IV.1 Introduction :

Les différents opérateurs mobiles utilisent plusieurs méthodes d'évaluation comme le Drive Test qui sert à veiller sur l'assurance de la qualité de service de leurs réseaux, l'optimisation et la surveillance des paramètres radio de l'Air Interface.

La finalisation de notre projet a mis en disposition une application qui donne une approche de ces techniques en offrant la possibilité d'écoute des paramètres radio, la géolocalisation ainsi que l'analyse wifi.

Dans ce chapitre nous allons présenter le développement et l'exploitation de notre application ainsi que le traitement des informations récupérées au niveau de la région de Tlemcen. Cette application peut être exécutée directement sur un mobile Android à partir du package.

IV.2 Structure de l'application :

L'application que nous avons développée figure-xx est composée de 5 modules :

- Détection de la signalisation GSM au niveau de l'Air Interface.
- Détection des réseaux wifi en proximité.
- Localisation GPS et affichage de la position sur Google Maps.
- Historique des mesures.

IV.2.1 La Localisation GPS et l'affichage de la position sur Google Maps :

Le premier service que notre application offre est de permettre à l'utilisateur de se localiser en connaissant sa longitude, sa latitude ainsi que l'adresse où il se trouve lors de l'utilisation. Et comme nous avons décrit dans le chapitre précédent dans la partie de la géolocalisation sous Android, la partie suivante va bien détailler l'utilisation de ces APIs ainsi que la structure du module et son diagramme de classe.

IV.2.1.1 Récupération de la clé :

La première étape que nous avons fait après la création du notre projet sous l'environnement du travail est la récupération de la clé Google Maps API pour pouvoir afficher la carte Map sur notre périphérique, il suffit juste d'avoir un compte Gmail pour l'avoir et la figure suivante montre notre clé utilisée.

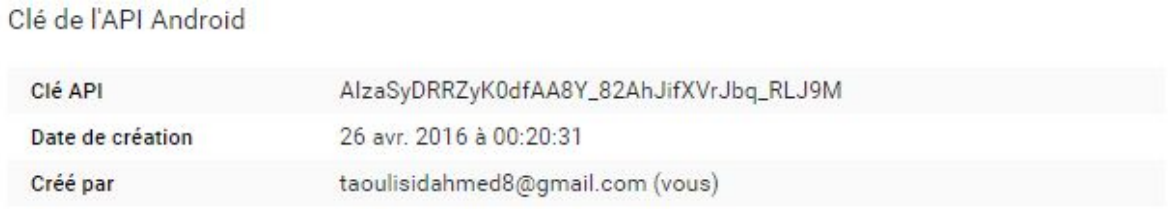

Figure IV.1 : *La clé API utilisée dans le module de la localisation.*

IV.2.1.2 Structure du module :

La classe responsable sur la géolocalisation a été nommée **MapsActivity**, elle étend le fragment **FragmentActivity** qui est le composant principal qui facilite l'utilisation de la carte sous forme d'une vue en implémentant l'interface **OnMapReadyCallback** qui appelée lorsque la carte est prête à être utilisée. La figure suivante montre la structure établie dans notre programme :

```
public class MapsActivity extends FragmentActivity implements OnMapReadyCallback {
 private GoogleMap mMap;
 LocationManager locationManager;
 @Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_maps);
```

Figure IV.2 : *Structure de l'entête du module localisation*.

En outre, pour pouvoir utiliser les services de la localisation nous avons utilisés les deux classes **GoogleMap**, **LocationManager**, La première sert à offrir le service de la géolocalisation ainsi que la deuxième permet d'accéder aux services de localisation du système. Ces services permettent aux applications d'obtenir des mises à jour périodiques de la situation géographique de l'appareil.

La méthode **onMapReady** est la méthode principale dans laquelle nous faisons l'appel aux services de Google Maps comme le fournisseur de localisation et aussi les services de la téléphonie pour pouvoir afficher quelques paramètres radio au niveau du lieu correspondant.

```
@Override
public void onMapReady(GoogleMap googleMap) {
    SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager()
            .findFragmentById(R.id.map);
    mapFragment.getMapAsync(this);

    TelephonyManager telephonyManager;
    LocationManager locationManager;
    String context = Context.LOCATION_SERVICE;
    locationManager = (LocationManager) getSystemService(context);
    String provider = LocationManager.NETWORK_PROVIDER;
    String context1 = Context.TELEPHONY_SERVICE;
    telephonyManager = (TelephonyManager) getSystemService(context1);
    Location location = locationManager.getLastKnownLocation(provider);
```

Figure IV.3 : *appel aux services de Google Maps et la téléphonie.*

Après l'obtention de la carte et la demande des services de Google Maps et la téléphonie, la deuxième opération est la récupération de la position exacte de l'utilisateur et ça concerne la longitude, la latitude ainsi que l'adresse actuelle et la vérification de l'état de connexion et la surveillance lors la manipulation au niveau de l'activité.

A l'aide des deux fournisseurs de localisation le GPS et le réseau mobile satellitaire nous récupérons la longitude et la latitude du lieu actuel de l'utilisateur.

```
if (location != null) {
    double lat = location.getLatitude();
    double lng = location.getLongitude();
    String latLongString = "Lat:" + lat + "\nLong:" + lng;
```

Figure IV.4 : *Récupération de longitude et latitude en temps réel.*

Nous offrons aussi à l'utilisateur le service de connaître son adresse actuelle pour faire la correspondance avec la longitude et la latitude récupérées précédemment. Pour cela nous l'avons implémenté via une liste au quelle nous lisons les informations concernées.

```
List<Address> addresses = gc.getFromLocation(lat, lng, 10);
StringBuilder sb = new StringBuilder();
if (addresses.size() > 0) {
    Address address = addresses.get(0);
    for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
        sb.append(address.getAddressLine(i)).append("\n");
```

Figure IV.5 : *La liste implémentée pour obtenir l'adresse actuelle.*

Lors l'utilisateur est en mode localisation, l'état de connexion, le changement de position et l'échec de la requête doivent être prises en charge pour cela nous avons ajoutés des méthodes qui interagissent avec les opérations précédentes comme la méthode **onLocationChanged** si l'utilisateur change de position ainsi que d'autres sont décrites dans la figure suivante :

```
LocationListener myLocationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
' Update application based on new location.
    }

    public void onProviderDisabled(String provider) {
' Update application if provider disabled.
    }

    public void onProviderEnabled(String provider) {
' Update application if provider enabled.
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras) {
```

Figure IV.6 : méthodes d'interaction en temps réel avec la position actuelle.

