Nyugat-magyarországi Egyetem

Simonyi Károly Műszaki, Faanyagtudományi és Művészeti Kar

Informatikai és Gazdasági Intézet

Mobil alkalmazás fejlesztése erdészeti adatok megjelenítésére

Szalai Mihály

Szakdolgozat

Konzulens: Dr. Horváth Ádám

2016. május 5.

Nyugat-magyarországi Egyetem

Simonyi Károly Műszaki, Faanyagtudományi és Művészeti Kar

H-9401 Sopron, Bajcsy-Zs. u. 4. Pf.: 132.

Tel: +36 (99) 518-101 Fax: +36 (99) 518-259

NYILATKOZAT

Alulírott **Szalai Mihály** (neptun kód: **TI30Q0)** jelen nyilatkozat aláírásával kijelentem, hogy a **Mobil alkalamazás fejlesztése erdészeti adatok megjelenítésére** című

szakdolgozat

(a továbbiakban: dolgozat) **önálló munkám**, a dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. tv. szabályait, különösen a hivatkozások és idézések tekintetében.

Hivatkozások és idézések szabályai:
Az 1999. évi LXXVI. tv. a szerzői jogról 34. § (1) és 36. § (1) első két mondata.)

Kijelentem továbbá, hogy a dolgozat készítése során az önálló munka kitétel tekintetében a konzulenst illetve a feladatot kiadó oktatót **nem tévesztettem meg.**

Jelen nyilatkozat aláírásával tudomásul veszem, hogy amennyiben bizonyítható, hogy a dolgozatot nem magam készítettem, vagy a dolgozattal kapcsolatban szerzői jogsértés ténye merül fel, a Nyugat-magyarországi Egyetem megtagadja a dolgozat befogadását és ellenem fegyelmi eljárást indíthat.

A dolgozat befogadásának megtagadása és a fegyelmi eljárás indítása nem érinti a szerzői jogsértés miatti egyéb (polgári jogi, szabálysértési jogi, büntetőjogi) jogkövetkezményeket.

Sopron, 2016. május 5.

……..

hallgató

Absztrakt

A félév során a feladatom egy olyan androidos mobilalkalmazás elkészítése volt, amely adatbázis kapcsolat segítségével képes erdészeti adatokat megjeleníteni térképes és grafikonos formában.

Az alkalmazáshoz az adatokat az ERTI (Erdészeti Tudományos Intézet) szolgáltatta. A program megjeleníti térképen az ERTI rovarcsapdáit és az OMSZ (Országos Meteorológiai Szolgálat) által szolgáltatott meteorológiai mérések földrajzi pozícióját a térképen. A felhasználó kiválaszthat egyet a csapdák közül a térképen, és lekérdezheti a csapda fogási adatait napi vagy havi bontásban. A felhasználónak a lekérdezéshez egy következő képernyőn meg kell adnia a megjelenítendő fajokat és az idő intervallumot, amire szűrni szeretne. Napi nézet esetén minimális limitet is beállíthat a fogás számra. Az egyes fajok fogási adatait meg lehet jeleníteni egy közös, vagy külön-külön grafikonon is. A felhasználónak lehetősége van havi nézetben egy oszlop kiválasztása után lefúrni, és napi nézetben megnézni az adott faj fogási adatait a kiválasztott hónapban.

Az alkalmazással szemben az is egy elvárás volt, hogy kapcsolódni tudjon az Informatikai és Gazdasági Intézetben futó Sensorhub architektúrához. A Sensorhub kiépítése ebben a félévben volt egyes hallgatótársaimnak a feladata. Mivel nem tudtam a félév elejétől használni ezt az adatforrást, ezért a félév elején elkészítettem egy prototípus verziót, amely adatforrásként helyi fájlokat használ. A félév végére elkészült a Sensorhub az intézetben, így én is el tudtam készíteni az alkalmazás végleges vékony kliens verzióját, amelyik a Sensorhub-ot használja adatforrásként.

A félév során jelentős tapasztalatokra tettem szert az Androidra történő fejlesztés területén, valamint megtanultam grafikonokat készíteni az MPAndroidChart eszköz segítségével. Reményeim szerint sikerült egy hasznos alkalmazást készítenem az ERTI számára.

Abstract

During the semester, my task was to develop an Android mobile application which is able to show forestry data on maps and on charts, too.

The Forest Research Institute (ERTI) gave a set of their data for the application. The program shows insect traps of the ERTI and the position of the data and the meteorological sensors of the Hungarian Meteorological Service (OMSZ) on the map. The user can select one of the traps on the map, and he can query the data of the traps by daily and by monthly view, too. On the next screen, the user has to write the selected species and the time interval for the query for filtering purposes. In case of the daily view, the user can give the lower limit of the catches. The program can draw the data of each species either to the same chart or to different charts, as well. In monthly view, the user can select one of the entries, he can drill down to daily view and he can see all the daily data of the selected month.

There were even an another expectation, to be able to connect to the Sensorhub platform of the Institute of Informatics and Economics. Some of my classmate's theses were to develop the Sensorhub platform during this semester. I had not been able to use this data source at the beginning of the semester, so I developed a prototype version, that uses local files as data source. Eventually, the Sensorhub platform was completed for our Institute in the end of the semester. So I could make the final version of the application as a thin client that uses the Sensorhub as data source.

I got a lot of experience in the topic of Android development during this semester. I have studied to draw charts with the MPAndroidChart API. I hope that I could make a useful application for the ERTI, too.

Mobil alkalmazás fejlesztése erdészeti adatok megjelenítésére

Tartalom

1. Bevezetés

Ebben a félévben a szakdolgozat tantárgy keretén belül egy androidos mobilalkalmazást készítettem el. Az alkalmazás legfőbb célja erdészeti szenzorok megjelenítése térképen, és a szenzorok által szolgáltatott adatok megjelenítése grafikonos formában.

A félév során az Informatikai és Gazdasági Intézetben futó Sensorhub projekt keretében készült el az alkalmazás. A Sensorhub egy speciális adattároló és -lekérdező környezet, a mobilalkalmazás is ezt a környezetet használja adatforrásként. Mivel a környezet most került kialakításra, ezért a félév során fájlban tárolt mintaadatokon dolgoztam, és a szemeszter végén álltam át a Sensorhub környezet felhasználására. A tervezés és a kivitelezés során ezt mindvégig szem előtt tartottam.

