits the Weathermap performance on Android devices. This implementation also does not allow the rest of the application to interact with the Weathermap.

The main goal of this project is to implement and integrate an Android native version of the Weathermap using Android SDK and GoogleMaps API. A native implementation involves the possibility of a deeper integration of the Weathermap and a better performance.

Here is the list of tasks carried out during the project:

-Implement an Android native Weathermap using Android SDK and GoogleMaps API.

-Integrate of the Weathermap into the existing Weather Station application.

-Add a new feature to the existing application. Users can set public stations as favorites and access to their data without going through the Weathermap.

-Integrate Netflux system in the existing application.

-Test and verify the implemented application.

2 State of the art

In this section are explained the most innovative tools and techniques that have been used for developing this project. Some design patterns and libraries tools that are part of the state of the art in Android developing are described in this section.

2.1 Presenter-Interactor design pattern

A very generic scenario in all mobile applications is to have some data that the user can visualize and modify. The main objective of this architecture is to make all visual elements (Views) independent of their data source. He can achieve this separation by removing all logic from the views and make then behave like a passive interface that displays data and routes user events.

Three different layers or domains may be defined in order to achieve this separation. In Figure 3 there is a schema where we can see these tree layers called Presentation layer , Domain Layer and Data Layer. Each layer has a different mission and works separately from the others.

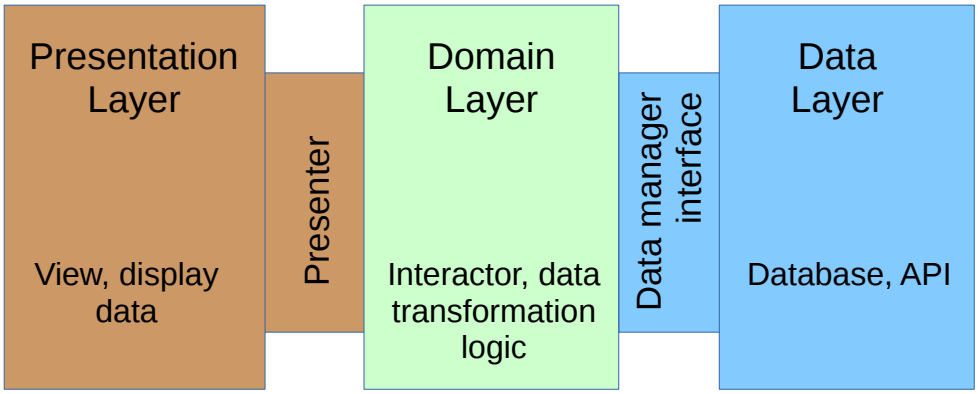


Figure 3: Layer separation

In this document, all schemes and graphs has been designed to follow the color code shown in Figure 3. Red color for all modules and components related to presentation layer. Green for all modules and components related for domain layer and finally blue color for data layer.

Presentation Layer

All views and logic related to visual resources, animations and user interaction are included in this layer. In this architecture each view will contain its own presenter and each presenter will contain its own view. The presenter is a component in charge of provide data to its view and communicate user interactions detected by its view to the application logic. The data provided by the presenter must be ready to display in order to keep all data processing logic out of this layer.

Domain Layer

All business logic of the application is included in this layer. In terms of android project structure, this layer is composed by pure Java modules without any android dependencies. Each presenter contains an interactor in this layer. The interactor is the component in charge of recollect and provide data to its present on demand. Due to the fact that there is a one to one relationship between views and presenters and between presenters and interactors, we can say that it exists a one to one relationship between views and interactors. The presenter acts as an interface between the view and the interactor.

Data Layer

All raw data needed for the application belong to this layer. As the data may be stored on the device or in a remote database accessed by an API, this layer also contains all cache implementations and networking tasks. The main purpose of this layer is to make the data origin transparent for the interactors.

In this project, presenters and interactors has been designed according to the following interfaces:

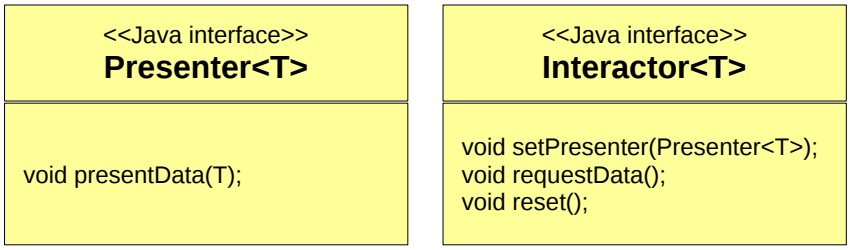


Figure 4. Presenter and Interactor interfaces

Generic types has been used to describe the interface in order to implement each pair of presenter-interactor with a different data type according to the visual information needed in each view. The presenter-interactor interaction during the time is described in the following schema:

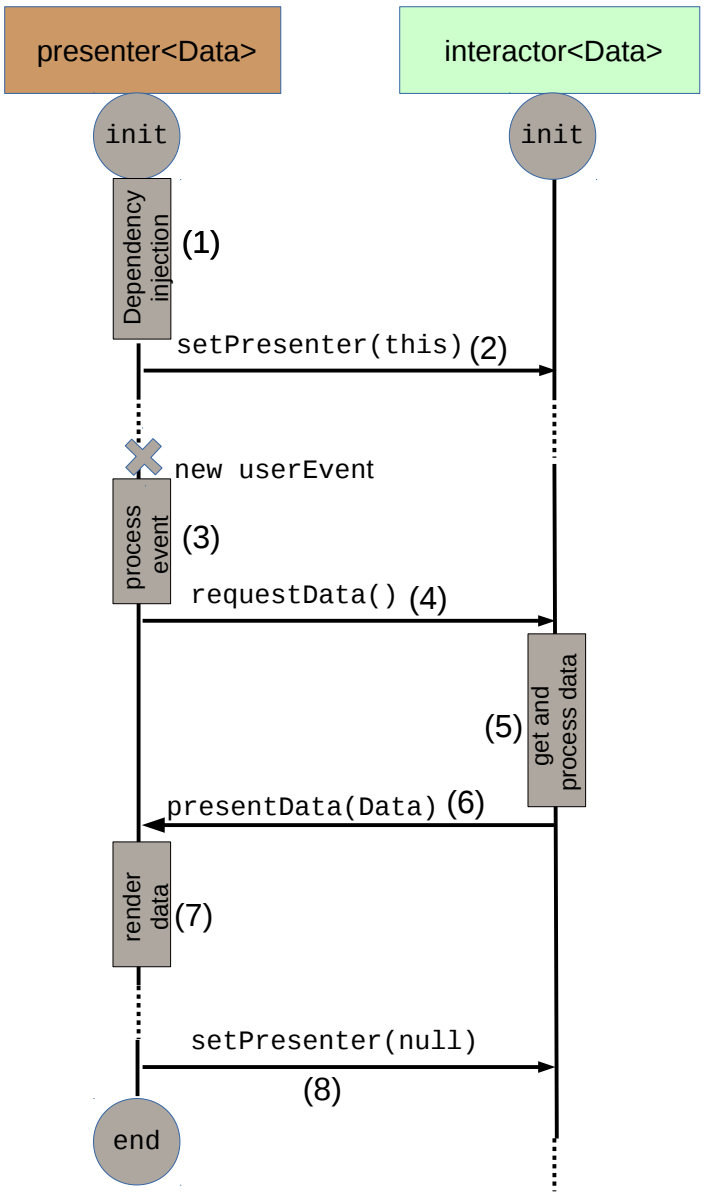


Figure 5: Interaction in time between presenter and interactor

(1) When the presenter is created and initialized, its interactor dependency is injected

(2) After injection the presenters registers itself to its interactor calling *setPresenter* method.

(3) When the view associated to the presenter receives a user event, the presenter does some processing to discover if new data is needed according to the event information.

(4) If new data is needed, the presenter calls *requestData* method. Its important to notice that this call is not blocking. After calling *requestData*, the presenter is free to process new user events.

(5) As soon as *requestData* is called, the interactor starts to execute all logic necessary to get the requested data from the data layer. Once the data is recovered, it process the data to transform it in in ready to display data.

(6) When the data is ready, it is delivered to the presenter by the interactor using *presentData* method.

(7) The presenter renders the received data to all visual elements of its view.

(8) This process is repeated meanwhile the vie is attached to the screen. When the view is about to be destroyed, the presenter is unregistered from its interactor by calling *setPresenter* method with null parameters.

Thanks to this architecture an application may be easier maintainable and readable. If we do this layer separation:

- Its easier to fix problems if we can distinguish between errors in the presentation layer and the data logic layer.

- Due to the fact that each view is independent of the others, it's possible to test all possible visual cases of our application by testing each view individually with fake data.

- If the data provider of our application changes (a change in a web service or in the database structure) its not necessary to change any view.

2.2 Dagger dependency injector

Dagger is an open source library for Java and Android that implements a dependency injection framework. Although it is not as powerful as other dependency injectors, Dagger is widely used noways for Android development due to its good performance in low-end devices.

This section introduces dependency injection problem in order to expose the motivation and the benefits of using Dagger. It is not the aim of this section making a tutorial about how to use this library.

2.2.1 Motivation

A dependency can be described as a coupling between two modules of a software project. In object oriented languages there is a dependency between two classes if one of them uses the other to do something.

The **single responsibility principle** states that every module or class should have responsibility over a single part of the functionality provided by the global software system, and that responsibility should be entirely provided by the module or class. This term is based on the **principle of cohesion** described by Tom DeMarco[1] and Meilir Page-Jones[2] and it was introduced by Robert C Martin[3] in 2003. Nowadays this principle is highly respected by the state of the art of object oriented software.

In order to create testable projects the single responsibility principle must be respected. Unit testing requires that, when testing a module, it must be isolated from the rest of modules. If the module under test has dependencies, we have to substitute with mocks. However, depending on how dependencies are implemented, we can not achieve this substitution. For instance imagine that we want to test *ModuleA* in the following code:

```
public class ModuleA{
	private ModuleB moduleB;

	public ModuleA(){
		moduleB = new ModuleB();
	}

	public void doSomething(){
		moduleB.doSomethingElse();
	}
}
```

It's not possible to test *doSomething* method without testing *doSomethingElse* method. We would not be able to know what method fails if the test fails. Every dependency class initialization is a hard dependency that we need to avoid according to the single responsibility principle. In order to make the last code testable we need to introduce some changes:

```
public class ModuleA{
	private ModuleB moduleB;

	public ModuleA(ModuleB moduleB){
		this.moduleB = moduleB();
	}

	public void doSomething(){
		moduleB.doSomethingElse();
	}
}
```

Now the dependency is created via constructor. *ModuleA* can be individually tested by providing a mocked *ModuleB* in the initialization.

This strategy is a particular case of the **dependency inversion principle** that will be further explained in the next section. The problem now is where instantiate the dependencies that will be passed as a constructor attribute when creating a new object. We can find other problem if there are classes with a high number of dependencies. In that case the number of attributes added to the constructor would be too hight to have a readable code.

Dependency injectors have been created as an answer to this problems. Dependency injectors are modules in charge of providing dependency instances to the rest of modules. Using them it's possible to localize the creation of modules in a single point in a software project.

2.2.2 Dependency injection

Dependency injection is a software design pattern that implements inversion of control[4] for resolving dependencies. A dependency is an object that is used by another.

An injection consist in providing a dependency to a dependent object. In this context the dependent object can be described as a client that will use the dependency object as a service. As a consequence, the client object does not need to find the service, create it or initialize it.

Dependency injection involves four roles:

- **Service** (an object to be used by others)
- **Client** (an object that uses other object)
- **Interface** (the way that a service is published for being used by clients)
- **Injector** (in charge of providing services to clients)

The **interfaces** are the types the client expects its dependencies to be. Interfaces allow clients to know how to use the injected service. Given a dependency, if the code of the service changes but its interface remains constant, I wont be necessary to recompile the client.

The **injector** introduces the services into the client. In a complex system where a dependent objects may be a dependency for other objects, injector is also in charge of creating and solving the dependency graph in order to instantiate and inject objects in the correct order.