La figure suivante englobe la structure présentée dans la partie qui précède en diagramme de classe UML :

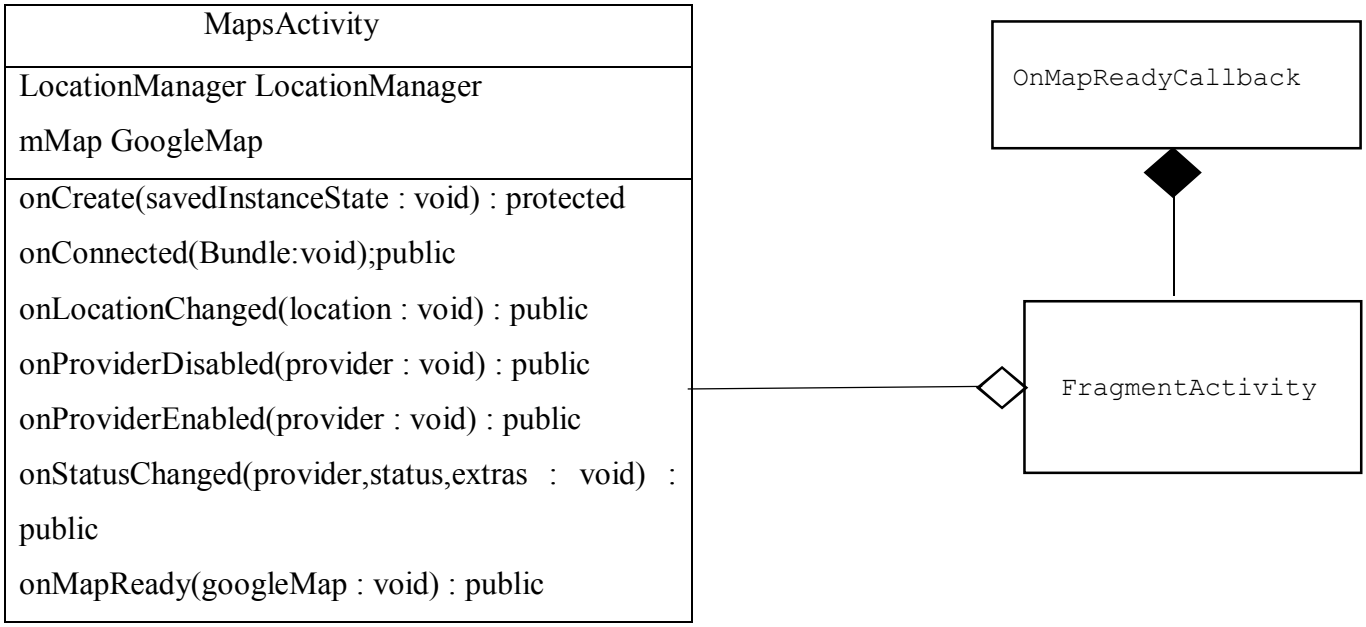

Figure IV.7 : *Diagramme de classe UML du module de la géolocalisation.*

IV.2.2 Détection de la signalisation GSM au niveau de l'Air Interface :

Suite à la partie de la téléphonie et les paramètres de signalisation décrite en troisième chapitre, nous allons présenter l'utilisation de ses APIs, en les exploitant sous forme d'un module GSM divisé en trois parties; une pour les paramètres radio utilisée par l'utilisateur en temps réel, la deuxième vise les informations propres à la carte SIM utilisée et la troisième partie couvre les paramètres de signalisation écoutées par le périphérique en mode de diffusion depuis les stations de base du même opérateur.

IV.2.2.1 Paramètres radio utilisées par l'utilisateur en temps réel :

Une fois l'utilisateur mobile aura l'accès au réseau de l'opérateur, le périphérique va interagir avec la station de base au quelle il est lié, en lui offrant le débit GSM, une puissance de signal pour qu'il puisse envoyer et recevoir aussi que d'autre paramètres d'identification comme sa zone de localisation et la cellule où il se trouve ainsi que l'état du roaming, cette première partie a été mise sous forme d'une classe nommée **GSM** pour que ces paramètres soient visibles par l'utilisateur en écoutant l'AIR interface.

La classe **GSM** étend la classe de base **AppCompativity** qui offre toutes les fonctionnalités pour une activité sous Android. Pour que nous avons eu un accès aux propriétés du réseau les objets **telephonyManager** et **location** propres aux deux classes **TelephonyManager** et **GSMCellLocation** respectivement réalise cette opération en passant par la demande de service de la classe **TelephonyManager** expliquée en troisième partie du chapitre trois, nous

avons implémenté les méthodes présentées dans la même partie pour récupérer ces paramètres, et la figure suivante démontre la structure de la classe **GSM**.

```
public class gsm extends AppCompatActivity {
 //les deux objets des classe TelephonyManager et GSMCellLocation
 TelephonyManager telephonyManager;
 GsmCellLocation location;
 //création de l'activité
 @Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.gsm);
     //Demande de service de la classe  TelephonyManager
     String service = Context.TELEPHONY_SERVICE;
     telephonyManager = (TelephonyManager) getSystemService(service);
     location = (GsmCellLocation) telephonyManager.getCellLocation();
     //différents paramètres avec leurs méthodes de récupération
     int cid = location.getCid(); //ID de la cellule
     int lac = location.getLac();//ID de la zone de localisation
     int psc = location.getPsc();//débit GSM offert
     boolean roaming = false;//état de Roaming
     roaming = telephonyManager.isNetworkRoaming();
```

Figure IV.8 : *Structure de la classe **GSM**.*

Pour la détection de signal reçu en dBm nous avons utilisé la classe **NeighboringCellInfo** sous forme d'une liste via l'objet **telephonyManager**.

```
//la puissance de signal GSM
String dBm=null;//initialisation du signal
//l'utilisation de la classe NeighboringCellInfo
List<NeighboringCellInfo> NeighboringList = telephonyManager.getNeighboringCellInfo();
//Détection du signal
for (int i = 0; i < NeighboringList.size(); i++) {

    int rssi = NeighboringList.get(i).getRssi();
```

Figure IV.9 : *Code utilisé pour obtenir la puissance de signal.*

IV.2.2.2 Caractéristiques de la SIM :

La classe **Sim** a été écrite pour récupérer les informations propres à la carte comme l'IMSI, le numéro de serial, son état actuel ainsi que la technologie utilisée et l'état de données.