A feldolgozandó adatokat az ERTI (Erdészeti Tudományos Intézet) tette elérhetővé a projekt számára. Az ERTI 1965-től kezdve napjainkig az ország különböző pontjain elhelyezett rovarcsapdák fogási adatait feljegyzi és eltárolja. Ezek a fogási adatok jelentik az én programom egyik fontos adatforrását is. A másik fontos adatsor az Országos Meteorológiai Szolgálat által mért meteorológiai adatsor.

Tanulmányaim során az előző félévekben az Önálló laboratórium 1 és Önálló laboratórium 2 tárgyak keretében megismerkedtem az Android operációs rendszerrel és az erre a platformra történő programozással. Ezért döntöttem úgy, hogy ebben a félévben is hasonló jellegű témával szeretnék foglalkozni. Jelen alkalmazás különböző szenzoradatok megjelenítésére és elemzésére koncentrál. Az egyik fő funkció, hogy a szenzorokat térben megjelenítse a felhasználó felé. Ehhez a program az Android beépített térképkezelő könyvtárát, a Google Maps szolgáltatásait használja. A másik fő funkcionalitás a szenzorok által mért adatok megjelenítése, akár nyers formában, akár különböző aggregációs vagy szűréses megoldások alkalmazása után. A grafikonok rajzolásához többfajta lehetőség létezik az Android platform alatt, én ezek közül az *MPAndroidChart* nevű eszközkönyvtárat használtam fel.

A félév során több fejlesztői környezetet, és szoftveres technológiát is felhasználtam. A legfontosabb az Androidra történő fejlesztést támogató Android Studio és a többek között Android programozásra alkalmas Java nyelv volt. A nem Android-specifikus osztályokat (például adatbetöltők vagy entitás osztályok) különböző Netbeans vagy Eclipse projektekben hoztam létre, és csak tesztelés után emeltem be az androidos környezetbe. A Sensorhub architektúra felhasználásának első szakaszában egy a saját gépemen futó Glassfish Java webszervert is használtam. A Sensorhubbal történő HTTP kommunikáció alapjait JSON objektumok jelentették a projekt során. Ezekből hoztam létre a letöltés után entitás példányokat, hogy a többi osztály is fel tudja őket használni. Mivel a Sensorhub a fejlesztés első szakaszában nem volt érhető az intézeti hálózaton kívülről, ezért a félév során az OpenVPN szoftvert is használtam a belső hálózaton kívülről történő elérés céljából.

A félév során elkészült program képes fájl, vagy Sensorhub adatforrásból adatokat feldolgozni és megjeleníteni. Térképen megjeleníti a meteorológiai adatok forráshelyét, illetve a rovarcsapdák pozícióját. A felhasználó számára lehetőséget biztosít a rovarcsapdák fogás adatait napi szinten lekérni, vagy havi összegzés létrehozására a kiválasztott fajok és időszakok

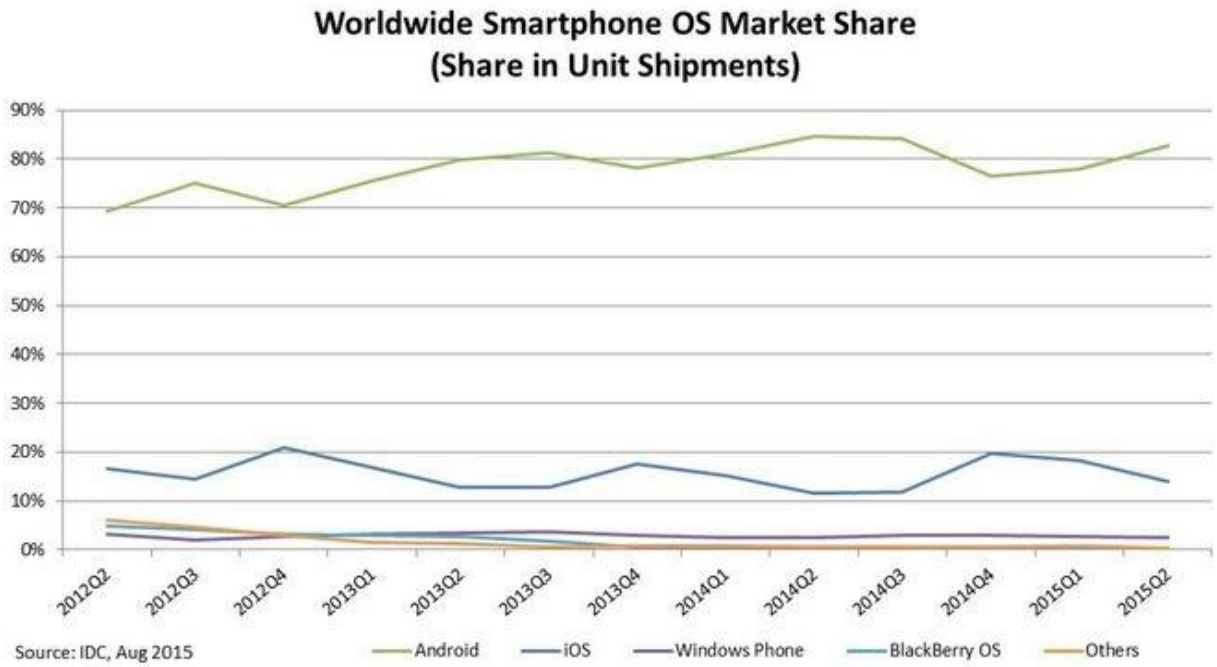
mentén. Jelen projektben csak kétféle szenzor szerepel, de a program támogatja további szenzortípusok definiálását és felhasználását is.
A dolgozatomban írni fogok az Android operációs rendszerről és annak programozásáról. Egy fejezetben bemutatom a felhasznált adattároló megoldást a Sensorhub-ot. Majd teszek egy kis kitekintést, és ismertetek hasonló célra készült mobilalkalmazásokat. Az utána jövő fejezetekben bemutatom a féléves munkavégzésem, az elkészült prototípus illetve vékonykliens alkalmazást. Végül ismertetem a munkám továbblépési lehetőségeit.

2. Az Android rendszerről, és az Android programozásról általában

A félév során a legtöbbet az Android rendszerre fejlesztettem. Ebben a fejezetben szeretném ismertetni az Androidot mint operációs rendszert és mobil platformot, illetve szeretnék kitérni az Android alapú szoftverfejlesztés alapjaira is.

2.1. Az Android operációs rendszer

Az Android jelenleg a legelterjedtebb mobil operációs rendszer a világon, ezért is választottam többek között ezt a platformot a fejlesztéseim megvalósításához. Az Android története 2008-ig nyúlik vissza, ekkor került piacra az első androidos mobilkészülék. Az Android rendszert a Google fejleszti. Az utóbbi években a legtöbb mobiltelefonon ez az operációs rendszer futott, mint ahogy az 1. ábra mutatja.