The client is not allowed to call the injector code. As a consequence the injection mechanism used is transparent for the client. The client only needs to know how to use a service, not how to construct it.

2.2.3 About Dagger

Most dependency injectors frameworks such as Spring rely on reflection to create and inject dependencies. This type of injectors have the same qualities as the reflection, they are very powerful and versatile but also very time consuming.

Dagger, on the contrary, uses a pre-compiler that creates all the necessary classes to create and inject dependencies. It implements the full stack injector with generated code. The guiding principle is to generate code that mimics the code that a programmer would write to initialize dependences. As no reflection is needed, Dagger is less time and resources consuming and, as a consequence, is optimal for low-level devices such as mobile devices.

2.3 Netflux

A very common scenario in mobile applications can be described as a data model that is accessed by multiple visual elements in order to display information to the user. This model usually changes because of user interactions or by automatic actualizations. For implementing a robust application, when there is a change in a part of the model, it is necessary to notify the new model information to all elements that consumes that part of the model at the same time. If not, its possible to have a dangerous state where two application components have different versions of the same data source.

Netflux system has been developed by Netatmo in order to solve this problem. Netflux system has two main functionalities:

-Provide an interface to the rest of the application for modifying the data model.

-Notify each change in the model to all elements that consumes that part of the model in order to be sure that all elements see the same version of the data model.

Netflux models application data as a tree graph. This model matches very well in object oriented programing environments where data objects contain other data objects as attributes. In the following figure we can observe a data model example where a student is modeled as:

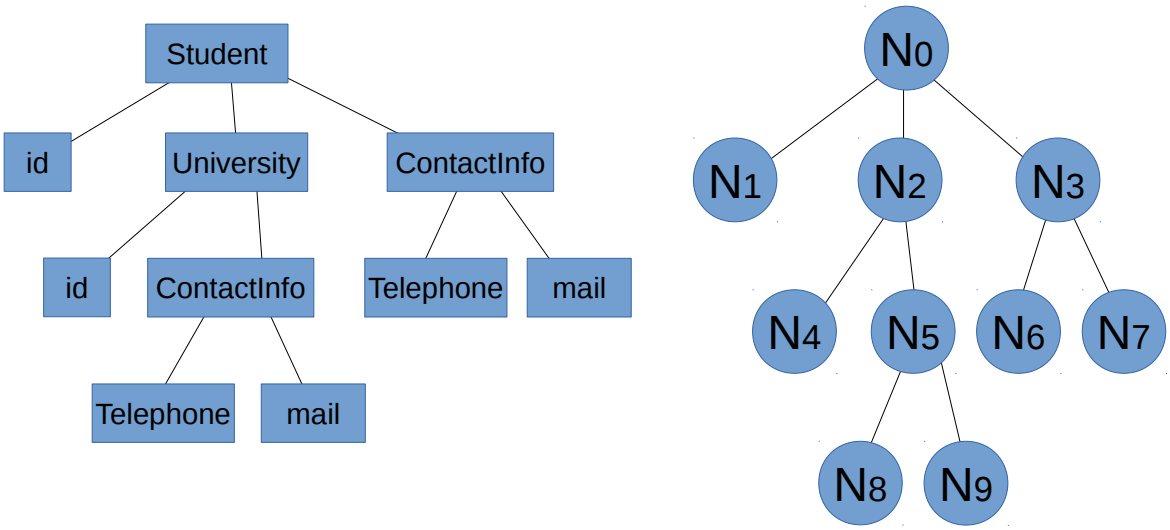


Figure 6: Data model tree graph example

Classes dependency can be translated into a tree graph. Each node in the graph N_i contains a part of the entire model $M(N_i)=M_i$. The root node N_0 contains the entire model. In the example in Figure 6 M_3 would be the student contact info.

Each node acts as a notifier. Components in the application can be subscribed to some nodes if they interested in the information contained in them. When a change in some node occurs, that node is in charge of notify the change to all its subscribers. Let $L(N_i)$ denote the set of listeners subscribed to node N_i.

Each node contains a reference to its parent. Let $P(N_i)$ denote the parent node of node N_i. This reference is null for the root node. For instance, in the example in Figure 6 we have $P(N_7)=N_3$. In the same manner, each node contains a reference to all of its children nodes. Let $C(N_i)$ denote the set of children nodes of node *Ni*. This reference is null for the end nodes. For instance, in the example in Figure 6 we have $C(N_3)=\{N_7, N_6\}$.

Each node implements a function called *reduce* that receives as parameter some data in its parent data domain and extracts all information that belongs to its data domain. For instance, in the example in Figure 6 function $N_3.reduce(\hat{M}_0)$ would return the contact info contained in a instance of student data model $\hat{M}_i$.

Each node implements a function called *map* for each of its children nodes. This function receives as parameters some data in the node domain and some data in the child data domain and returns a new version of the first parameter where information contained in the second parameter is included. For instance, in the example in Figure 6 function $N_3.map(\hat{M}_3, \hat{M}_6)$ would return the contact info contained in $\hat{M}_3$ updated with a new mail address contained in $\hat{M}_6$.

The rest of the application can modify the data model by calling *changeModel* function implemented in each node. The change to be performed is represented as an instance of the node data domain and it is passed as a parameter. This function is also in charge of propagate the change throughout the graph. The propagation chain finish when it reaches the root node. In that moment the entire model has rebuilt according with the the change that has been introduced.

$N_i.changeModel(M_i^{new})$

```
if M_i ≠ M_i^new
    if ∃P(N_i)
        M_parent^new ← P(N_i).map( M(P(N_i)) , M_i^new )   (create parent new data model)
        P(N_i).changeModel(M_parent^new)                   (propagate model change)
    else
        N_i.notify(M_i^new)   (start notification process if root node)
    end
end
```

Once the entire model has been modified, its necessary to notify the change to all components in the application that are listening to the part of the model that has been modified. For achieving this each node implements a function called *notify*. This function creates a propagation chain that starts from the root node and reaches all nodes whose model has been changed. During this propagation each node also stores, if necessary, the new model.

$N_i.notify\left(M_i^{new}\right)$

```
if  M_i ≠ M_i^new
    M_i ← M_i^new      (store new model)
    ∀ l ∈ L(N_i):
        l.update(M_i^new)      (notify subscribers)
    end
    ∀ N_k ∈ C(N_i):
        M_k^new ← N_k.reduce(M_i^new)      (create child new data model)
        l.update(M_i^new)                  (propagate notification)
    end
end
```

For instance, we can imagine that in the scenario described in Figure 6 a component wants to update university contact info. However this update involves a change in the mail information but not in the telephone information. In this case the component would call $N_5.changeModel\left(M_5^{new}\right)$ where parameter M_5^{new} contains the desired new contact info. After the call a propagation chain would be started as described in Figure 7.

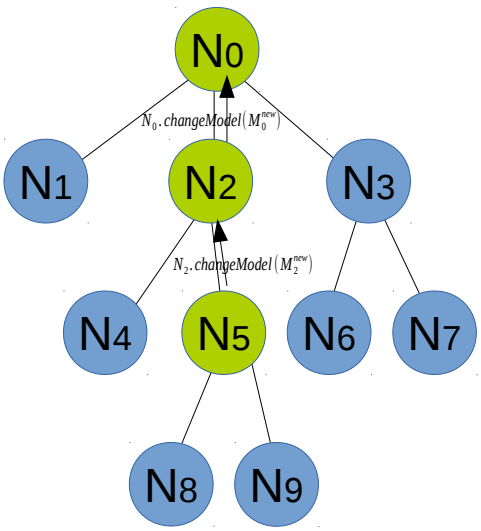


Figure 7: changeModel chain propagation

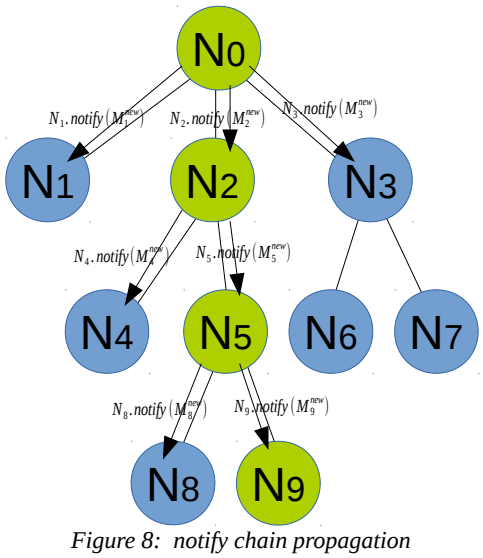


Figure 8: notify chain propagation

When root node is reached, notification chain starts as described in Figure 8. Nodes that are represented in green are the ones that would perform notifications to its listeners.

3 Project work description

In this section it is exposed and explained all the work done during the execution of the project. First the the final application visual result is shown. Then all considerations that have been taken during the workflow are detailed. This section also explains the architecture of the implemented application. Finally programing details about Weathermap implementation are deeply explained.

3.1 Application flow diagram

The aim of this section is to show the changes introduced in Netatmo Weather Station application from the point of view of the final user. Visual details and functionalities of the baseline application and the final application are shown by a flow diagram.

Figure 9 shows the flow diagram of the baseline application. Login screen is the first screen that a user see when launching the application for the first time. When the user logins (1) Dashboard screen is shown.

Dashboard screen is the main screen of the application. Is the first screen when launching the application if the user is already logged in. This screen displays all measures from the selected station. This screen contains a hidden navigation drawer that can be displayed as described in (2). Navigation drawer contains the list of stations that the user owns. By clicking in one element of the list the selected station changes (3). Navigation drawer also allows users to open the Weathermap web application (4).

There are three elements on the Dashboard screen, outdoor view, forecast view and indoor view. By swapping outdoor view (5) different measures from different exterior modules are displayed. By swapping indoor view (6) different measures from different indoor modules are displayed. By clicking forecast view (7) a new screen is displayed with extra forecast details. User can view old measures from the selected station on an interactive graph by changing device orientation (8).

Figure 10 shows a flow diagram that contains all changes introduced in the application. In the new version login screen contains a new button (9) for using the native Weathermap without longing in.
A new slide has been added to the indoor swap view. This slide is a map where selected sttation location is displayed. By clicking on this slide (10) native Weathermap is displayed and centered in the location of the selected station.

In the new application, when navigation drawer is displayed (11), it also shows the list of public stations that the user has set as favorites. When clicking on a favorite station on the list (12), dashboard screen is updated with public data. As indoor measures are private, the only slide shown in the indoor swap view is the map. A user may delete a station from the list of favorites by clicking on the star-shaped button (13).

Navigation drawer also contains a link to open the native Weathermap (14) where it exists a button to display (15) and hide (18) Weathermap controls screen. This screen allows users to change the measurements displayed on the map to rain (16) or wind (17). If user clicks on a station displayed on the Weathermap (19), dashboard information is updated with the measurements of the clicked station. In this state, a user may add the clicked station to the list of favorites by clicking on the star-shaped button (13).