Elle étend la même classe de base **AppCompativity** et utilise la **TelephonyManager** pour obtenir ces paramètres.

```
public class ParamSim extends AppCompatActivity { //Extension de la classe de base
    TelephonyManager telephonyManager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.param);
        //Demande de service
        String context = Context.TELEPHONY_SERVICE;
        telephonyManager = (TelephonyManager) getSystemService(context);
        //Parametres de la Sim
        String IMEI = telephonyManager.getDeviceId();
        String IMSI_id = telephonyManager.getSubscriberId();
        String SimSerialNumber = telephonyManager.getSimSerialNumber();
```

Figure IV.10 : *Récupération des différents paramètres SIM.*

Pour l'écoute de l'état de données nous avons utilisé la classe **PhoneStateListener** qui met le périphérique en état d'interception avec la méthode **DataStateListener** et voici une partie de code.

```
//Ecoute de l'état de données
PhoneStateListener dataStateListener = new PhoneStateListener() {
    public void onDataActivity(int direction) {
        String dataActivityStr = "None";
        TextView telechargement = (TextView) findViewById(R.id.up);
        switch (direction) {
            case TelephonyManager.DATA_ACTIVITY_IN:
                dataActivityStr = "Downloading";
                break;
            case TelephonyManager.DATA_ACTIVITY_OUT:
                dataActivityStr = "Uploading";
                break;
```

Figure IV.11 : Interception de l'état de données.

IV.2.2.3 Ecoute des paramètres radio des cellules voisines :

Comme nous avons indiqué dans le premier chapitre le canal BCCH est le canal de diffusion des informations comme la puissance, la zone de localisation et les autres informations citées avant, lors que le périphérique reçoit plusieurs trames BCCH de la part de plusieurs station de base qui sont situés à six canaux dans la norme GSM et dans ce concept cette partie a comme rôle d'extraire ces informations et les mettre en disposition de l'utilisateur.

Ces informations seront affichées dans la première vue lors le lancement de l'application, Elles sont implémentées dans la classe principale **Main** qui étend la même classe de base **AppCompativity** et utilise les services de la **TelphonyManager**. Son principe est presque

Identique à la première partie sauf que dans la liste implémentée à l'aide de la classe **NeighboringCellInfo** nous lions toutes les informations contenues dans cette liste et qui donne les paramètres radio des cellules voisines. Nous avons ajouté la méthode **get(i)** avant chaque récupération d'un paramètre pour que nous obtenions ces données et la figure suivante illustre le code utilisé.

```
//Lecture et Affichage des informations de toutes les cellules reçues
stringNeighboring = stringNeighboring+"\n"
        + String.valueOf(NeighboringList.get(i).getLac()) + "   "
        + String.valueOf(NeighboringList.get(i).getCid()) + "   "
        + String.valueOf(NeighboringList.get(i).getPsc()) + "        "
        + String.valueOf(NeighboringList.get(i).getNetworkType()) + "
        + dBm + "\n";NeighboringCellInfo
```

Figure IV.12 : Interception des informations des cellules voisines.

Le diagramme de classe suivant va schématiser la structure des trois parties :

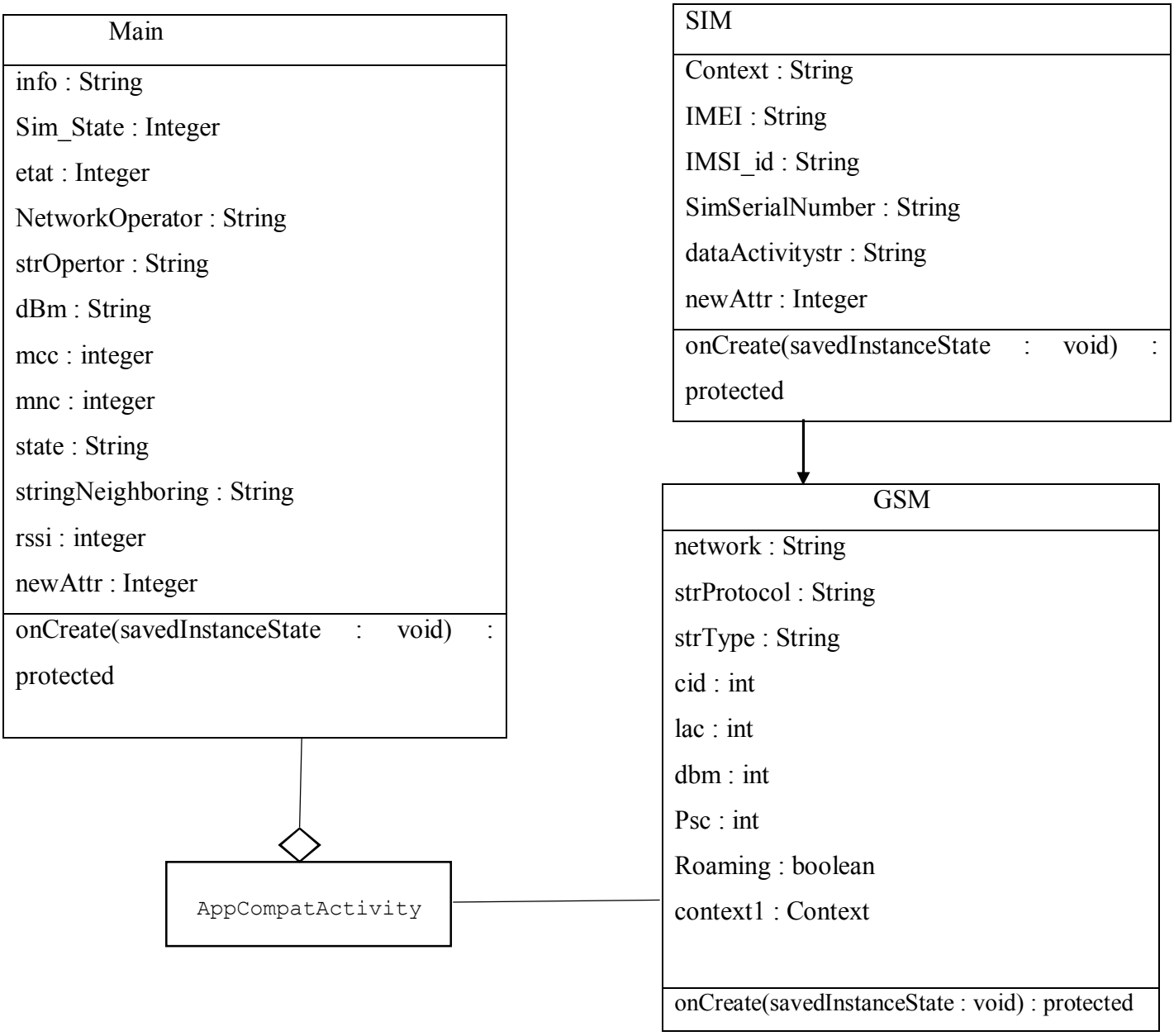

Figure IV.13 : Diagramme de classe pour la détection de signalisation GSM.