1. ábra - Az Android piaci részesedése
[forrás: http://www.idc.com/prodserv/smartphone-os-market-share.jsp]

Az Android operációs rendszer egy nyílt forráskódú, szabadon módosítható Linux alapú rendszer. Alkalmazások Java nyelven készíthetőek rá, de ez nem azt jelenti, hogy egy az egyben futna Androidon egy asztali Java program. A platform biztosít natív programozói interfészt is a programozók számára, ez a támogatás a C, illetve a C++ nyelvekkel történik. A rendszer magja, és több rendszer közeli eszközkönyvtár is ezen a nyelven van implementálva. Az Android rendszer több rétegből épül fel, ezek a következők:

- Linux-kernel;
- Rendszerkönyvtárak - ide ékelődik be a Runtime modul;
- Alkalmazás keretrendszer;
- Alkalmazások.

A legalsó szinten egy Linux-kernel helyezkedik el, mivel az Android egy Linux alapú operációs rendszer. Ezen a szinten találhatóak a hardverek eszközkezelői. Ilyenek például a képernyő- és hangkártya-vezérlők, a GPS eszközkezelő, az USB illesztőprogram, vagy az energia menedzser.

A következő szinten találhatóak az úgynevezett rendszerkönyvtárak, amelyek C, illetve C++ nyelven vannak implementálva. Ilyen eszközkönyvtár például az SQLite adatbázis-kezelő, az SSL, vagy az OpenGL 3D grafikai könyvtár. Ebbe a rétegbe ékelődik az Android Runtime modul. Ebben fut a Dalvik nevű virtuális gép, amelyen az Android alkalmazások futnak. A Dalvik nagyban különbözik a Sun Microsystems által kiadott JVM-től. A .class kiterjesztés helyett annál tömörebb .dex kiterjesztésű fájlokat futtat, és utasításkészlete is eltér a JVM-étől. Az Android 4.4-es KitKat verzióban elkezdődött a Dalvik virtuális gép lecserélése, egy nála gyorsabb ART nevű új megoldásra. A KitKat-ben még csak választható volt az ART, de az 5.0-s Lollipop óta ez az alapértelmezett virtuális gép a Dalvik helyett. Az alkalmazások közül mindegyiknek van egy saját virtuális gép példánya, mindegyik önálló Linux felhasználóként fut a rendszerben. Korlátolt CPU és memória áll az alkalmazásoknak a rendelkezésére, az operációs rendszer osztja ki a különböző alkalmazások között a fizikai erőforrásokat.

A következő rétegben találhatóak az alkalmazás keretrendszerek. Ez a keretrendszer adja a felhasználó számára érzékelhető Android operációs rendszert. A réteg feladata az alkalmazások kiszolgálása, és hozzáférés biztosítása a rendszer különböző erőforrásaihoz. Ez a réteg a felel a felhasználói alkalmazások kezeléséért is, és elfedi a felhasználó elől a virtuális gépet, és a Linux rendszert. Ez a réteg már Java nyelven íródott.

A legfelső réteg a felhasználói alkalmazásokat tartalmazza, ilyenek például a telefon vagy az email alkalmazás, vagy bármilyen androidos játék szoftver. A félév során elvégzett munkám nagyjából erre a rétegre korlátozódott, legfeljebb szolgáltatásokat hívtam meg az alkalmazás keretrendszertől. Az Android architektúráját a 2. ábra mutatja be.

2. ábra - Az Android architektúrája [forrás: https://source.android.com/security]

Az Android rendszer elfedi a programozó elől a hardver paramétereit, a programozónak nem kell a programlogika szintjén foglalkoznia például a képernyő méretével, vagy pixelsűrűségével. Az Android rendszer a minősített képernyő erőforrásokon keresztül biztosít lehetőséget a képernyőméret kezelésére. Négyfajta képernyőméret kategória van definiálva az Android rendszerben [1]:

- small;
- normal;
- large;
- xlarge.

A rendszer mindig futásidőben lekéri a készülék adatait, és kiválasztja a megfelelő, vagy legmegfelelőbb képernyő-erőforrást az éppen aktuális készülékre. Ez a folyamat a háttérben történik, a programozónak nem kell foglalkoznia vele, csak biztosítania kell a különböző méretű erőforrásokat.

Hasonló elven támogatja a különböző képernyősűrűségeket is az Android rendszer. Erre azért van szükség, mert nem ugyanúgy jelennének meg a képernyő erőforrások akkor, ha egy inch szélességen sok pixel található a készüléken, mintha kevesebb lenne elérhető. Ennek a megoldására vezették be a dinamikus pixel fogalmát, így ugyanolyan képarányban fog megjelenni egy pixel egy nagy, és egy kis sűrűségű képernyőn is. Egy dinamikus pixel egy 160 dpi-s telefon egy valós pixelének felel meg. Képernyősűrűség tekintetében hat kategória létezik az Androidban:

- ldpi;
- mdpi;
- hdpi;
- xhdpi;
- xxhdpi
- xxxhdpi.

Az Android rendszer hasonló minősítők segítségével tudja támogatni például az álló, és a fekvő (land) képernyőket is. A szöveges erőforrásokat kezelő *string.xml* fájlokból is hozhatunk létre különböző nyelvekkel minősítetteket, így a többnyelvűség is könnyen támogatható. Az Android további nagy előnye, hogy egy bizonyos verzióra megírt alkalmazás futni fog az összes újabb verzión is.

2.2. Az Android rendszer programozása

Az Android rendszerre történő alkalmazásfejlesztés Java nyelven történik, ennek köszönhetően nem Android-specifikus funkciókat, mint például fájlok olvasásáért, háttérszámítások elvégzéséért, vagy entitások megjelenítéséért felelő osztályokat meg lehet írni bármilyen más Java IDE segítségével, akár Netbeans vagy Eclipse projekt keretében belül. Az Android természetesen definiál saját osztálycsomagokat is, a képernyő kezelésére, vagy bármilyen más Android erőforrás kezelésére, például GPS kezelésre vagy WiFi-kapcsolat használatára.

Az Android alkalmazás alapvetően négy komponensből áll [1]. Ezek különböző feladatokat látnak el, különböző életciklus modellel rendelkeznek. A négy fő komponens a következő:

- *Activity;*
- *Service;*
- *ContentProvider;*
- *BroadcastReceiver.*

Az alkalmazás ezekből a komponensekből tetszőleges számút tartalmazhat (az *Activity-t* leszámítva akár nullát is). Egy Android projekt természetesen ezeken kívül tartalmazhat Android-specifikus és nem Androidra jellemző osztályt is. A következőkben, pár sorban a négy komponenst ismertetem.