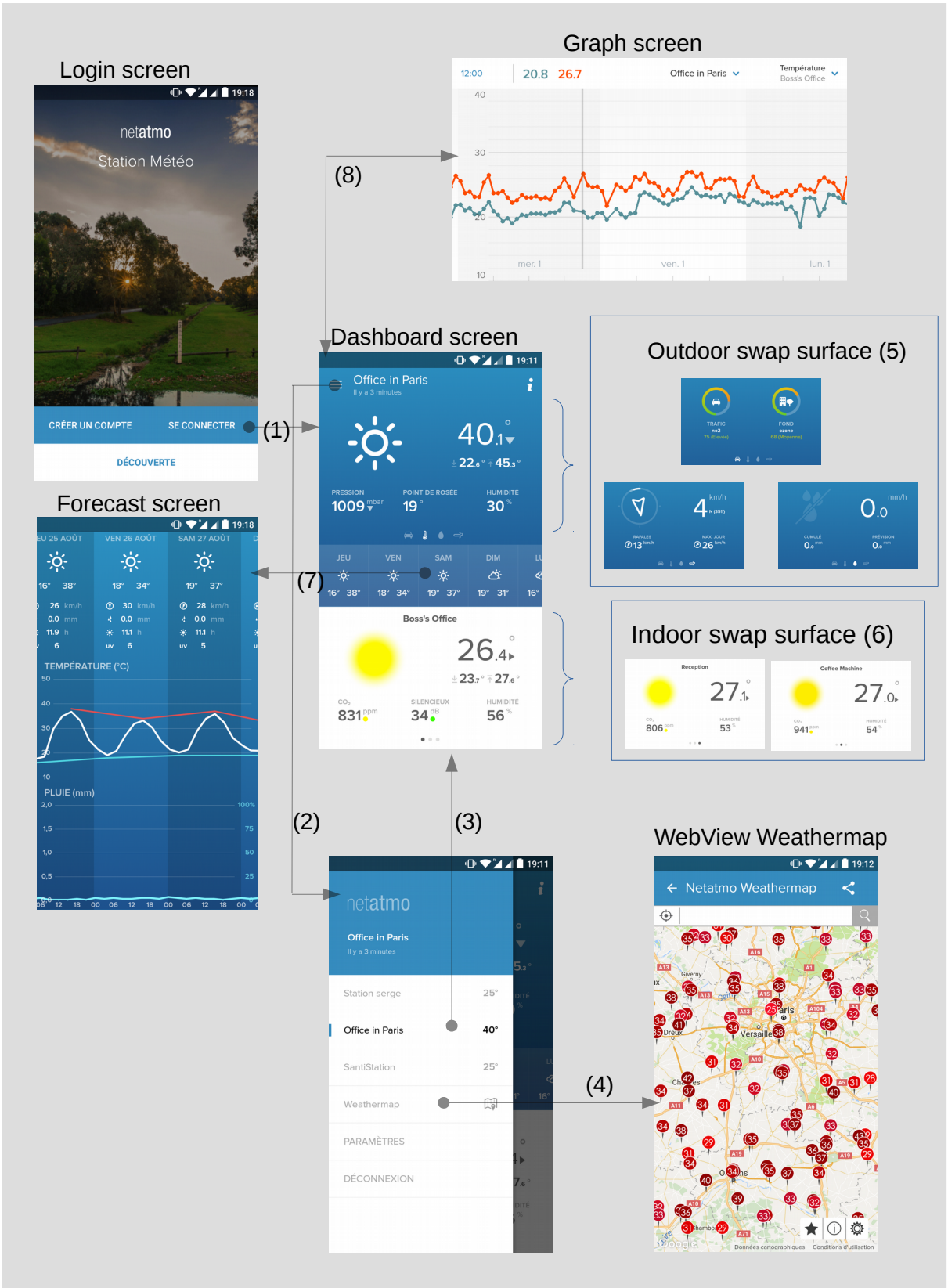


Figure 9: Baseline application flow diagram

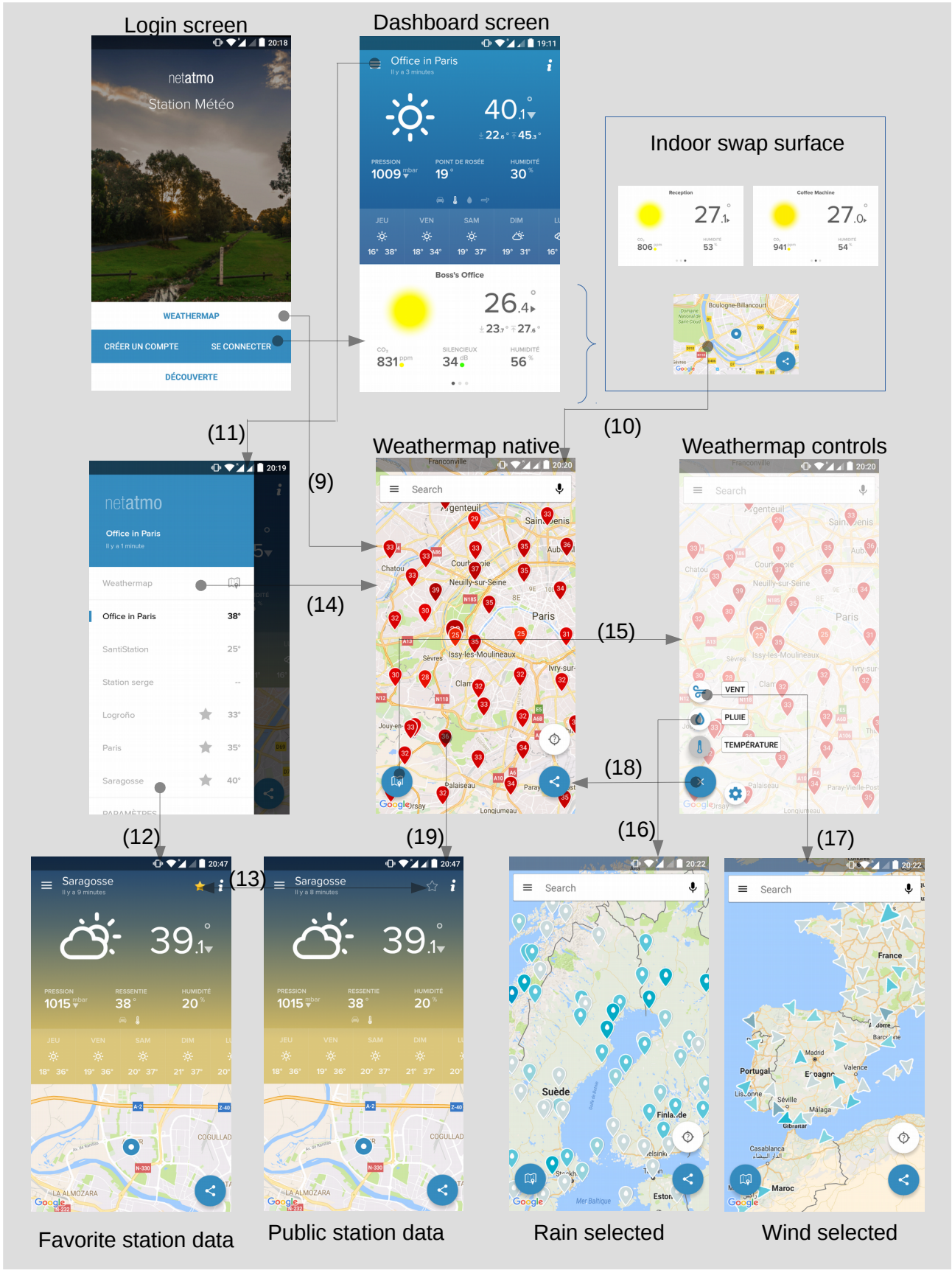


Figure 10: Final application changes flow diagram

3.2 Considerations

This section shows the process that has been followed for deciding all implementation variables of the Weathermap. All theoretical and technical considerations made for solving the problems found during the project are explained in this section.

3.2.1 Getting pubic stations data

Netatmo private REST service provides a method for getting public stations data in a certain rectangular region. This method receives some parameters in order to choose what type of stations we want, how many and in which region. The following table shows a list of parameters that can be used:

Parameter	Description	Possible values
φ_{NE}	North east latitude in degrees of the requested rectangular region. [WGS 84]	$-90 \leq \varphi_{NE} \leq 90$ $\varphi_{NE} > \varphi_{SW}$
φ_{SW}	South west latitude in degrees of the requested rectangular region.[WGS 84]	$-90 \leq \varphi_{SW} \leq 90$ $\varphi_{SW} < \varphi_{NE}$
λ_{NE}	North east longitude in degrees of the requested rectangular region.[WGS 84]	$-180 \leq \lambda_{NE} \leq 180$ $\lambda_{NE} > \lambda_{SW}$
λ_{SW}	South west longitude in degrees of the requested rectangular region.[WGS 84]	$-180 \leq \lambda_{NE} \leq 180$ $\lambda_{NE} > \lambda_{SW}$
N_s	Max number of stations to receive.	$0 < N_s < 1000$
type	All received stations will contain this type of measure.	temperature, rain, wind

For instance, if we send a request with parameters:

$$\varphi_{NE} = 60°, \varphi_{SW} = -60°, \lambda_{NE} = 100° \quad \text{and} \quad \lambda_{SW} = -100°$$

We would receive stations in rectangular area showed in Figure 11. The parameter condition $\varphi_{SW} < \varphi_{NE}$ implies an important restriction when using GoogleMaps that uses a Mercator projection with cyclic longitude implementation. That means that its possible to see in the screen the rectangular area showed in Figure 12. In that case the parameter condition $\varphi_{SW} < \varphi_{NE}$ is not satisfied. If we want to receive stations in the rectangular area shown in Figure 12 it's necessary to perform two request. Each request correspond with a rectangular area that satisfies the condition. In this case the parameters for the requests would be:

$$\text{req1:} \quad \varphi_{NE} = 60°, \varphi_{SW} = -60°, \lambda_{NE} = 80° \quad \text{and} \quad \lambda_{SW} = -180°$$
$$\text{req2:} \quad \varphi_{NE} = 60°, \varphi_{SW} = -60°, \lambda_{NE} = 180° \quad \text{and} \quad \lambda_{SW} = 100°$$

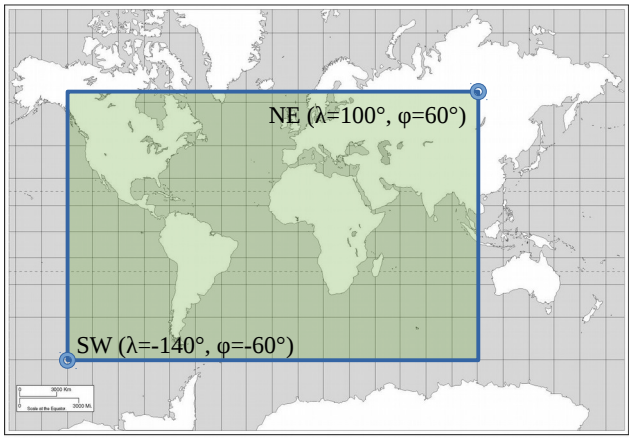


Figure 11: rectangular map region example

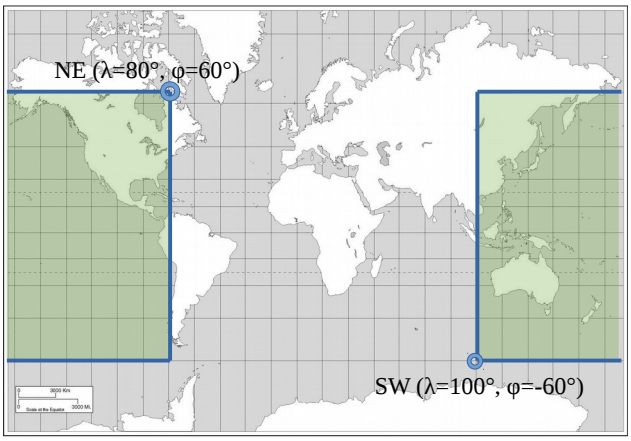


Figure 12: rectangular map region example

3.2.2 Cache system experiment

Looking for stations in a certain area on database is an expensive task in terms of time and resources. When a request is processed by a backend server, the response is stored in cache during 30 minutes. When a new request arrives, if its parameters matches a response in cache, it is not necessary to process the request. Location parameters are compared with a precision of 4 decimals.

Initially, two ways for implementing the Weathermap has been considered:

a) Each time that user moves the map, a single request that matches the screen is done. This way is very simple and robust. However the number of possible request parameters is infinite and as a consequence backend cache system would be never used.

b) Each time that user moves the map, the new map region that is visible in the screen is divided in a set of subregions. If we achieve to normalize these subregions by making them equal for all devices, backend cache system could be used. This way is more complex than the first one and, as a consequence, it could potentially introduce unexpected errors.

Choosing one strategy is a trade off between simplicity and performance. Using cache potentially mean a reduction in the waiting time since the user moves the map until the information is presented. If this time reduction is negligible compared to the delay introduced by the network, the complexity introduced by the second strategy would not be worth.