IV.2.3 Détection des réseaux wifi en proximité :

Hors la géolocalisation et la signalisation GSM, nous avons proposé un module pour la détection des réseaux wifi en proximité divisé en deux classe ; la première offre à l'utilisateur la possibilité d'analyser son réseau wifi auquel il est connecté en lui fournissant des informations réseaux comme la puissance du signal reçu, la fréquence utilisée ainsi que d'autre paramètres servent à diagnostiquer la liaison wifi courante. La deuxième classe a presque le même fonctionnement de la première car elle prend en charge les mêmes paramètres sauf qu'elle est sous forme d'un scan réseau pour tous les points d'accès disponible en portée.

IV.2.3.1 Structure de la première partie :

D'un point de vue que l'utilisateur a le choix de s'en servir lors la connexion soit par la 3G soit via une connexion wifi, l'idée de cette opération est venue pour qu'il puisse surveiller son réseau auquel il est connecté et évaluer la qualité de signal offerte. Nous offrons cette possibilité par la classe **Wifi**.

Une fois l'utilisateur est en état de connexion via un point d'accès wifi, il a la main d'analyser son réseau en obtenant l'adresse physique et le type de cryptographie du point d'accès, la fréquence actuelle, la puissance de signal reçu, le débit offert et les adresse physique et logique du périphérique.

La classe **Wifi** étend la classe de base **AppCompativty** et utilise les deux classe **ConnectivityManager** et **WifiManager** décrites en troisième chapitre dans la partie gestion du réseau et la connectivité internet en créant les objets Contex1et WifiManager respectivement qui facilitent l'accès aux services et la surveillance du point d'accès.

```
public class Wifi extends AppCompatActivity {
//Demande des services réseau
WifiManager wifiManager; //Pour la surveillance du réseau
ConnectivityManager context1; //Demande de service
WifiInfo connectedInfo; //Information contenues lors la liaison WiFi est établite
```

Figure IV.14 : La classe **Wifi** avec les services utilisés.

Les méthodes implémentées pour rechercher les différents paramètres sont incluses dans la méthode **onCreate** et elles sont représentées dans la figure suivante :

```
//Demande de service à l'aide de la méthode getSystemService
wifiManager = (WifiManager) getSystemService(context_wifi);
context1 = (ConnectivityManager) getSystemService(CONNECTIVITY_SERVICE);
connectedInfo = wifiManager.getConnectionInfo();
//Les paramétres de la connexion WiFi
String ad_Mac = wifiManager.getConnectionInfo().getMacAddress();//adress MAC du périphérique
int puissance_Wifi = connectedInfo.getRssi();//puissance de signal
int ipAddress =connectedInfo.getIpAddress();//adresse ip
//La conversion d'adresse ip en décimale
String decimale_ip =  String . format("%d.%d.%d.%d", (ipAddress & 0xff), (ipAddress >> 8 & 0xff),
        (ipAddress >> 16 & 0xff), (ipAddress >> 24 & 0xff));
int debit = connectedInfo.getLinkSpeed(); //débit de la connexion
int frequence= connectedInfo.getFrequency();//fréquence utilisée
String SSID=connectedInfo.getSSID();//nom du point d'accès
String Mac_Modem=connectedInfo.getBSSID();//adresse MAC du point d'accès
```

Figure IV.15 : Le code utilisé pour la récupération des informations wifi.

IV.2.3.2 Scan des réseaux wifi disponibles :

Comme nous avons expliqué dans le début de cette partie la classe **ScannWifi** se diffère de la classe **Wifi** dans l'analyse de tous les points d'accès vacants. La classe **WifiReceiver** autorise Cette opération après le passage par la demande du service toujours et l'implémentation de ces paramètres dans une liste nommée **ScannResult**.

```
public class ScannWifi extends AppCompatActivity{
    TextView mainText; //texte d'affichage
    WifiManager mainWifi;
    WifiReceiver receiverWifi; //détection des réseaux disponibles
    List<ScanResult> wifiList; //liste contenate tous les informations reçu par le périphérique
    StringBuilder sb = new StringBuilder();
```

Figure IV.16 : entête de la classe **ScannWifi**.

La classe a comme caractéristique de vérifier l'état du wifi et l'activer avant commencer le scan via la méthode **setWifiEnabled**. Nous avons implémenté une autre classe fille **WifiReceiver** avec une seule méthode **onReceive** afin d'extraire les informations concernées depuis la liste **ScannResult** et pouvoir surveiller l'état du scan s'il y a un changement ou un mouvement par l'utilisateur.

L'activité affiché à l'aide de cette classe a le même cycle de vie comme tous les activités sous Android et ceci grâce aux méthodes **onResume** et **onPause**.

```
class WifiReceiver extends BroadcastReceiver {

    // This method call when number of wifi connections changed
    @TargetApi(Build.VERSION_CODES.M)
    public void onReceive(Context c, Intent intent) {

        //le sb est utilisé pour l'affichege des informations
        sb = new StringBuilder();
        wifiList = mainWifi.getScanResults();
        sb.append("\n Number Of Wifi connections: "+wifiList.size()+"\n\n");

        for(int i = 0; i < wifiList.size(); i++){

            sb.append(new Integer(i+1).toString() + ". ");
            sb.append("SSID: "+(wifiList.get(i)).SSID+"\n"+
                    "Mac_modem: "+(wifiList.get(i)).BSSID+"\n"+
```

Figure IV.17 : Utilisation de la classe interne **WifiReceiver** et la lecture de la liste **WifiList**.

La structure des deux parties est représentée dans le diagramme de classe suivant pour que l'explication précédente se schématise :