Az *Activity* a legfontosabb androidos alkalmazás komponens. Az *Activity-k* valósítják meg az Android alkalmazások felhasználói felületét. Természetesen egy alkalmazás több *Activity-t* is tartalmazhat, melyek különböző képernyőfelületeket írnak le, különböző funkciókat valósítanak meg.

A *Service* a háttérben futó, hosszabb ideig működő szolgáltatásokat megvalósító alkalmazáskomponens. A *Service* osztálynak nincs felhasználói felülete, viszont indíthat Activity-ket, vagy dobhat fel ablakokat PopUp vagy Notification formájában. A rendszer több háttérszolgáltatást is futtat folyamatosan, amelyek a rendszer megfelelő működését biztosítják. Service-szel általában helymeghatározást, média lejátszást, vagy nagyobb hálózati forgalmat igénylő folyamatokat lehet elvégeztetni.

A *ContentProvider* tartalomszolgáltató komponensként működik. Adatforrások kezelése és adatlekérdezések kiszolgálása a fő feladata. Célja az, hogy az adatforrás fajtáját elfedje az őt hívó szolgáltatások elől.

A *BroadcastReceiver* különböző eseményekre aktiválódik, és valamilyen feladatot hajt végre annak hatására. Az Android rendszer megvizsgálja egy broadcast esemény érkezésekor, hogy melyik alkalmazásban melyik *BroadcastReceiver* tudja lekezelni az eseményt és elindítja azt. A komponensben történő események implementálása már a programozó feladata. *BroadcastReceiver* használatára jó példa, hogy e-mail érkezésekor a telefon feldob egy ablakot.

2.3. Az Android Studio projekt felépítése

Az Android komponensek leírása után szeretném az Android projekt felépítését is ismertetni. Egy Android Studio projekt több mappát tartalmaz. Részleteibe menően nem ismertetem a mappaszerkezetet, mivel nagyon sok generált fájl található benne, ezek nagy része a fordításért felel, illetve *build* és *make* jellegű funkciókat lát el. Ezeken kívül a telefonra telepítést lehetővé tevő .apk fájlok is találhatóak a projekt mappáiban. A fejlesztés szempontjából az */app/src/main* mappa bír nagyobb jelentőséggel. Fontos még a */app/libs* mappa: ide lehet telepíteni a külső jar függőségeket. Ebben a projektben egy *MPAndroidChart* (*mpandroidchartlibrary-2-1-6*) grafikonrajzoló eszközkönyvtár található itt, illetve egy a Sensorhub-hoz való csatlakozást elősegítő *QueryApi.jar*, amelyet hallgatótársam, Pintér Tamás fejlesztett. További fontos állomány az *R.java,* amely az XML erőforrásokat teszi elérhetővé a Java osztályok számára *Integer* konstansokon keresztül. A fordító generálja az osztályt, nem kell a programozónak foglalkoznia vele, sőt írásvédett fájlról van szó.

A programozó számára */app/src/main* mappa bír a legtöbb jelentőséggel. Itt található a */java* mappa, ebben vannak a programozó által írt osztályok. Itt természetesen több csomagot is definiálhat a fejlesztő. A */res* mappában találhatóak a különböző erőforrás fájlok. A *layout* mappákban találhatók a képernyők felületét leíró XML fájlok. A *drawable* és a *mipmap* mappában találhatók a különböző kép erőforrások, utóbbi mappában az alkalmazás ikonja, melyet az alkalmazások listájában látunk a telefonunkon. A *raw* mappában találhatóak a különböző nyers fájl erőforrások. A program prototípus változata az ebben a mappában lévő fájlokból tölti be az entitásokat az alkalmazásba. A *values* mappa különböző XML fájlokat tartalmaz, amelyek az egész alkalmazás felhasználói felületét meghatározzák. Ezek a kinézetért felelő fájlok a *colors.xml*, a *dimens.xml,* amely a margók méretét írja le, a *styles.xml* felel az alkalmazás grafikai stílusáért, a különböző nyelvekkel minősített *strings.xml* fájlok segítségével pedig (*strings-hu.xml, strings-en.xml*) az alkalmazást többnyelvűvé tudjuk tenni.

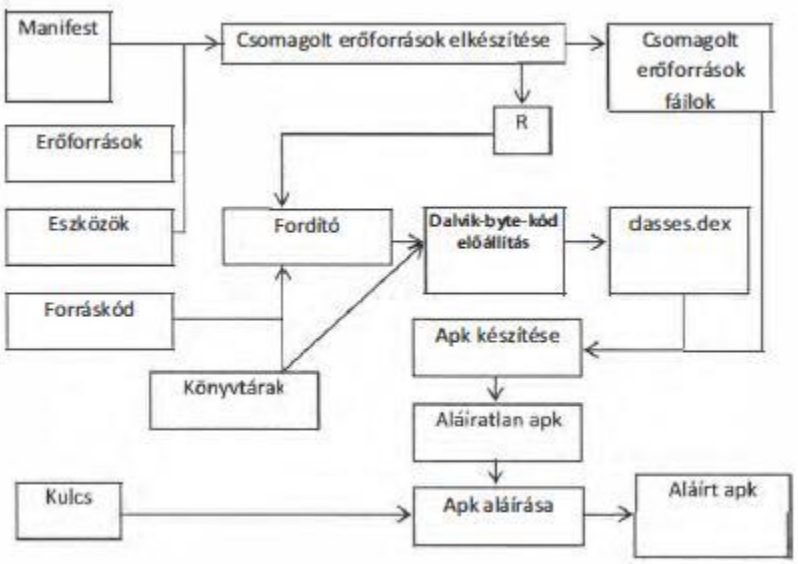
Ebben a mappában található még az *AndroidManifest.xml* állomány. Ez a fájl tartalmazza az alkalmazásban lévő csomagokat, leírja az alkalmazás komponenseit, illetve az alkalmazás futása számára szükséges engedélyeket. Ebben az alkalmazásban az internet használatára,