In order to measure the benefits of using cache an experiment has been made. Using three different devices, several requests has been made with different parameters. In the following table are described all request done during the experiment:

Var	Description	Number
Device model	{Motorola Nexus 6, BQ Aquaris E5, Huawei Ascend G620s}	3
$\varphi_{NE}, \varphi_{SW}, \lambda_{NE}, \lambda_{SW}$	$\begin{bmatrix}\varphi_{NE}\\ \varphi_{SW}\\ \lambda_{NE}\\ \lambda_{SW}\end{bmatrix} = \begin{bmatrix}53.75°\\ 51.4°\\ 13.36°\\ 9.49°\end{bmatrix} + k\cdot\begin{bmatrix}0\\ 0\\ 0.001°\\ 0.001°\end{bmatrix},\ k=0,1,\ldots,499$	500
N_s	{4 , 9 , 16 , 25 , 36 , 49 , 81 , 100 , 144 , 196 , 225 }	11
type	temperature	1
iterations	Each request has been sent two consecutive times. This way the first request uses cache and the second one don't.	2
Total number of requests = $3\cdot500\cdot11\cdot1\cdot2=33000$		

For each request backend execution time T_{exec} has been measured as well as total time T_{tot} since it was sent until the information is attached to the screen. We can assume that T_{tot} is the addition of backend execution time, network delay $T_{network}$ and the time T_{device} required for the device for precessing the response and display the information on the screen. With this assumption $T_{device}+T_{network}$ can be calculated as:

$$T_{d,n}=T_{device}+T_{network}=T_{tot}-T_{exec}$$

The following graphs show the different time measures obtained for different number of stations requested with a 95% confidence interval:

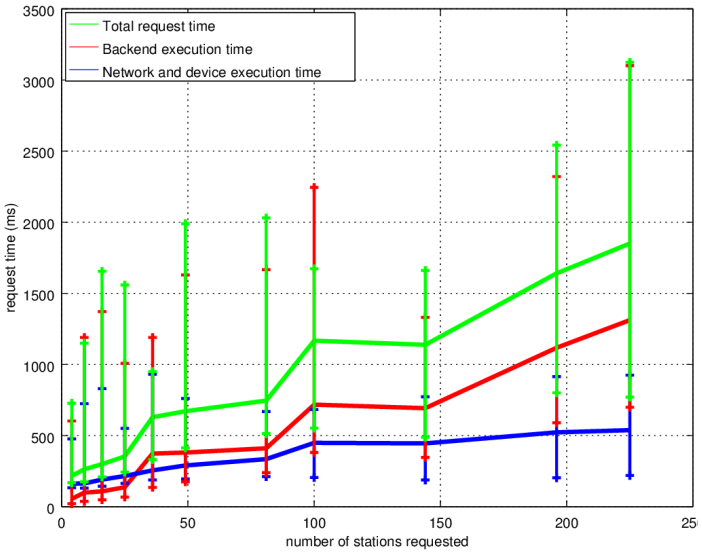


Figure 13: Times measured without cache

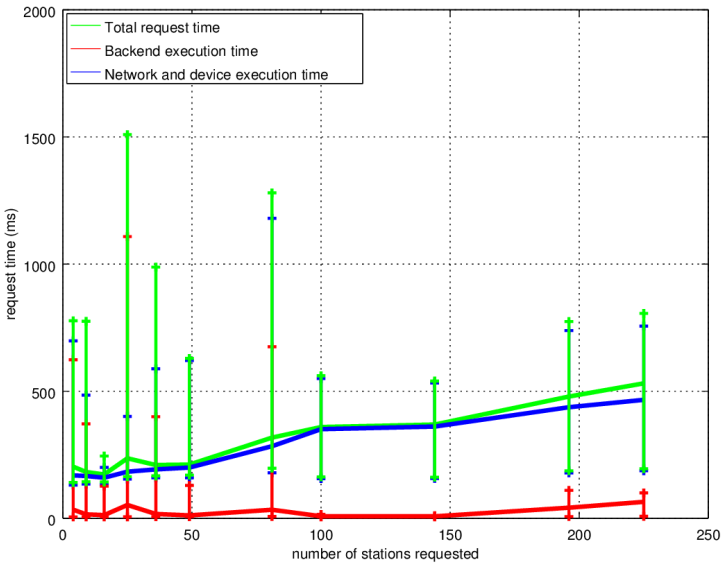


Figure 14: Times measured without cache

Figure 13 shows T_{tot} , T_{exec} and $T_{d,n}$ as a function of N_s for requests without cache. As expected all times increase when N_s increase due to the fact that more data is needed to be searched, transmitted and processed. We can observe that T_{exec} increases faster than $T_{d,n}$, that means that the higher number of stations requested, the more influence execution time has over the total request time. In fact, from a number of stations bigger than 25, more than 50% of total time is due to execution time.

Figure 14 shows T_{tot} , T_{exec} and $T_{d,n}$ as a function of N_s for requests with cache. We can observe that in this case execution time is negligible because no deep database search is needed and as a consequence almost 100% of the total time is due to networking and device processing tasks.

As conclusion, execution time is not negligible. As is expected to display between 30 and 50 stations depending on device screen size, according with this results its possible to save more than 50% of the total request time using cache. Of course, for having this time reduction it's also necessary that the request sent is in cache. However due to the big number of active Weather Station users per day and their graphical distribution its very probably that one user navigates on the same region where other user was navigating 30 minutes ago. Because of the results, we have decided to use the strategy that profits backend cache system.

3.2.3 Map division problem

Lets imagine that two users are navigating in the same world map region. The facto that screens have different size and proportion makes impossible that both users see exactly the same map region in their screens. In addition, GoogleMaps camera position coordinates are codified with double precision what makes it even more impossible. Figure 15 shows a picture to illustrate this situation where rectangles represent the portion of map seen in each screen. If both users make a single request that matches their screen coordinates, the probability of find a response in cache is almost zero.

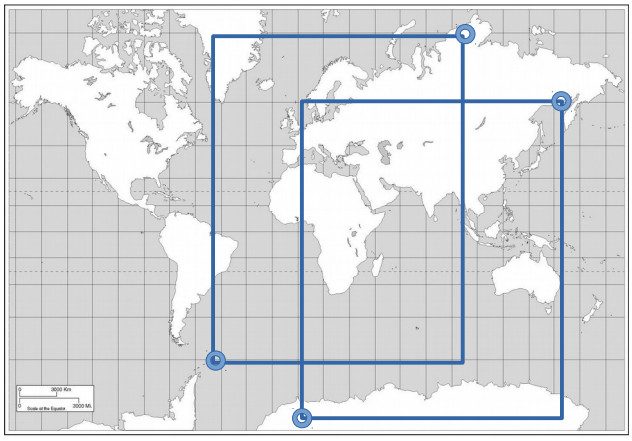


Figure 15: Screens overlapped example 1

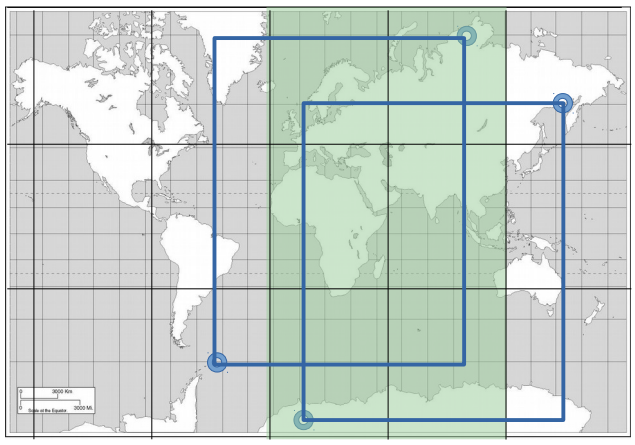


Figure 16: Screens overlapped example 2

To benefit backend cache system it's necessary that all devices navigating in the same world map region make requests with the same parameter values. This condition involves a discretization of the infinite set of all possible world map areas into a finite subset. As backend method only allows to receive public stations in a rectangular area, subset areas must be rectangular shaped.

The simplest way of doing it Is to divide the entire world map area into subareas using a rectangular grid. In Figure 16 world map has been divided using a rectangular grid. In this situation, if each user makes one request for each subregion that is intersected by its screen, green subregions could be shared in the cache system.

First approach

The first strategy that has been considered was based on using geographical coordinates to create a grid similar to the parallel and meridian system. The grid is build by the intersection of vertical parallel lines and horizontal parallel lines. Vertical parallel lines are $\Delta\lambda$ degrees equally spaced and horizontal parallel lines are $\Delta\varphi$ degrees equally spaced.

GoogleMaps uses WGS 84 Web Mercator projection for displaying bi-dimensional maps. Earth surface parametrized by WGS 84 datum and protected onto a rectangular bi-dimensional plane. The height and width of the pane is 256 units. Each point expressed in geographical coordinates (λ,φ) has a projection (x,y) on this plane according to the following equations:

$$\begin{aligned} x &= P_x(\lambda) = \frac{128}{\pi}(\lambda+\pi) \\ y &= P_y(\varphi) = \frac{128}{\pi}\left(\pi - \ln\left[\tan\left(\frac{\pi}{4}+\frac{\varphi}{2}\right)\right]\right) \end{aligned} \quad (1)$$

Where latitude and longitude are expressed in radians. Horizontal component *x* has a lineal dependency with longitude, as a consequence, if parallel vertical lines are equally spaced in longitude, the projected vertical lines will be also equally spaced on the map. However vertical component *y* does not have lineal dependency with latitude. As a consequence parallel horizontal lines wont be equally spaced on the map. This behavior can be observed on Figure 17 where world map has been divided using a grid with parameters $\Delta\varphi = \Delta\lambda = 30^o$

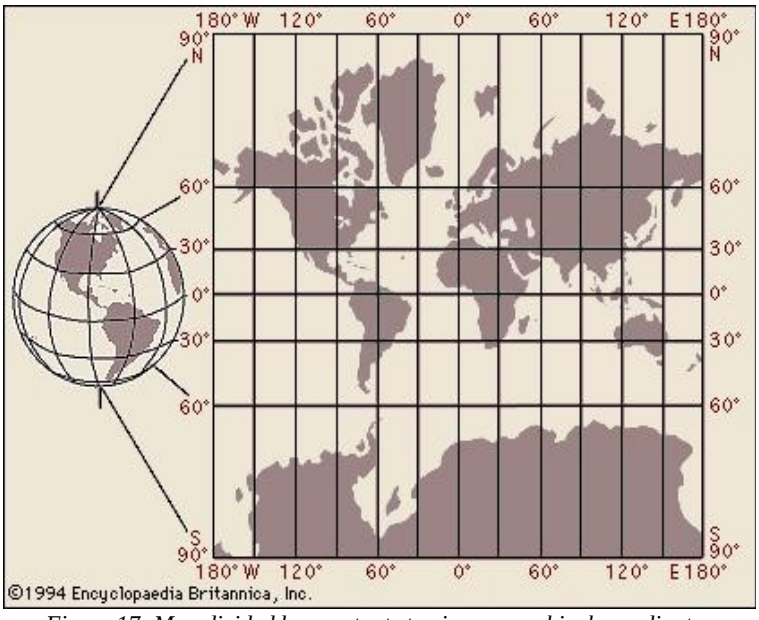
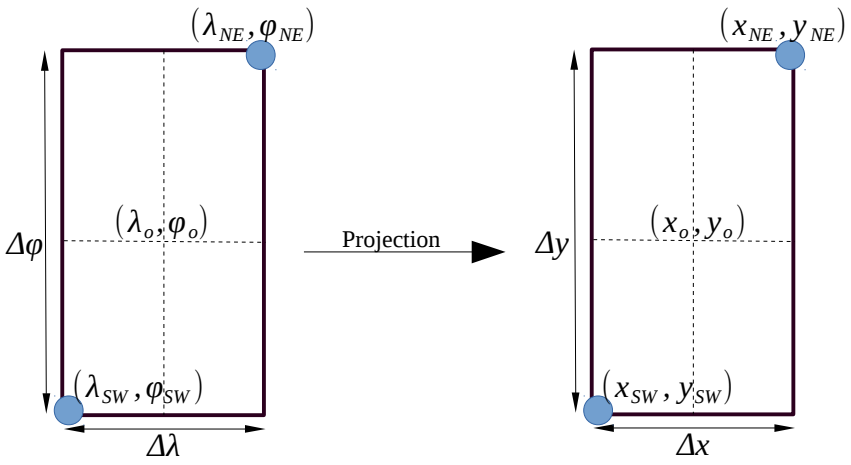


Figure 17: Map divided by constant step in geographical coordinates

As protected lines are not equally spaced the area of each subregion is different. On Figure 17 we can see that the area of subregions placed on the center is smaller that the are of subregions placed on the top and bottom. As a consequence, if we request the same number of stations for each subregion the density of stations will be different for each one. We can use projection formulas in order to know if this density distortion will have a significant visual impact on the Weathermap.