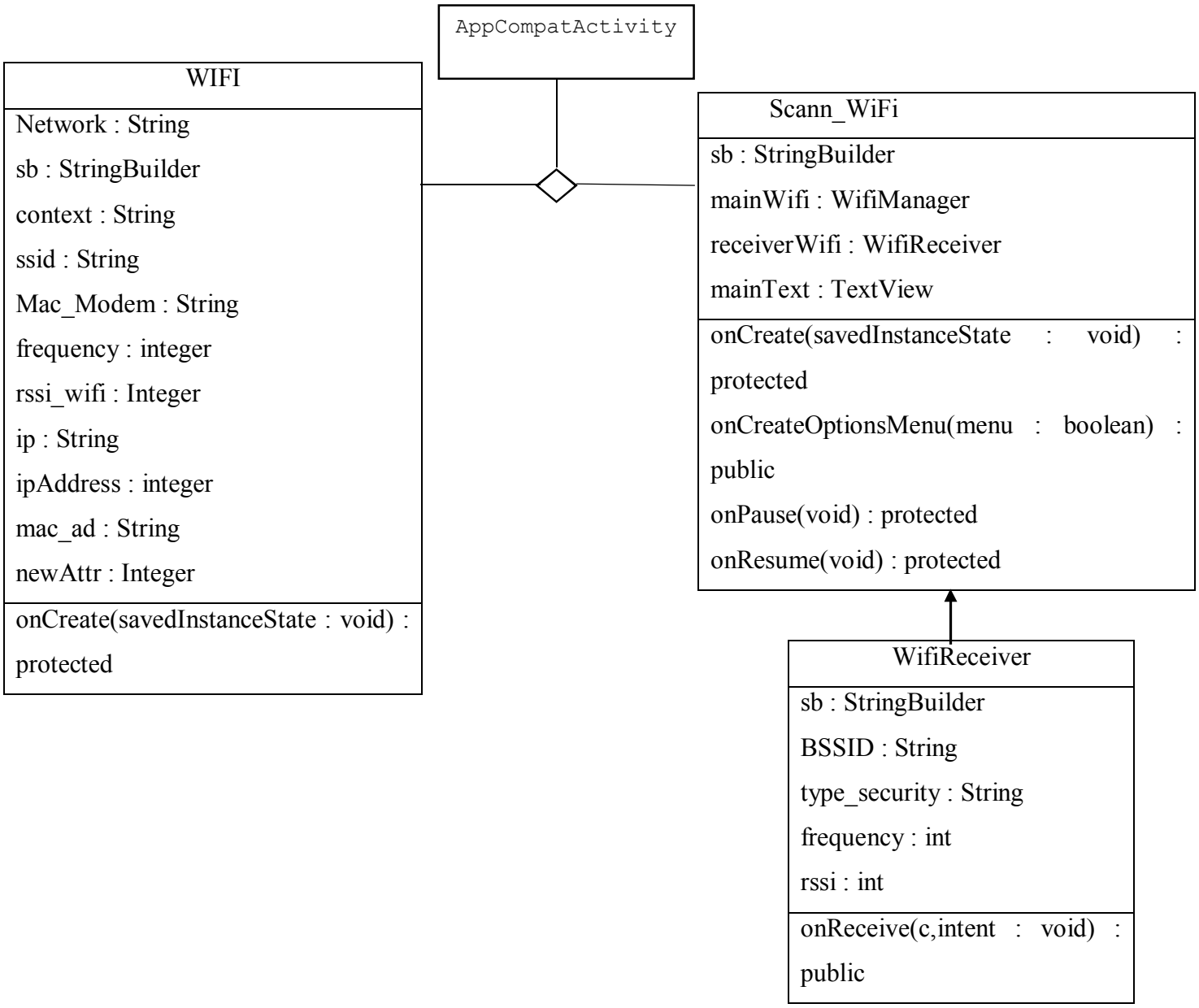

Figure IV.18 : Diagramme de classe du module wifi.

IV.2.4 L'historique des mesures :

Notre vue depuis l'application que nous avons développé est qu'elle sera utilisée par les opérateurs mobiles sous forme d'une approche à la surveillance de leurs réseaux et l'optimisation des mesures. L'historique des mesures s'approche de ce concept car elle est sous forme d'une base de données au niveau de l'application enregistre les mesures effectuées au niveau de plusieurs positions.

Nous avons implémenté la classe **Save** qui prend les mêmes caractéristiques de la première partie dans le module GSM en étendant la classe basique et l'utilisation des services de la classe **TelephonyManager**, Nous sommes intéressé à enregistrer les mesures propres à l'utilisateur ou d'autre façon les paramètres radio situés entre le mobile et la station de base au quelle il est lié.

Les deux méthodes **getAdd** et **Play** réalise ce fonctionnement, la première assure l'accès aux données en les contrôlant lors la mise en état du stockage lors que la deuxième méthode les affiche.

```
public void getAdd(View view) {
    SharedPreferences sharedPreferences = getSharedPreferences("userf", Context.MODE_PRIVATE);
    SharedPreferences.Editor editor = sharedPreferences.edit();
    editor.putString("cid", Cid.getText().toString());
    // editor.putString("puissance", dbm.getText().toString());
    editor.commit();
}
public  void  play(View view) {

    SharedPreferences sharedPreferences = getSharedPreferences("userf", Context.MODE_PRIVATE);
    String name1 = sharedPreferences.getString("cid", "");
    laff.setText(name1);
```

Figure IV.19 : Les deux méthodes responsables sur la réalisation de l'historique.

Nous finissons cette partie par le diagramme de classe qui illustre la composition de la classe Historique.

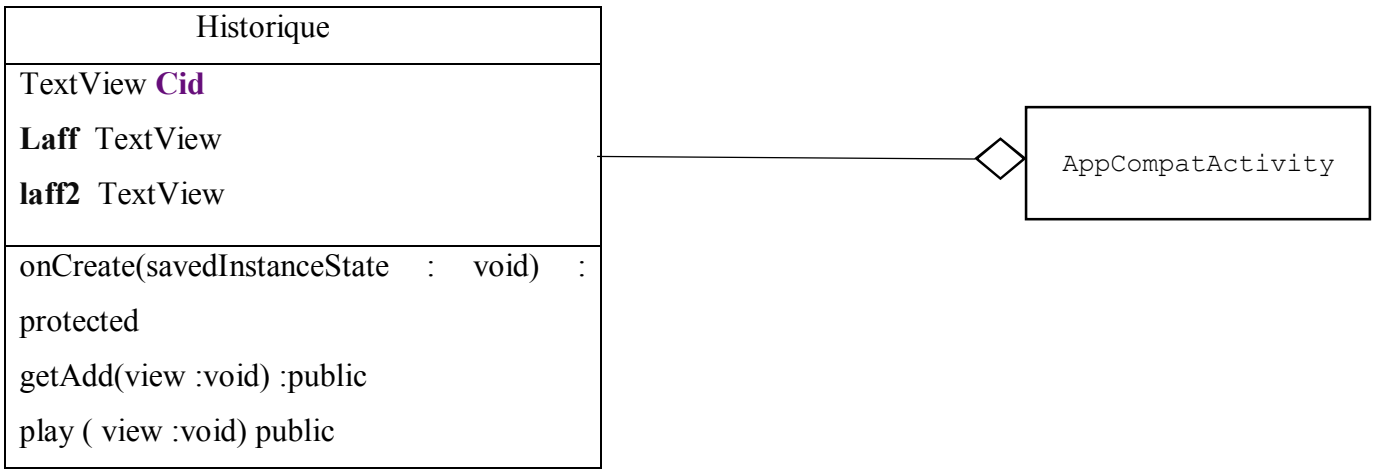

Figure IV.20 : Structure générale de la classe **Save** en diagramme UML.