illetve a pozíció meghatározására kérünk engedélyt. A manifest állományban van regisztrálva az összes Activity, amelyet a program használ. A *MainActivity*-nél található egy *intent-filter*, amiben jelezve van, hogy ez az alkalmazás belépési pontja, ezt kell elindítani, ha az alkalmazást elindítja a felhasználó.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="szalaimihaly.hu.ertidataviewer">
    <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <application
        android:name=".dataloader.App"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".activities.MainActivity"
            android:configChanges="orientation">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
```

2.4. Az .apk állomány

Az andoidos alkalmazásokat egy *.apk* állomány telefonra történő másolásával lehet telepíteni a készülékre. Fejlesztés és hibakeresés közben is egy aláíratlan *.apk* fájl kerül át a készülékre, illetve a Google Play áruházból is ilyen fájlokat töltünk le, és ezekből települ fel az alkalmazás a készülékre. Az *.apk* fájl a manifest állomány, az XML erőforrások, illetve a forráskód alapján készül el. Az *.apk* készítése után egy hitelesítetlen, aláíratlan fájlról beszélünk. Ezt egy *.jks* kiterjesztésű fájl segítségével aláírhatjuk, ezáltal hitelesíteni tudjuk a munkánkat. A Google Play-re történő alkalmazás-feltöltéshez is aláírt *.apk* fájl szükséges. Az .apk fájl generálásának folyamata a 3. ábrán látható.



3. ábra - Az .apk fájl generálásának folyamata [1]

3. A Sensorhub architektúra

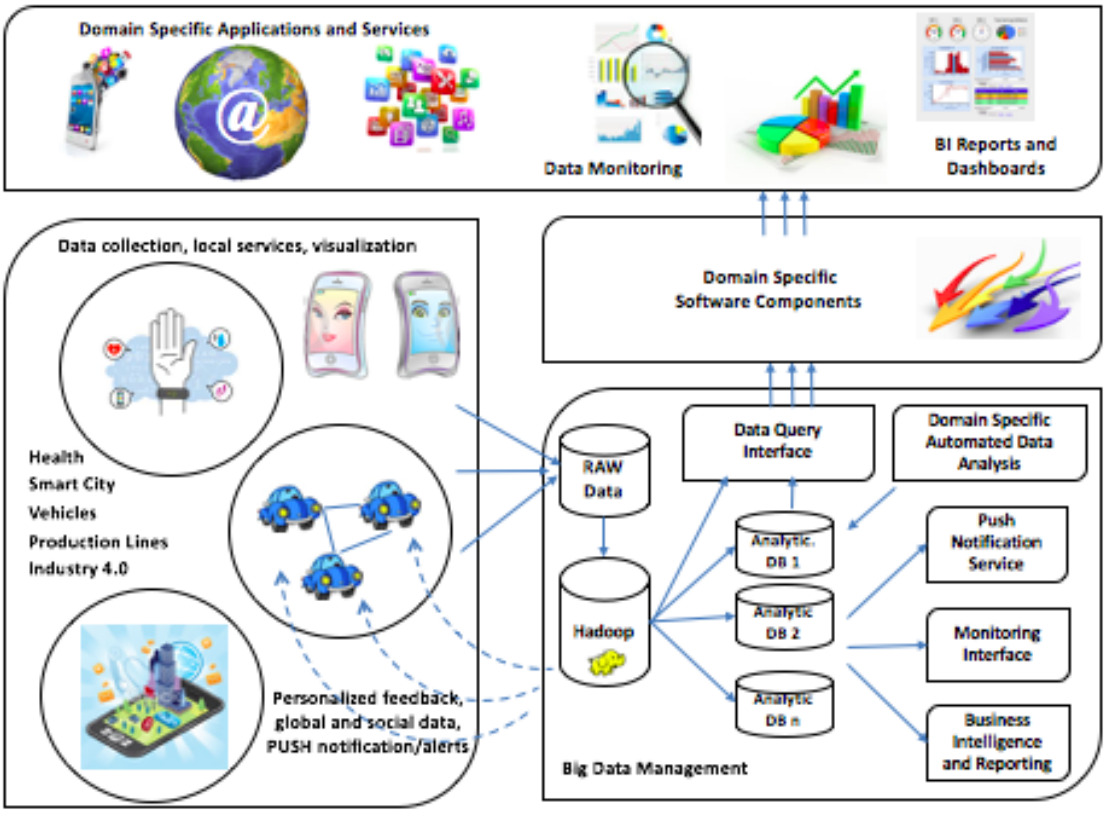
A munkám során, a félév elején helyi szövegfájlokat használtam az alkalmazás adatforrásaként az alkalmazás prototípus verziójában. A szakdolgozat célkitűzésében egy vékonykliens alkalmazás elkészítése volt megfogalmazva, amely az intézeti Sensorhub architektúrát tudja lekérdezni. Mivel a Sensorhub csak a félév végére készült el, ekkorra lett feltöltve adatokkal, és vált lekérdezhetővé, ezért csak ekkor kezdtem el ezt használni adatforrásként az alkalmazás teljes verziójában. Ebben a fejezetben szeretném bemutatni a Sensorhub architektúrát, és bemutatni, hogy a saját munkám hogyan kapcsolódik hozzá.

A Sensorhub az Internet of Things (IoT) koncepció köré épül [2], amely a hétköznapi dolgainkból hoz létre egy információs hálózatot. Ezeket az eszközöket vezetéknélküli hálózat köti össze egymással. A szenzorok által mért adatok a nagyméretű adattárolást lehetővé tevő adattárházakba kerülnek betöltésre. A szenzorok több fajta adatot mérhetnek, ezek legtöbb esetben valamilyen GPS pozícióhoz köthető entitások. Az én alkalmazásomban meteorológia szenzorokról és rovarcsapdákról van szó. Ezeket a *Sensorplace* osztály testesíti meg a programban, az általuk mért adatokat pedig az *ObservedObject* osztály. A program képes bármilyen szenzort kezelni, a magas absztrakciós szintnek köszönhetően. Mind az én alkalmazásom számára, mind a Sensorhub felől nézve bármi szenzornak tekinthető, amely képes a környezetéről adatokat mérni, és ezeket az adatokat képesek közvetíteni környezetük felé [3].

A Sensorhub lehetőséget biztosít domain-specifikus alkalmazások készítéséhez. Itt olyan alkalmazásokról van szó, amelyek egy szenzorhálózaton keresztül adatokat gyűjtenek, és az adattárolást felhő alapokon valósítják meg.

A Sensorhub architektúráját az 4. ábra mutatja be, a következő részekből áll:

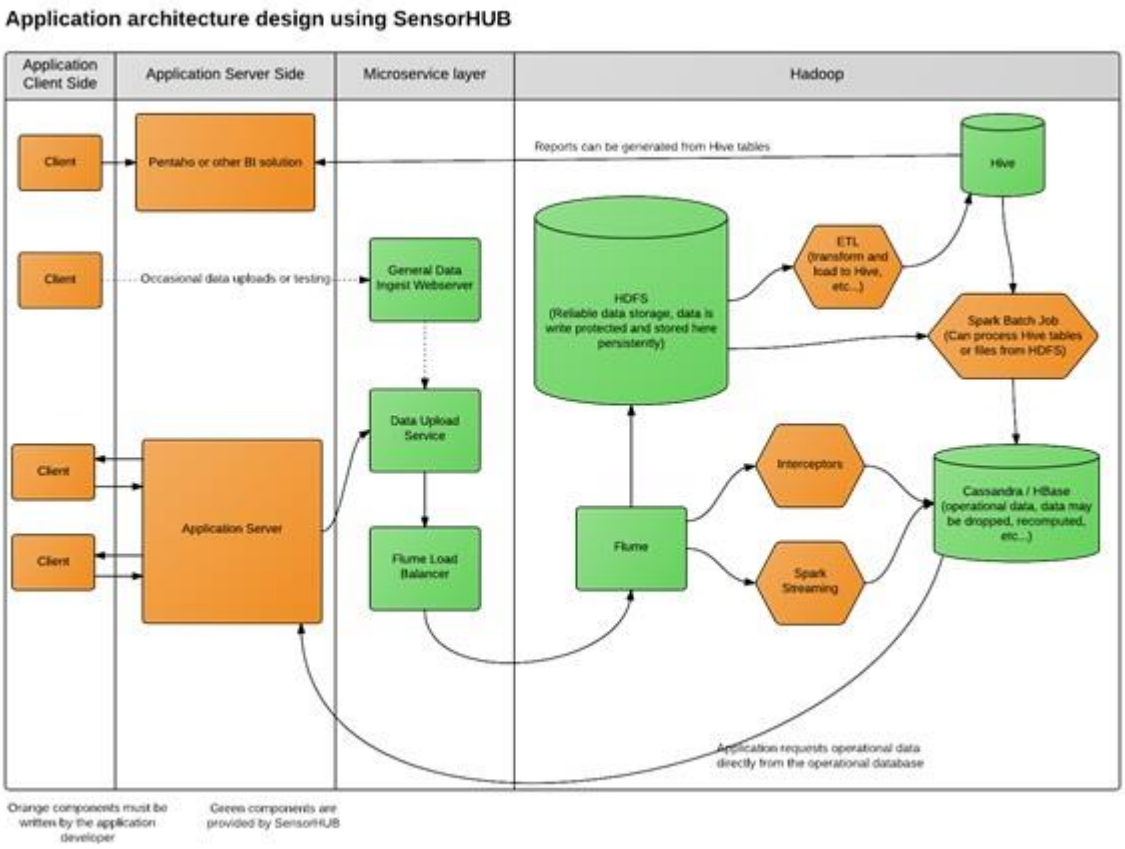
- Szenzorok, adatgyűjtés, helyi számítások, kliensoldali megjelenítés és adatátvitel (balra lenn)
- Felhőalapú kiszolgáló réteg big data adattárolással, adatelemző és -menedzselő szolgáltatások megvalósításával (jobbra lenn)
- Alkalmazásfüggő szoftverkomponensek (középen)
- Alkalmazások, szolgáltatások, üzleti intelligencia (felül)



4. ábra - A Sensorhub architektúrája [2]