Lets consider a grid with a certain latitude separation $\Delta\varphi$ and longitude separation $\Delta\lambda$. If we consider a particular rectangular whose center coordinates are (λ_o, φ_o) we can calculate its area A in projected units as a function of its center coordinates:



$$A(\lambda_o, \varphi_o) = \Delta x \cdot \Delta y = (x_{NE} - x_{SW}) \cdot (y_{NE} - y_{SW}) = (P_x(\lambda_{NE}) - P_x(\lambda_{SW})) \cdot (P_y(\varphi_{NE}) - P_y(\varphi_{SW})) \quad (2)$$

Applying projection equations (1) and doing the following substitution:

$$\begin{aligned} \lambda_{NE} &= \lambda_o + \Delta\lambda/2 \\ \lambda_{SW} &= \lambda_o - \Delta\lambda/2 \\ \varphi_{NE} &= \varphi_o + \Delta\varphi/2 \\ \varphi_{SW} &= \varphi_o - \Delta\varphi/2 \end{aligned} \quad (3)$$

finally we obtain:

$$A(\lambda_o, \varphi_o) = A(\varphi_o) = \left(\frac{128}{\pi}\right)^2 \Delta\lambda \cdot \ln \frac{\tan\left(\frac{2\varphi_o + \Delta\varphi + \pi}{4}\right)}{\tan\left(\frac{2\varphi_o - \Delta\varphi + \pi}{4}\right)} \quad (4)$$

As expected, rectangular region area does not depend on its center longitude. The minimum area rectangle would be placed on $\varphi_o = 0$ meanwhile the maximum area rectangle would be placed in the higher latitude represented in the map. As maximum latitude represented by WGS 84 Web Mercator projection is 85º, we can do an approach and suppose that the maximum area rectangle is placed on $\varphi_o = 85^o - \Delta\varphi/2$. If we request the same number of station for all subregions, the visual impact on the Weathermap can be measured by the density difference ratio *K* between the biggest subregion area and the smallest subregion area:

$$K(\Delta\varphi) = \frac{A(85^o - \Delta\varphi/2)}{A(0)} \quad (5)$$

If we apply equation (4) in (5) we can observe that *K* only depends on *Δφ*. The following graph displays *K* as a function of *Δφ:*

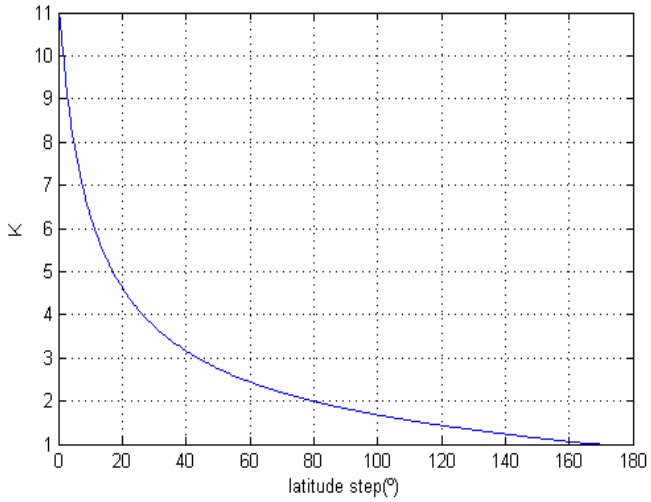


Figure 18: Area distortion for different division step values

We can observe that for *Δφ* values bigger than 85º the density distortion is bigger than 2. This means that if we divide the map using a grid with more than two horizontal lines, the area of some subregions would be at least twice the area of other subregions. This theoretical results shows that this strategy for creating a grid is not correct from a visual point of view.

Final strategy

For obtaining a grid where all rectangles are constant in the projected domain, its necessary to define vertical and horizontal parallel lines equally spaced in projected domain. Now the grid is defined by the intersection of vertical parallel lines equally spaced by *Δx* projected units with horizontal parallel lines equally spaced by *Δy* units. For an easier implementation we have chosen *Δx = Δy = Δl* creating a square grid.

For getting public stations data with this strategy, two unit conversations are needed. Figure 19 shows the process since the user changes screen position until request are made:

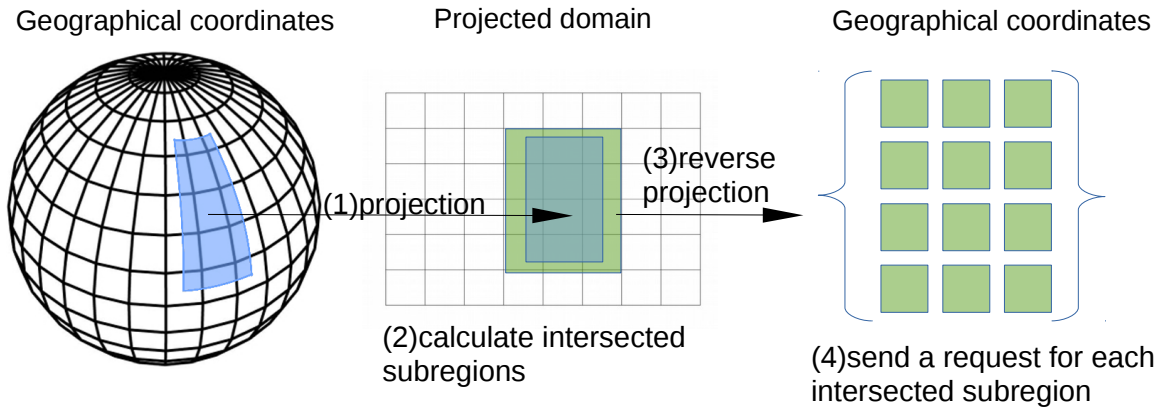


Figure 19: Map division in projected units process

(1) When user moves the screen into a new map position, new position rectangle is projected using WGS 84 Web Mercator projection equations.

(2) The projected screen is transformed into a subset of subregions defined by the grid. This subset is calculated as the intersection of the projected screen with all subregions of the grid.

(3) Geographical coordinates of each intersected subregion are calculated. WGS 84 Web Mercator projection equations are bijective and as a consequence they can be used directly for perform this projection reverse step.

(4) Finally, for each intersected subregion, a request is sent using its geographical coordinates in order to receive public stations.

3.2.4 Map zoom problem

GoogleMaps implements a zoom system. User can increase and reduce map zoom level by spreading and pinching on the screen. This means that, for the same device screen, the represented map region area may be different depending on the zoom level. Or in the other way, the same region in projected units would be represented with a different size depending on the zoom level.

If a single grid is used as described in last section, the number of stations displayed on the map would be different depending on the zoom level. For a good visual performance of the Weathermap the desired behavior involves a constant number of stations displayed on the screen regardless the zoom level.

A single grid with constant parameters Δx and Δy is no longer valid. Grid parameters must be a function of the zoom level z in order to adapt grid squares size :

$$\Delta x(z) = \Delta y(z) = \Delta l(z)$$

For finding a valid adaptation function first is necessary to know how zoom system is implemented in GoogleMaps. Zoom level z is codified by a real number between 1 and 20. Given a map region defined in projected units, this value determines the number of pixels used for representing that region. The bigger the zoom value is, the more pixels will be used. When user performs a spreading

gesture on the screen, zoom level is increased by a quantity of units proportional to the spreading movement distance. In the same manner, when user performs a pinching gesture, zoom level is decreased by a quantity of units proportional to the pinching movement distance.

New equations are defined in order to project each geographical point into a bi-dimensional position in terms if pixels. These equations are based on WGS 84 Web Mercator projection equations (1) and are defined as:

$$x_{pixel} = 2^z P_x(\lambda) = 2^z \frac{128}{\pi}(\lambda + \pi)$$
$$y_{pixel} = 2^z P_y(\varphi) = 2^z \frac{128}{\pi}\left(\pi - \ln\left[\tan\left(\frac{\pi}{4} + \frac{\varphi}{2}\right)\right]\right) \quad (6)$$

We can observe a base 2 exponential dependency between the projected pixel position and the zoom level. This means that for a given geographical point, if we increase zoom level by one unit, its projection components in terms of pixels will be multiplied by 2.

For compensate this exponential effect, grid parameters must have an exponential dependence of zoom values:

$$\Delta x(z) = \Delta y(z) = 2^{-z} L_o \quad (7)$$

Where L_o may be interpreted as the size of the squares in a grid when zoom value is zero. We can configure the size of the squares by changing L_o value.

Using equations in (7) we obtain an adaptive grid where square regions size is constant in terms of screen pixels. As the set of possible values of zoom levels is infinite, using directly equation in (7) would create a infinite set of grids and the probability of using backend system would be zero.

In order to get a finite set of grids its necessary to transform the infinite zoom domain *z* into a finite domain $\hat{z}$. We can achieve this discretization with a quantification function with uniform quantification step Δz :

$$\hat{z} = \left\lfloor \frac{z + \Delta z/2}{\Delta z} \right\rfloor \Delta z \quad (8)$$

Equation described in (8) transforms a zoom value into the closest multiple of Δz value. Choosing the optimal Δz value is a trade off between probability of using cache system and visual performance :

- The bigger Δz value is, the less number of possible zoom values and as a consequence the less number of possible grids. If the number of grids is reduced, the total number of subregions is reduced too and its more probable that two users navigate on the same subregion.

- If Δz is small, the grid size change when the user zooms in or zooms out would be small too. This means that the change in the number of stations in the screen would be smooth.

We have decided to choose the biggest Δz value that involves a transition smooth enough to create a pleasant visual sensation. Different Δz values have been tasted with different users that have made a qualitative assessment of the performance. Finally $\Delta z = 1$ has been chosen.

3.2.5 Choosing subregion size experiment

The only parameter that remains to be defined is the base length L_o of the squares in the grid. The goal is to choose a value that optimizes the time since the user moves the screen until the information is attached on the screen. A priori we can make some assumptions:

-The bigger is the value of L_o the fewer subregions would be intersected by user screen and, as a consequence, fewer requests would be needed consuming less network resources. Moreover, as each request would receive an HTTP response, the more request are made the more overhead introduced by HTTP headers.

-On the other hand, as the goal is having a constant public stations density on the screen, the more number of intersected subregions the fewer number of station are needed in each request. Simulation results in Figure 13 show that the execution time in backend server can be significantly reduced by requesting small number of stations. As backend server processes requests in parallel, making a big number of request may involve a global time reduction.

-Results in Figure 13 also show that the time required by the device to process a server response can be reduced by requesting small number of stations. In the same manner, if the responses are processed in the device in parallel, global time may be also reduced by using small L_o values. However making assumptions about Android parallelization issues may be dangerous because of the big variety of microprocessors on the market.