IV.4 Essai de l'application dans la région de Tlemcen :

Pour que le travail sera valorisé et bien compris, nous avons exécuté notre application dans la région de Souahlia en essayant d'exploiter les différents modules développés.

IV.4.1 La géolocalisation :

Le résultat obtenu est dans la figure suivante, Nous avons localisé la station mobil en récupérant la longitude, la latitude ainsi que l'adresse actuelle, la zone de localisation et la puissance de signal reçu.

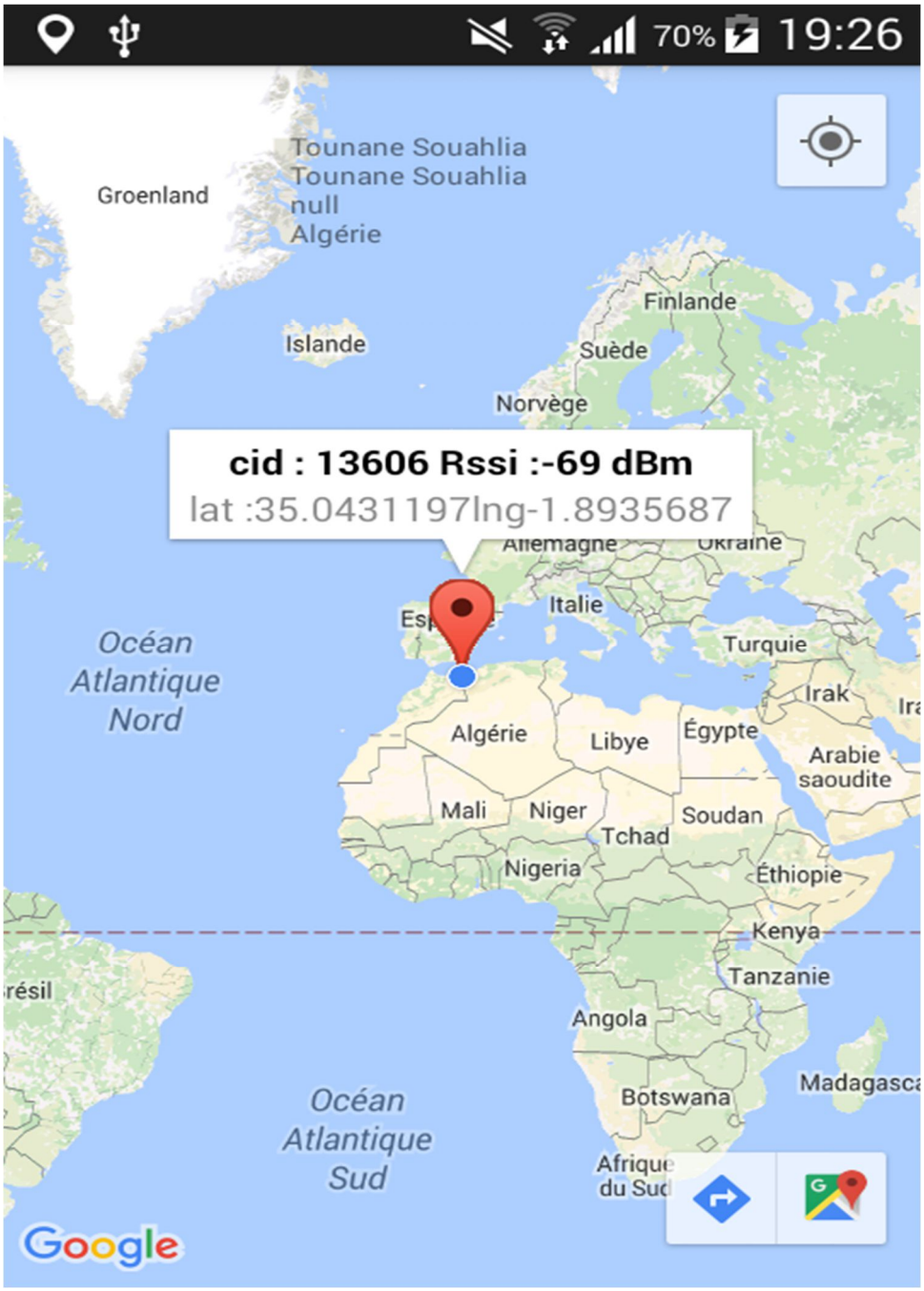

Figure IV.21 : *Résultat de la géolocalisation.*

IV.4.2 Paramètres GSM :

Après le développement de la classe GSM et l'essai nous avons obtenus les résultats suivant dans le même lieu localisé précédemment :

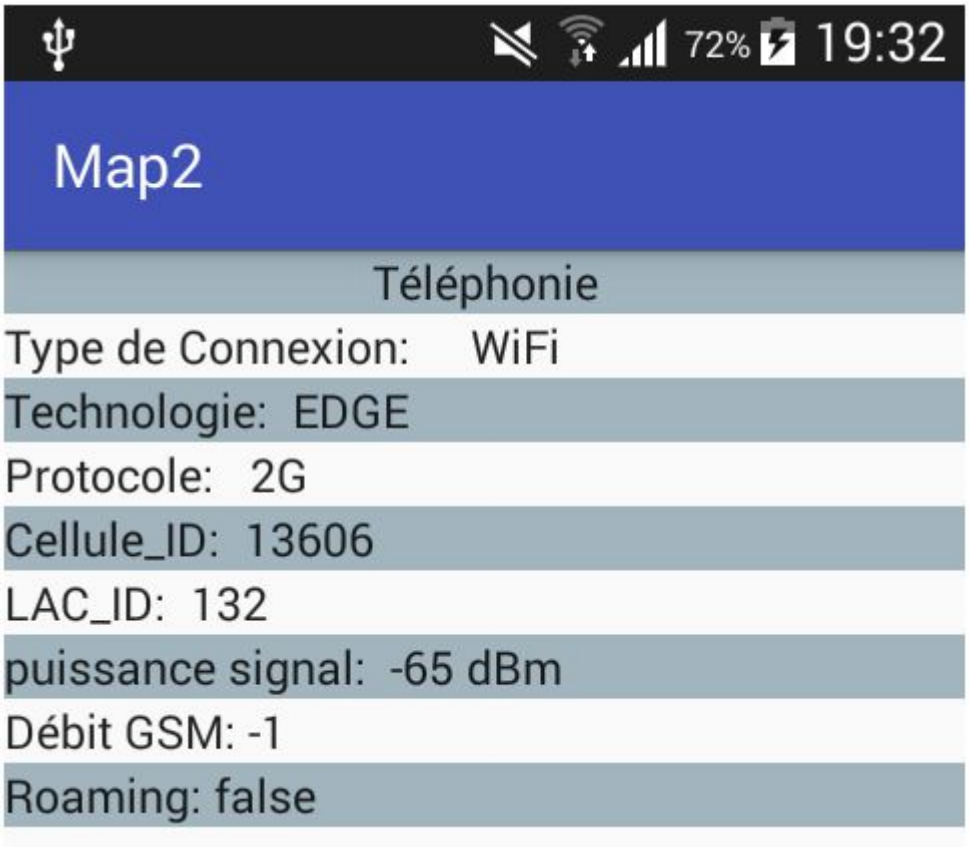

Figure IV.22 : *Paramètres GSM récupérés.*

IV.4.3 Paramètres SIM :