A szenzor adatokat, melyeket feldolgoztam, az ERTI szolgáltatta a projekt számára. Ezek meteorológiai és rovarfogási adatok. A Sensorhub képes valós idejű adatbetöltésre is, de ebben a félévben még kézi betöltéssel történt az adatok betöltése a Raw Data interfészen keresztül. Az adattárolást az intézetben futó Hadoop klaszter végzi el, amely elosztottan, több adatbázisban teszi lehetővé az adatok elhelyezését. A Hadoop mögött futó Cassandra adatbázisok sémamentesek, NoSQL alapúak. Ez a szemlélet gyorsabb adatelérést, és lekéréseket tesz lehetővé nagyobb adatmennyiségen is, mint a hagyományos SQL alapú adatbázisok. Ez adja a Sensorhub architektúra egyik nagy erősségét. A középső rétegbe tartozik a Pintér Tamás által írt *QueryApi.jar* ennek segítségével tud a mobilalkalmazás a Sensorhub-hoz csatlakozni. Az API elfedi a fejlesztő elől a Sensorhub felépítését, így a munkám során nem kellett vele foglalkoznom, hanem csak az onnan érkező JSON objektumokat kellett feldolgoznom, és megjelenítenem. Ennek köszönhetően az én munkám a legfelső, alkalmazási rétegben zajlott le, teljesen függetlenül a Sensorhub architektúrától.

A 5. ábrán a Sensorhub architektúrája látható egy másik megközelítésben. A narancssárga színű komponenseket kell a programozóknak testre szabniuk, a zöld színűek pedig az alap működését adják a Sensorhub-nak. Az általam megírt mobil alkalmazás egy az intézeti Sensorhub-ot lekérdező kliensek közül.

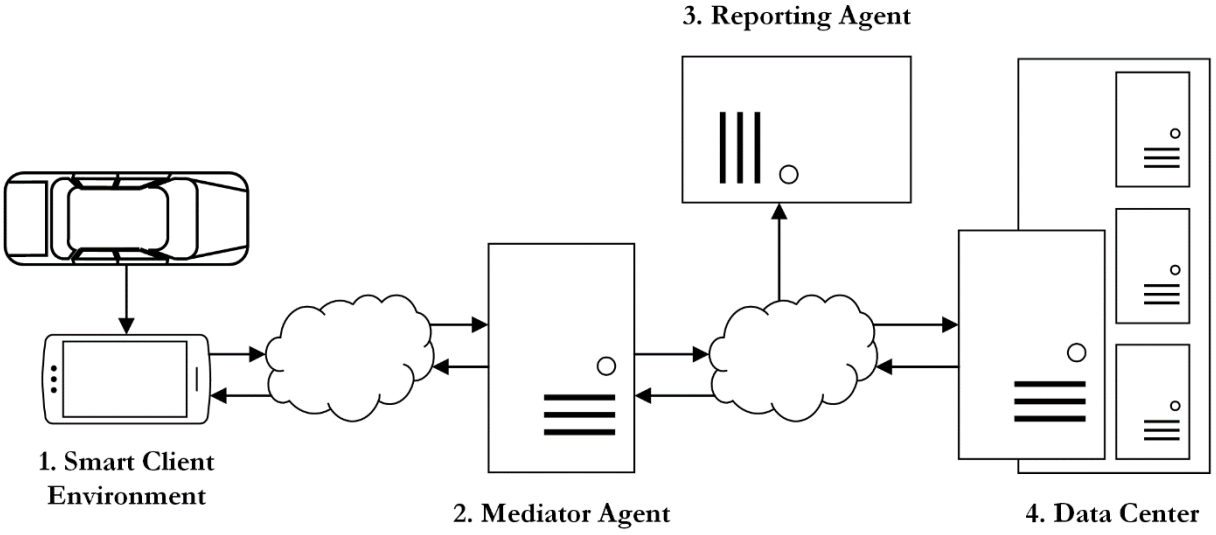


5. ábra - A Sensorhub komponensei [2]

4. Kitekintés, kutatómunka bemutatása

Ebben a fejezetben bemutatok két olyan alkalmazást, amely szintén szenzorokból gyűjt adatokat, és az adatok tárolására Big-Data környezetet használ. A fejezet végén fejlesztéshez kapcsolódó kutatómunkámat is megemlítem.

Az egyik ilyen alkalmazás a VehicleICT platform [4], amelyet a BME Villamosmérnöki és Informatikai Kara készített el, az egyetem Közlekedés és Járműmérnöki Karával közösen. A platformnak a célja az, hogy az autókat lehessen mobiltelefon készülékek segítségével monitorozni. A mobiltelefonok szenzorai a GPS pozíció meghatározására szolgálnak, de a rendszer biztosít további szenzorokat is, amelyek a gépjármű telemetriai adatait mérik, mint például a motor fordulatszáma vagy fogyasztása. Ahhoz, hogy bármilyen VehicleICT szolgáltatást használó alkalmazást tudjunk használni egy mobiltelefonon telepíteni kell, egy felülette nem rendelkező VehicleICT platform alkalmazást a készülékre, amely a platform szolgáltatásait nyújta a felhasználói felülettel is rendelkező alkalmazás felé. Mivel az adatok keletkezési időköze kevesebb másodperces nagyságrendű, ezért szükséges kiépíteni a megfelelő adat tároló architektúrát a rendszer számára. Ezt elosztott megoldással egy Apache Hadoop klaszter végzi el. Az androidos mobil alkalmazások nem közvetlenül kommunikálnak a data center-rel, hanem egy mediator agent-en keresztül. Az architektúra egy negyedik része a reporting agent, ami üzleti logikai funkciókat tartalmaz, és offline adatelemzést tesz lehetővé A 6. ábrán a VehicleICT platform architektúrája látható.



6. ábra - A VehicleICT platform architektúrája [4]

Egy másik ilyen alkalmazás Ikvahidi Ádám hallgatótársam Önálló laboratórium 2. tárgy keretében elkészített alkalmazása a Geosten [3]. Ez az alkalmazás földbe süllyesztett szenzorok hőmérséklet adatait tudja lekérdezni, és képes őket megjeleníteni 3D-ben is. Az alkalmazás által monitorozott föld szenzorok a Nyugat-magyarországi Egyetem soproni botanikus kertjében találhatóak, és 7 koordinátáról, több mélységben szolgáltatják az adatokat. Ez az alkalmazás képes együttműködni a VehicleICT platformmal is, de az esetek több részében a BME/NymE Sensorhub architektúrát használja adatforrásként. Az alkalmazás többek között képes megjeleníteni grafikonon napi és éves bontásban is a szenzorok által mért hőmérsékleti adatokat. Egy kiválasztott mélységben lévő szenzorok adatait, és az összes szenzor elhelyezkedését a föld alatt.

Mivel számora a legnagyobb újdonságot a félévben a grafikonok rajzolása jelentette, erre vonatkozóan végeztem még kutatómunkát a félév elején. Több megoldás is létezik, amellyel grafikonokat lehet rajzolni egy androidos alkalmazás keretein belül. Az egyik legelterjedtebb eszköz a *Google Chart*. Ezt a megoldást webes környezetre tervezték [5]. Androidos alkalmazásba úgy lehet elhelyezni ilyen grafikonokat, hogy úgynevezett WebView-kat helyezünk el az alkalmazáson belül. Ezeknek a tartalma Javascript nyelven programozható. Én egy Java nyelven programozható eszközkönyvtárat szerettem volna kipróbálni, a választásom így az *MPAndroidChart* nevű androidos eszközkönyvtárra esett, amely spesciális View elemeket tartalmaz, amiket az alkalmazás kezelő felületére helyezve grafikonok megjelenítésére használhatunk [6].