As this optimization problem involves too many variables for being solved in a theoretical way, an experiment has been made in order to find the optimal L_o value. Three Android devices have been used with very different computing capabilities to do the experiment as representative as possible. For different world map regions a constant number of public stations has been requested using different values of L_o. The following table shows the details about the experiment:

Var	Description	Number
Device model	{Motorola Nexus 6, BQ Aquaris E5, Huawei Ascend G620s}	3
L_o	{720,640,560,480,400,320,280,240,200,160,120,80}	12
Screen regions	$\begin{bmatrix}\varphi_{NE}\\ \varphi_{SW}\\ \lambda_{NE}\\ \lambda_{SW}\end{bmatrix}=\begin{bmatrix}53.75°\\ 51.4°\\ 13.36°\\ 9.49°\end{bmatrix}+k\cdot\begin{bmatrix}0\\ 0\\ 0.001°\\ 0.001°\end{bmatrix},\ k=0,1,\ldots,30$	30
N_s	100 to be divided between each subregion.	1
type	temperature	1
iterations	Each request has been sent two consecutive times. This way the first request uses cache and the second one don't.	2
Total number of requests = $3\cdot30\cdot12\cdot2=2160$ (requests for each subregion are not considered)		

The following graphs shows the different time measures obtained for different values of L_o with a 95% confidence interval:

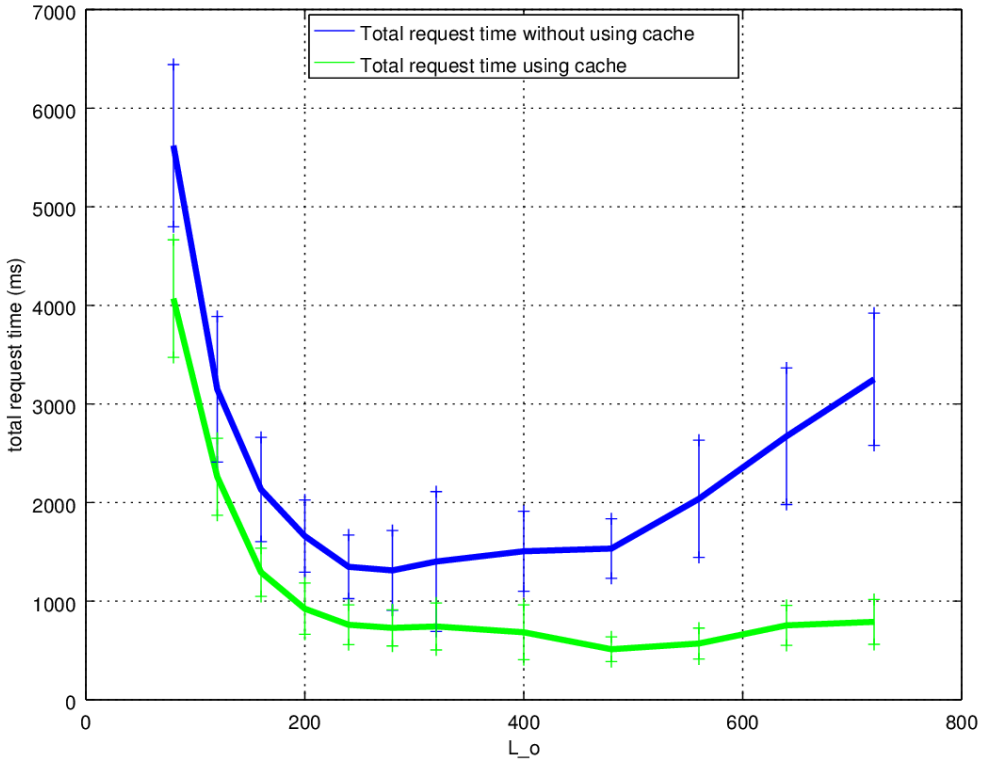


Figure 20: Time measured for different square sizes

We can observe that for values of L_o smaller than 200 the total request time is very big for both cases. This may be caused of the big overhead introduced by making a lot of requests.

For the curve where cache is used we can observe that it exists a minimum at $L_o = 280$. For values bigger than 480 total request time starts to grow lineally. This behavior may be due to the fact that at this operation point the number of stations requested per request is very big and backend executions time makes the most significant contribution to the global request time.

On the other hand, the curve where cache is not used remains almost constant for values of L_o bigger than 200. As no backend processing is needed, execution time is not a constrain.

Although according to the results $L_o = 280$ is the optimal point in terms of time, $L_o = 480$ has been used for implementing the system. As can be observed on the results, the mean request time for $L_o = 480$ is only 100ms higher than for $L_o = 280$. However, using $L_o = 480$ we are receiving potentially stations in a bigger area than using $L_o = 280$ and, as consequence, its more probable that user moves into a new region that is already included in a previous request.

3.3 Application architecture

The aim of this section is not to explain in depth the implementation of the application. The overall architecture is explained from a low level point of view in order to understand how new features have been added.

3.3.1 Baseline application architecture

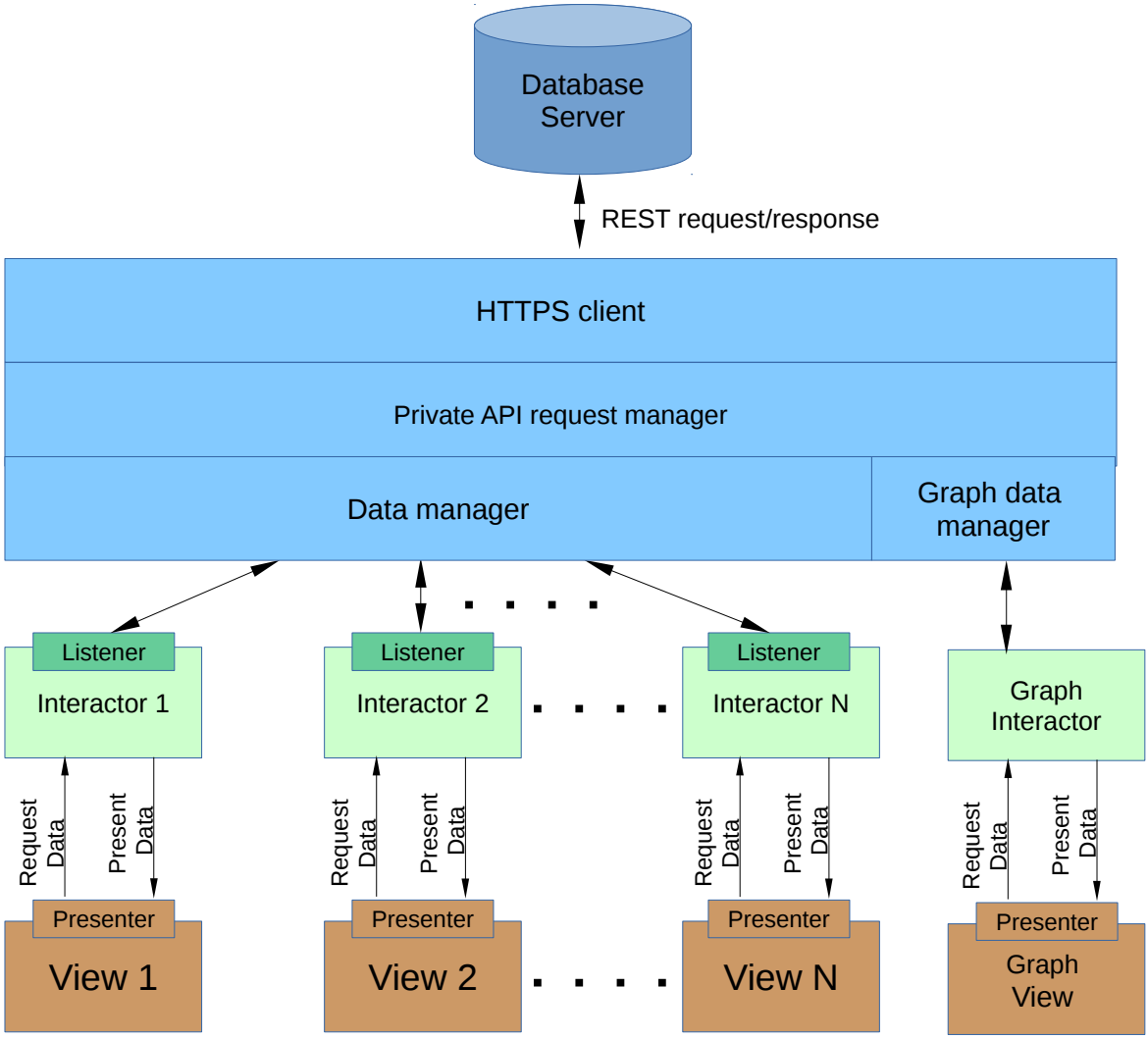


Figure 21: Baseline application architecture

Following the **presenter-interactor** design pattern, presentation, domain and data layer can be identified in red, green and blue colors respectively. Each view contains an interactor and works independently from other views.

Interactors are in charge of recover and process data that will be presented to the presentation layer. The only interactor that comunicates with networking modules is the **graph view interactor**. The rest of them communicate with **data manager** module. This module implements a cache system to optimize the number of requests needed by the application. When an interactor request data to the **data manage**r module, this checks if that data is already in cache. If not, it performs the API calls necessaries to receive all the requested data.

Data manager has been designed for getting and storing actualized data such as last measurements, user profile or weather station status. However the amount of data needed by the graph is much bigger due to the fact that using graph involves a big collection of historical measures. **Graph interactor** do not use data manager module because managing graph data requires more resources and a more sophisticated cache system.

The network connection with the REST server has been implemented using HTTP Volley library. Module **HTTPS client** uses the resources in that library to perform REST requests and to receive responses. This module is also in charge of parsing server JSON responses and transforming them into Java objects.

The rest of the application can interact with the networking layer through **Private API request manager**. This module is an interface to publish the REST methods that are available in the server. This module is also in charge of checking that all the parameters for each request are correct.

3.3.2 First approach application architecture

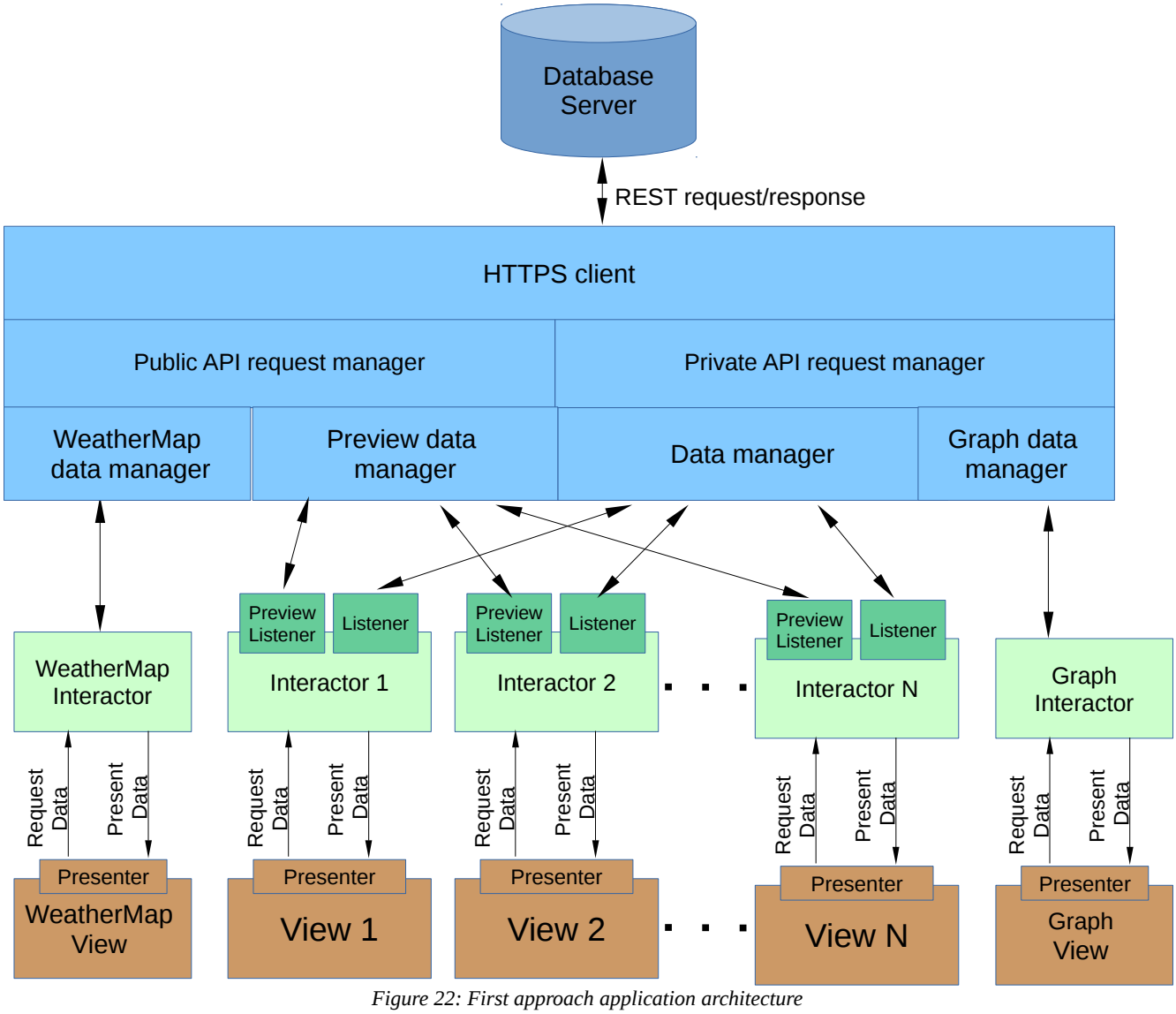


Figure 22: First approach application architecture

Software incremental strategy [5] has been used to implement the desired new features. The aim of this strategy is to create a new software system based on a existing one without changing the behavior of the existing system.