Les trois paramètres de la carte SIM ont été obtenus dans la figure suivante :

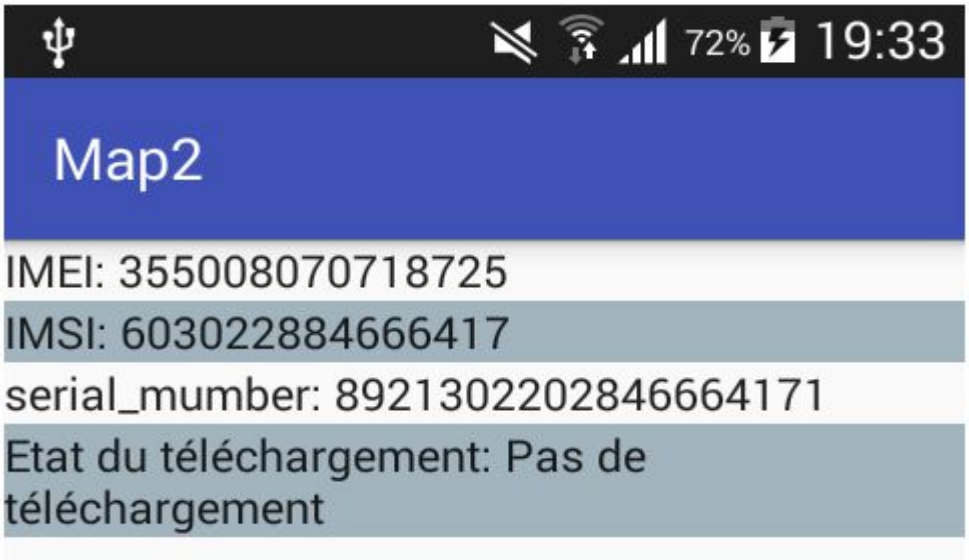

Figure IV.23 : *Paramètres SIM.*

IV.4.3 Paramètres radio des cellules voisinage avec l'historique :

L'opération de l'historique permet d'intercepter les paramètres radio des cellules voisinage avec une possibilité d'enregistrement au niveau de l'application.

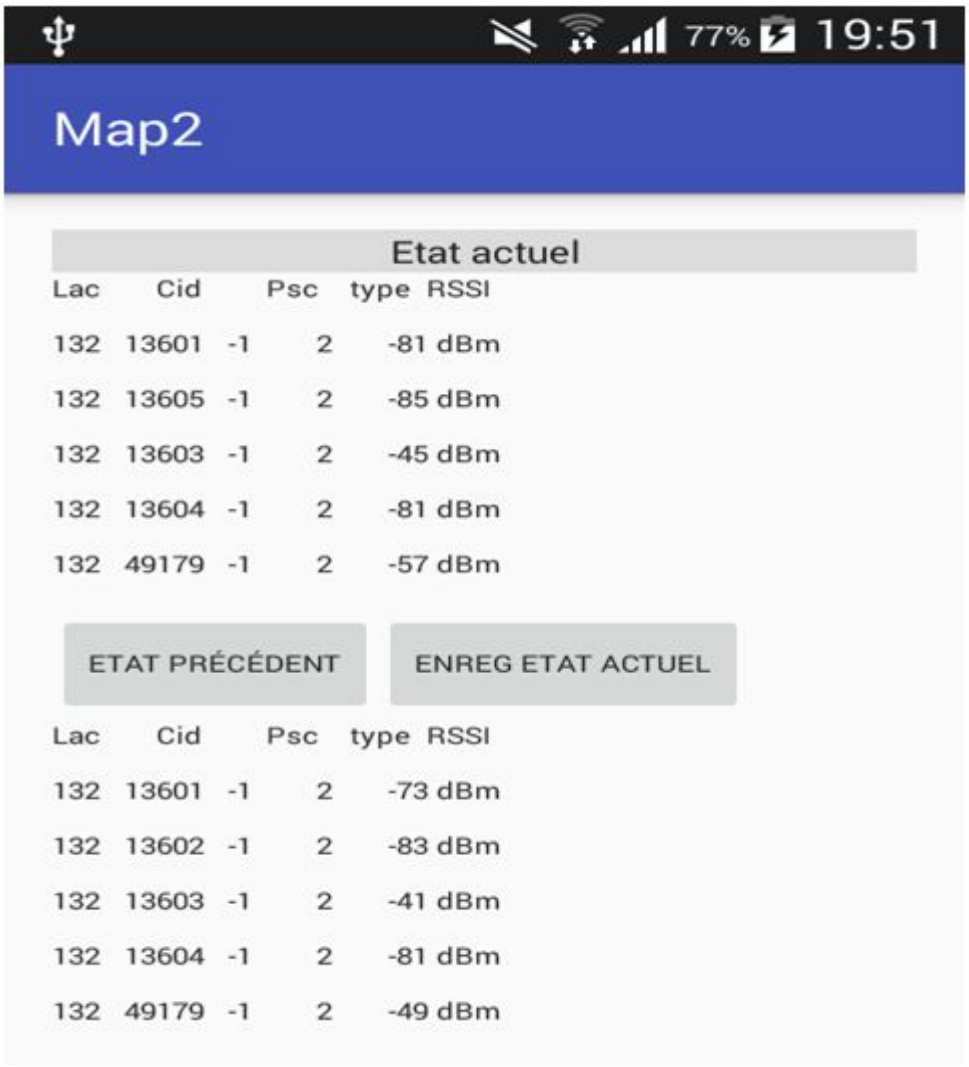

Figure IV.23 : *Paramètres radio des cellules voisinage et l'historique.*

IV.4.3 Module Wifi et l'opération du scan :

Nous avons indiqué avant que le module wifi se compose de deux opérations une pour le point d'accès où l'utilisateur est connecté et l'autre est sous forme d'un scan pour tous les points d'accès disponibles. Les deux figures suivantes illustrent ce fonctionnement.