5. Az ErtiDataViewer alkalmazás bemutatása

A következő fejezetekben a saját fejlesztési munkámat fogom bemutatni, amit elvégeztem a tárgy keretein belül a félév során. A feladatom egy androidos mobilalkalmazás elkészítése volt, amelyik képes az ERTI által szolgáltatott adatok megjelenítésére, viszont, ha szükséges, akkor más szenzorokat, és adatait is meg tudja jeleníteni. Másik fontos követelmény volt az alkalmazással szemben, hogy az Informatikai és Gazdasági Intézetben futó Sensorhub környezethez tudjon kapcsolódni, és azt adatforrásként tudja használni.

A Sensorhub környezetet is ebben a félévben alakították ki az intézet más hallgatói, ezért nem tudtam a félév elején ezt használni adatforrásként. Ez a tényező nagyban meghatározta a féléves menetet is számomra. Ezt figyelembe véve terveztem meg az alkalmazást a félév elején, és kezdtem el az implementálását is.

Az adatbetöltéshez kapcsolódó funkciókat emiatt egy interfészben határoztam meg, és a benne definiált funkciókat először az 1995-ös minta adatokra kezdtem el megvalósítani helyi fájlolvasás segítségével. A félév végén elkészítettem az interfésznek egy másik implementációját is, amely a Sensorhub-bal kommunikál, és onnan tölti le az adatokat. Mivel a két osztály funkcionalitásának volt egy közös metszete, mint például az adathiányok kezelése, ezért később ezeket a funkciókat egy közös ősosztályba helyeztem. Az adatkezeléssel kapcsolatban azt kell még megemlíteni, hogy az adatáramlás szigorúan egyirányú, a mobilkliens nem tölt fel adatokat a szerverre, és nem is módosítja a szerveren lévő adatokat. A fájl implementáció hasonló elven csak olvassa az adatfájlokat, és nem módosítja azokat.

Fontos volt az entitás osztályok szerkezetének definiálása is. Az alkalmazás két fajta entitás osztályt használ. Az egyik típusba a szenzorok tartoznak, a másikba pedig a szenzorok által mért értékek. Jelen alkalmazás mindkét absztrakt osztályból két konkrét osztályt definiál, mivel két fajta adattal dolgozik a program. Egyrészt feldolgozza az OMSZ által szolgáltatott meteorológiai szenzorok adatait, és megjeleníti őket a térképen. Hasonló elven dolgozza fel az ERTI rovarfogás adatait, az ország különböző rovarcsapdáiban, és jeleníti meg a csapdákat térképes formában.

Az alkalmazást úgy alkottam meg, hogy az képes legyen különböző szűrési feltételek alapján ábrázolni a rovarfogásokat az egyes csapdákban. Ehhez a legszemléletesebb megoldásnak a grafikonok választása tűnt.

Az alkalmazás térképeket is használ a szenzorok helyeinek a megjelenítésére. Nagyon jó, könnyen kezelhető eszközkönyvtárat biztosít az Android programozók számára is a Google Maps szolgáltatás. A félév során ezt a megoldást használtam a térképek, és rajtuk a szenzorok megjelenítésére.

Az alkalmazás felhasználói felületét igyekeztem minél felhasználóbarátabb módon megtervezni. Ahol lehetett a dátum mezők kitöltéséhez naptár nézetet használtam. A számokat is speciális beviteli mezőkön keresztül kéri be a program, hasonlóan a fajnevek megadásához. Az androidos készülékek kijelző méretei különbözőek, ezért léteznek olyan megoldások, amelyek egy mobiltelefonon működnek, viszont egy táblagépen nem mutatnak jól, és fordítva. A felhasználói felület megalkotásakor ezt is figyelembe vettem, és külön nézetet készítettem telefonok, és tabletek számára. Az alkalmazás színvilágában is az erdő köszön vissza, ami a célközönség számára kellemes felhasználói élményt nyújt.

A félév során elkészítettem egy prototípus verziót, amely fájladatokon dolgozik, és a felhasználói felülete nem olyan kifinomult. Ennek a verziónak a feladata a funkcionalitás megalkotása volt. Majd elkészült ez alapján egy vékonykliens alkalmazás, ahol már az adatforrás a Sensorhub, és a felhasználói felület is optimalizálva van. Ez a verzió egy pár új funkciót is tartalmaz, mint például az internet kapcsolat kezelése, és támogatja a magyar mellett a német és angol nyelvű használatot.