The main idea is to create a new architecture that satisfies the desired new requirements without changing the existing implementation. The baseline application was released 2 years ago and has been deeply tested. With this strategy if a new bug appears after implementing the new features, we can be sure that the error is due to the new code.

All data used in the baseline application belongs to a user private context. That means that the data displayed only can be read by the owner of the station. However for implementing the new features a new type of public data is necessary:

-Weathermap: Its necessary to receive location and measures of stations that do not belong to the user.

-PreviewVisualization: When the user selects a station it is necessary to update the dashboard with extra information.

-Add Favorites: When the user adds a station to favorites, user station list is updated and pushed to the server. From that moment all data from the added station belongs to the user private data context.

The existing module **Private API request manager** only implements methods to receive user private data. As the objective is not modify the existing architecture, a new API module has been included instead of expanding the existing one. It is called **Public API request manager** and is in charge of provide an interface for publishing all server methods needed to get public data.

Preview data manager module has the same role as data manager module but with public data. It implements a cache system and isolates interactors from networking tasks.

In this architecture, interactors communicate with both data managers. They are in charge of choosing the correct data manager depending on the state of the application when a view request data. Thanks to this architecture based on presenter-interactor design pattern all visual elements have been fully reused because, from the point of view of presentation layer, the data precedence is transparent.

The behavior of the **Weathermap** is very similar to the graph. Both require a big amount of data during a short period of time. Because of this Weathermap architecture follows the same schema as the graph. Weathermap interactor contains an exclusive data manager in order to cache and request data more efficiently.

3.3.3 Final application architecture

Netatmo programmers team have developed a new deliver information system called **Netflux**. More details about this system can be found in section 2.3. I was also in charge of including this new system in the Weather Station application.

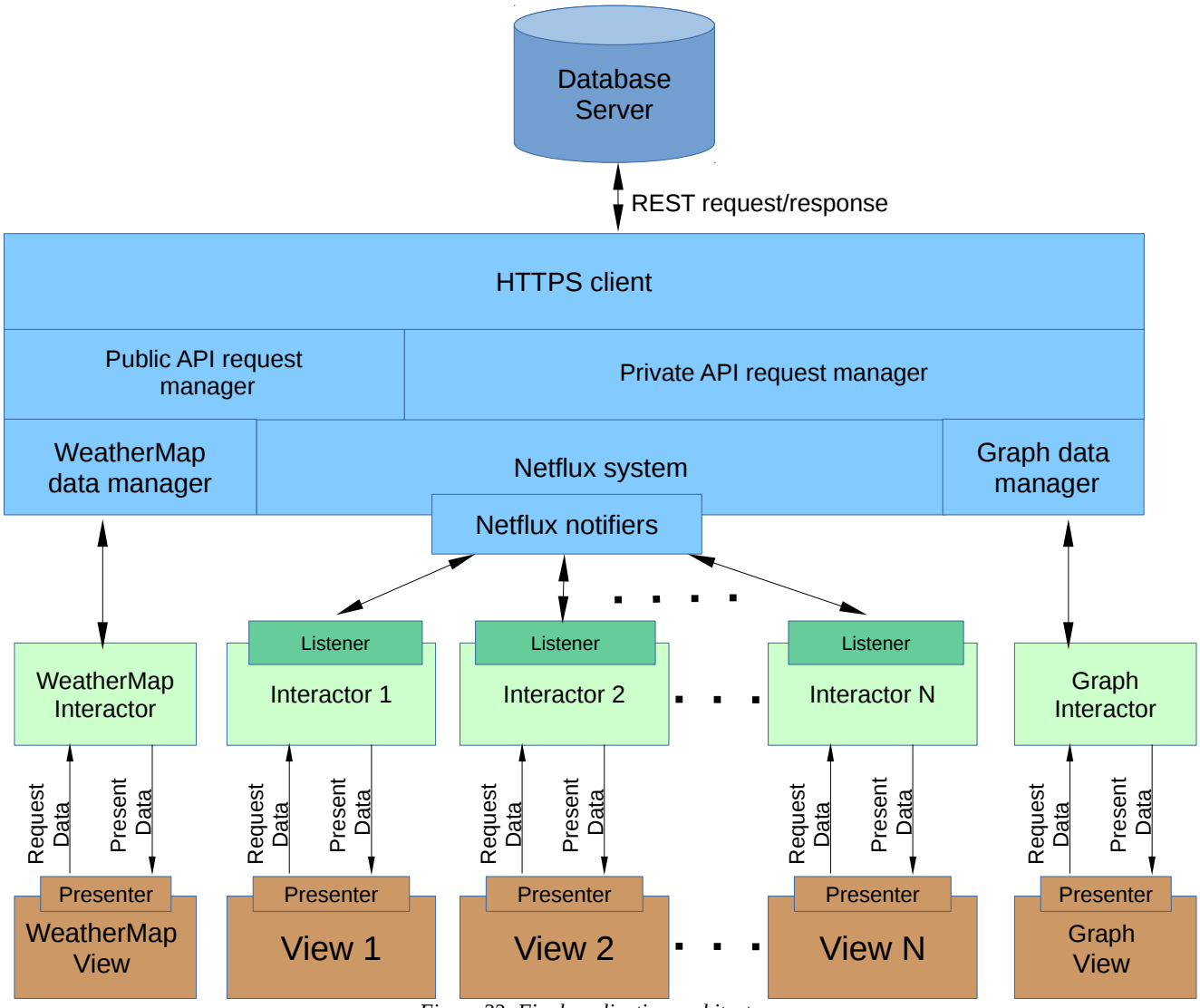


Figure 23: Final application architecture

Now there is no separation between public and private data. Netflux data model contains both kind of data and have access to public and private APIs. When an interactor needs data, netflux module extracts from the data model the piece of information requested. When the model changes, maybe for a new REST response or maybe for new application state, netflux module notifies the change to all interactors that are interested in the part of the model that has been changed.

Netflux module only changes the way of storing and delivering application data. The rest of the architecture such as presentation and networking layer remain constant.

3.4 Weathermap implementation

In this section Weathermap implementation will be deeply explained. This is the only component of the application that is explained with code details in this document because:

-It represents the core of my work during the internship.
-It has been designed and implemented entirely by me
-The rest of my contribution to the application has been modifying other components that have been designed and implemented by other programmers of the company.

I the following figure we can see the class diagram for the Weathermap implementation. Only public methods are included:

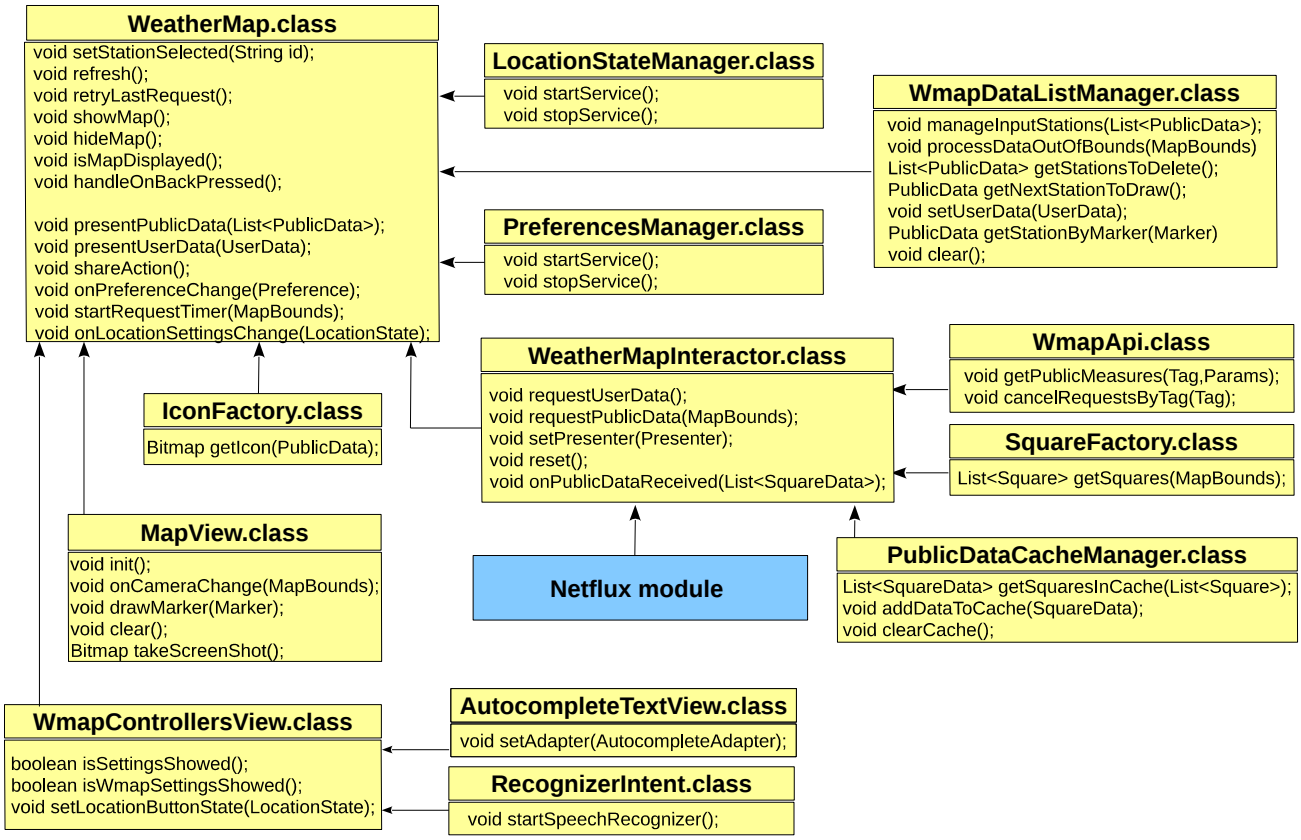


Figure 24: Weathermap implementation class diagram

Single responsibility paradigm has been used for the design. Each class corresponds with a single functionality and functionalities are not shared between classes. However, due to the nature of the Weathermap behavior, all functionalities are highly coupled. For example user location service must interact with presentation layer to display user position in the map or user preferences service must interact with the graphical interface to provide an output of the preferences selected.

Other criteria followed in the design is that each class corresponds with a project specification. This way if there is a change in a particular specification, only a class needs to be changed.

WeatherMap.class

This class is the core of the implementation. It centralizes communications between other classes and provides an interaction mechanism for the rest of the application. All components in the application interact with the Weathermap trough this class.

We can divide methods in this class into “exterior” and “interior”. Exterior methods are used by other classes in the application that do not belong to the Weathermap implementation. Interior methods are used by other Weathermap implementation classes.

Exterior methods:

-*void setStationSelected(String id);*

This method is used for setting the current selected station that is displayed on dashboard.
The parameter is an id that is unique for each existing station.

-*void refresh();*

This method forces the Weathermap to refresh the information displayed on the screen. There is a timer service in the application that forces all visual elements to be refreshed periodically.