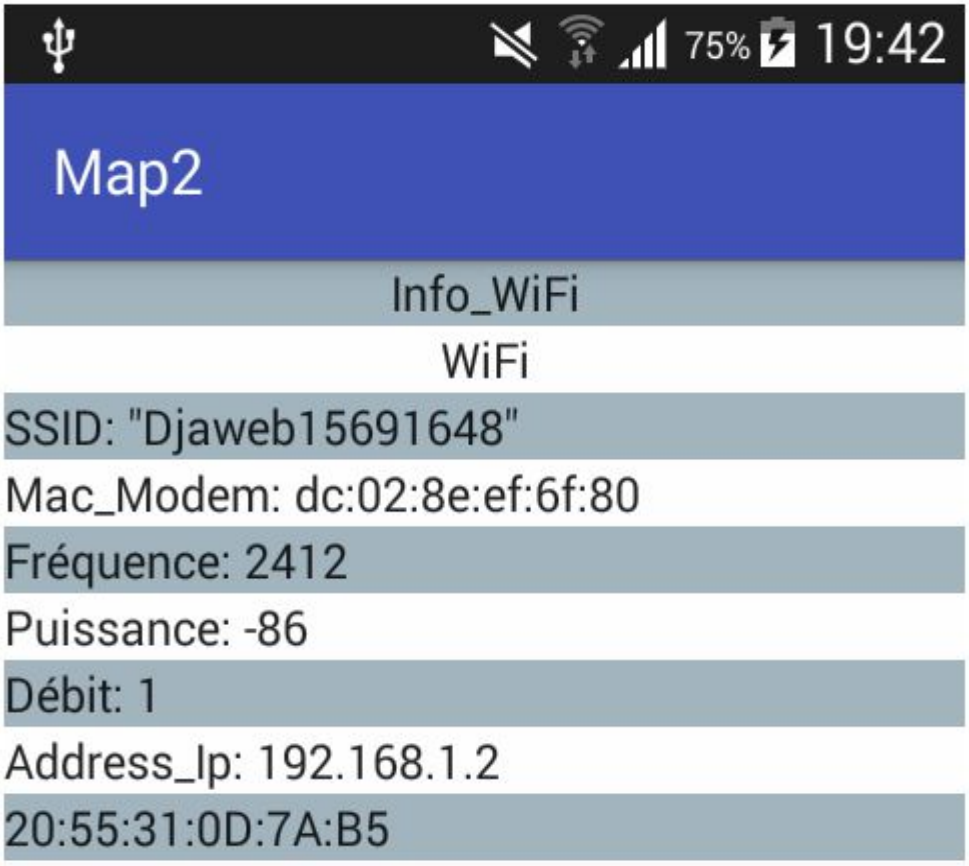

Figure IV.24 : *Analyse du point d'accès où le périphérique est connecté.*

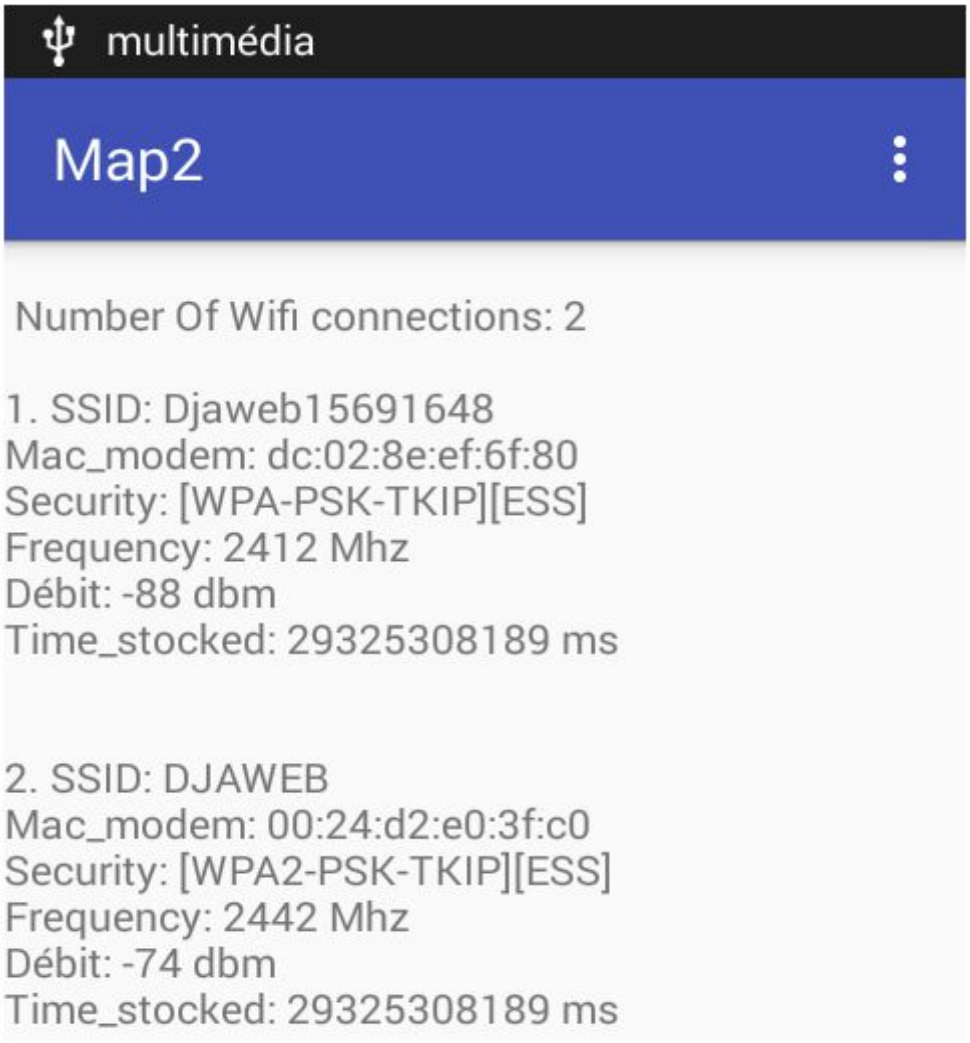

Figure IV.24 : *Résultat du Scan de tous les points d'accès disponibles.*

IV.4 Conclusion :

Dans ce chapitre nous avons présenté le développement de notre application en sa globalité. Le logiciel une fois lancé sur un smartphone a atteint les objectifs convoités. En effet, il nous a été possible de la tester sur plusieurs endroits de la wilaya de Tlemcen. Des information sur les réseaux GSM et Wi-Fi ont été visualisés et analysées.

Contenu