-*void retryLastRequest();*

This method is used when a connection problem has been detected. When called, the last request performed is repeted.

-*void showMap() / hideMap();*

Make the Weathermap visible/invisible with an animation.

-void isMapDisplayed();

Returns true if Weathermap is visible

-*void handleOnBackPressed();*

This method handles what to do when the user clicks the go back Android button. The action to be done depends on the status of the graphical interface of the Weathermap. For example if settings are showed the action is to close it or if Autocomplete suggestions are displayed the action is to hide them.

Interior methods:

-void presentPublicData(List<PublicData>);

Called by *WeatherMapInteractor* to deliver ready to display data about public stations. *PublicData* class represents a public station.

-void presentUserData(UserData);

Called by *WeatherMapInteractor* to deliver information about the user. *UserData* class contains all data related to the user such as the list of stations, measure units preference or language. Some functionalities depend on this information.

-void shareAction();

This method is called when share button is clicked. When called, a screen-shot of the map is taken and delivered to the current Activity to start a *shareIntent*.

-void onPreferenceChange(Preference);

Called when a user setting has been changed. *Preference* contains information about the change. In this method all elements are configured according to the new user preference.

-void onLocationSettingsChange(LocationState);

Called when location permission has been changed. *LocationState* represents a permission state.

-void startRequestTimer(MapBounds);

WeatherMap class contains a timer. When this method is called, if the timer is stopped, it is launched. If not, the timer is restarted. When the timer expires a request is performed in order to receive public stations data in a certain world region given by *MapBounds*.

This mechanism avoids to send unnecessary requests when the user is navigating with short and consecutive touches on the screen. A request is sent if, during a certain period of time from the last touch event, there are no new touch events. This period of time is configured in the timer and has been set to 750 ms.

IconFacory.class

This class is in charge of providing icons that will be displayed on the Weathermap. Due to the fact that graphical resources related to icons are stored as vectors, this class implements logic to transform a vector into a bitmap taking into account different screen resolutions and Android version compatibility issues.

-Bitmap getIcon(PublicData);

Given a public station, this method returns a bitmap that contains the icon to be draw. *PublicData* contains all data related with the measurements of a received public station.

MapView.class

This class contains and controls all graphical resources related to the map. As GooglaMaps API has been used, this class contains a *GoogleMap* instance. This class, provided by GooglaMaps API, is an interface to configure and control a GoogleMap in an Android application.

-void init();

This method initializes and configure *GoogleMap* instance. Native GoogleMaps is a free service but it requires application level authentication using an access token. During this method authentication process is performed. After authentication, *GoogleMap* instance is configured to listen map camera changes events and clicks on markers(icons).

-void onCameraChange(MapBounds);

This method is called when there is map camera change. Parameter *MapBounds* contains the bounds of the new camera position in geographic coordinates.

-void drawMarker(Marker);

This method draws an icon on the map. *Marker* class is provided by GoogleMaps API and contains the location of the icon and the bitmap to be draw with.

-void clear();

This method delete all icons draw on the map.

-Bitmap takeScreenShot();

This method takes a screen-shot of the current state of the map and returns it as a bitmap. All icons are included. This method is used when the user clicks on share button.

WmapControllersView.class

This class manages all graphical interface elements except the map. It is in charge detecting user interactions with each button and create transition animations according to each user event. It implements public methods used by *WeatherMap* to know the current graphical interface state.

This class also manages the bar on the top of the Weathermap. The bar is a complex graphical element that implements an auto-complete search mechanism and the possibility of search a place by voice. An exclusive class has been created for each functionality.

AutocompleteTextView implements an auto-complete text view using GooglePlaces API. When the user writes a sequence of characters, a GooglePlaces request is sent in order to obtain a list of suggested places based on the input sequence. This process requires application authentication and is performed by an instance of *AutocompleteAdapter* that is initialized by *WmapControllersView. AutocompleteAdapter* implements an adapter interface for *AutocompleteTextView* fin order to display suggestions when they are ready.

About voice search, Google provides a class called *RecognizerIntent* that automatically creates a new Activity that internally uses GoogleSpeechRecognition service. The recognition result is received by *WmapControllersView.*

-*boolean isSettingsShowed();*

returns true is preference settings are displayed.

-*boolean isWmapSettingsShowed();*

returns true is change measurement buttons are displayed.

-*void setLocationButtonState(LocationState);*

This method changes location button image according to the application location state given by *LocationState*.

WmapDataListManager.class

This class resolves the next problem. Imagine that in a certain moment there are some public stations draw on the map. The user touch the screen and navigates to a new map region where some station are already draw. We will receive new public stations data for this new area that are already draw on the map. The simplest solution is to delete all stations and redraw all when new data received, however this solution is inefficient and generates an annoying blinking effect in those icon that are deleted and redraw. *WmapDataListManager* implements a better solution, when new stations arrives, it decides what stations need to be draw, what stations need to be deleted and what stations remain constant. This class also makes all user station being displayed on the map.

We can formalize the problem to understand better the implementation of this class. Consider *M* as the set of stations that are displayed on the map in a certain moment. Then the user navigates and we receive a set of stations *R* to be displayed in the new region. As user stations must be displayed, if we define U as the set of user stations, the set of stations that need to be displayed is given by $R \cup U$. Following Figure 25 we can affirm:

Stations that are in *M* and in $R \cup U$ belongs to the set of stations *N* that have to remain constant.

$$N = M \cap (R \cup U)$$

Stations that are in *M* but not in $R \cup U$ belongs to the set of stations *C* that have to be deleted.

$$C = M - (R \cup U) = M - N$$

Stations that are in $R \cup U$ but not in M belongs to the set of stations D that have to be draw.

$$D=(R\cup U)-M=(R\cup U)-N$$

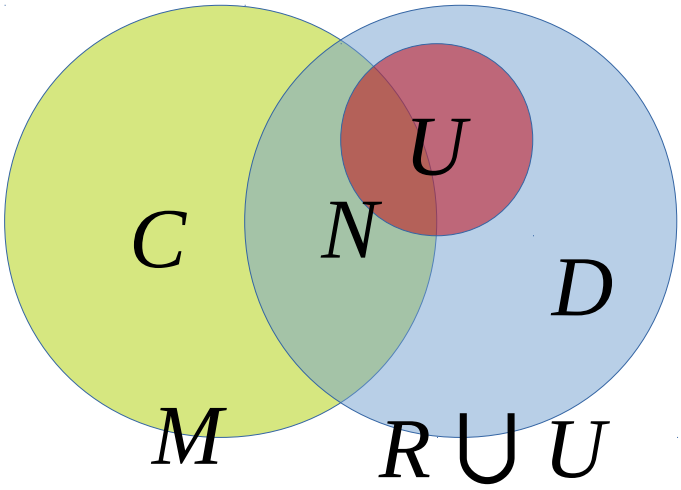


Figure 25

According to the formulas, when a new set of stations R arrives, the operations to be performed are:

$$\begin{aligned} R' &\leftarrow R\cup U \\ N &\leftarrow M\cap R' \\ C &\leftarrow M-N \\ D &\leftarrow R'-N \end{aligned} \quad \text{(step 1)}$$

After deleting all stations in C, sets should be actualized as follows:

$$\begin{aligned} C &\leftarrow \emptyset \\ M &\leftarrow N \end{aligned} \quad \text{(step 2)}$$

After drawing a station d from D, sets should be actualized as follows:

$$\begin{aligned} D &\leftarrow D-\{d\} \\ M &\leftarrow M+\{d\} \end{aligned} \quad \text{(step 3)}$$

After drawing the last station sets should satisfy:

$$\begin{aligned} D &= \emptyset \\ M &= R' \end{aligned}$$

We have solved the problems using set notation. All operations that has been used such as union, intersection addition and substation are already defined in Java List abstract class. Programming the algorithm described above using Java *List* interface is trivial.

-void manageInputStations(List<PublicData>);

This method receives a list of new public stations to be displayed and calculates what stations in the map must be deleted and what new stations must be draw. It runs algorithm step 1.

-List<PublicData> getStationsToDelete();

Returns a list with the stations that must be deleted. It also runs algorithm step 2.

-PublicData getNextStationToDraw();

Returns a new station to be draw or null if the list is empty. It also runs algorithm step 3.

-void setUserData(UserData);

This method is used to set the user station list.

-PublicData getStationByMarker(Marker);

Returns the public station associated to a certain marker given as a parameter. *Marker* represents an icon draw on the map. This method is used by *WeatherMap* to recover a public station info when its icon is clicked. If there is no station in the list whose marker associated match the parameter, the method returns null.

LocationStateManager.class

The functionality of this class is to interact with the OS to determine if the application has location rights to get user geographical position. It also listens OS events in order to detect location settings changes.

-void startService();

When this method is called, this class starts to listen OS events.

-void stopService();

When this method is called, this class stops listening OS events.

PreferenceManager.class

This class is in charge of loading user preferences for Weathermap configuration. It also listens to changes in user preferences. Basically, when user changes a setting concerning to Weathermap, this class notifies the change to *WeatherMap.*

-void startService();

When this method is called, this class starts to listen user settings changes.

-void stopService();

When this method is called, this class stops listening user settings changes.

WeatherMapInteractor.class

This class implements an interactor for *WeatherMapView*. It is in charge of collecting all data needed by Weathermap and processing the data to make it ready to be displayed.

When user information is needed, *WeatherMapInteractor* request user data to the *Netflux* module. However, when new public stations data is needed, this class interacts directly with networking layer thought *WmapApi*.

As seen in section 3.2, to profit backend cache system, map bounds request parameters must be transformed into a set of square bounds that are normalized to all screen densities. When its presenter request new public data in a certain bounds, this class uses *SquareFactory* in order to obtain the set of normalized squares associated to the requested bounds.

WeatherMapInteractor also implements a local cache system for public data using an instance of *PublicDataCacheManager.* Once the squares are calculated, *PublicDataCacheManager* determines for what square regions there is already data in cache and for what square regions a new request is necessary. When a new public stations data arrives from networking layer, it is associated to a certain square region and stored in cache.

-void requestUserData();

When this method is called, *WeatherMapInteractor* starts getting user data from Netflux system.

-void requestPublicData(MapBounds);

When this method is called, *WeatherMapInteractor* starts getting public stations data.

-void setPresenter(Presenter);

Sets the presetner associated to this interactor.

-void onPublicDataReceived(List<SquareData>);

This method is called by networking layer when a public data response is ready.

4 Conclusions

This section contains a list of things that I have learned during the execution of this project. The elements of this list does not strictly belong to the domain of Computer Science. They are useful lessons about how the work environment works.

-Documentation is probably the most important step during the execution of a project. It is important to understand very well the problem you are trying to solve. In this case, knowledge about map projection has been useful to predict potential problems during implementation.

-Simulations and tests are very important in order to detect possible problems. In this case the implemented application was tested by people outside the project. This allowed to find many mistakes I had missed.

-Communication between team members is crucial. Different people with different backgrounds were involved in this project. Marketing people, designers and programmers have a different points of view about how to manage a project. Specifications were created in an iterative process where all opinions where taken into account and, from my opinion, that is the key to carry out a project with the highest possible quality.

5 References

[1] DeMarco, Tom. (1979). Structured Analysis and System Specification. Prentice Hall. ISBN 0-13-854380-1.

[2] Page-Jones, Meilir (1988). The Practical Guide to Structured Systems Design. Yourdon Press Computing Series. p. 82. ISBN 978-8120314825.

[3] Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. pp. 95–98. ISBN 0-13-597444-5.

[4] Ralph E. Johnson & Brian Foote (June–July 1988). "Designing Reusable Classes". Journal of Object-Oriented Programming, Volume 1, Number 2. Department of Computer Science University of Illinois at Urbana-Champaign. pp. 22–35.

[5] Kung-Kiu Lau, Keng-Yap Ng, Tauseef Rana, and Cuong M. Tran. Incremental construction of component-based systems. In Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012, part of Comparch ’12 Federated Events on Component-Based Software Engineering and Software Architecture, Bertinoro, Italy, June 25-28, 2012, pages 41–50, 2012.