A következő fejezetekben leírom az alkalmazás entitásait, adatkezelését és felhasználói felületét. Majd részletesebben ismertetem a prototípus verziót, ezután bemutatom a vékony kliensre történő átállást és a vékonykliens többlet funkcióit.

5.1. Az entitás osztályok

Az alkalmazás működése során két fajta entitással dolgozik. Az egyik entitáscsoport a szenzorokat reprezentálja, a másik pedig a szenzorok által szolgáltatott értékeket. Jelen alkalmazásban mindkét entitástípusnak két képviselője van, az egyik a meteorológiai adatokra, a másik a rovarcsapdákra fókuszál, de tetszőlegesen kiterjeszthető bármilyen más szenzorhely, illetve szenzoradat objektumra.

A szenzorhelyek megjelenítéséért a *SensorPlace* absztrakt osztály felel. Ennek két gyermekosztálya van implementálva a programban, a *MeteorologicalPlace* illetve, a *TrapPlace* osztályok. Az első a meteorológiai adatok statisztikailag származtatott helyeit tartalmazza, ezek a térképen, egy 20 km-es oldalhosszúságú négyzetekből álló rácsot alkotnak. A másik osztály a rovarcsapdák tényleges helyeit reprezentálja, melyeknek elhelyezkedése szabálytalan az ország területén belül.

A *SensorPlace* osztály tartalmazza az adott szenzornak az azonosítóját, illetve annak szélességiés hosszúsági koordinátáit. A gyermekosztályok számára biztosít üres, paraméter nélküli konstruktort, illetve egy olyat, amelyik mindhárom adattagot beállítja. Az osztály a tagváltozókhoz publikus beállító, illetve lekérdező metódusokat is ad. Az osztály tartalmaz egy *getLocation()* nevű függvényt is, amely a két koordinátából egy *android.location.Location* objektumot hoz létre, és azzal tér vissza. Ez a *Location* osztály tartalmaz beépített távolságszámító szolgáltatást, ez több helyen könnyebbséget jelent a programozás során. Az osztály egy további fontos tulajdonsága, hogy megvalósítja a *Serializable* interfészt. Ez az interfész semmilyen metódus felüldefiniálását nem teszi kötelezővé, csak egy jelzés a fordító felé, hogy az osztály bájtfolyammá alakítható. Ennek a tulajdonságnak köszönhetően az osztály, és gyermekosztályai lehetnek egy Intent extra paraméterei, így átadhatók két Activity között is.

A projektben két *SensorPlace* implementáció található. A *MeteorologicalPlace* osztály felel a meteorológiai szenzorhelyek megjelenítéséért. Csak az objektumstruktúra teljességéért lett létrehozva, semmilyen funkcionalitással nem egészíti ki az ősosztályt, saját tagváltozókat, sem definiál.

A másik *SensorPlace* gyermekosztály a *Trapplace* nevű entitásosztály. Ez a komponens felel a rovarcsapdák megjelenítéséért a programban. Az ősosztályt kiegészíti egy *city* nevű *String* paraméterrel, amelyik a csapda települését tartalmazza, és természetesen lekérdező, és beállító metódusokat is szolgáltat hozzá.

Az *ObservedObject*, magyarul megfigyelt objektum osztály felel a szenzorok által mért adatok megjelenítéséért. A *SensorPlace* osztályhoz hasonlóan ez is implementálja a Serializable interfészt. Az osztály tartalmazza az adott szenzoradat mérésének az idejét, pozícióját, illetve a mért adatokat. Az osztály lehetőséget teremt arra, hogy olyan objektumokat is ábrázoljunk, amihez nem tartozik mérési érték. Ehhez egy logikai változóban tárolja el, hogy az adott adatmező érvényes-e vagy sem. Az osztály az időparamétert öt tagváltozó segítségével tárolja el, az évestől a perces nagyságrendig. Jelen alkalmazás napi mélységig ábrázolja csak az adatokat. A pozíciót két koordináta formájában tárolja az osztály, e mentén lehet összekapcsolni a mért adatokat a szenzorokkal. Mivel nem csak egy adatot mérő szenzorok létezhetnek ezért a mért adatokat egy Double generikussal rendelkező listában tárolja az osztály. Minden gyermekosztály annyi elemet foglal magának a tömbből, ahány mért értéke van, és saját getter és setter metódusaival a tömb létezését elfedi a használója elől. Ugyanígy létezik egy Boolean

alapú *naValues* nevű lista is, amely azt tárolja el, hogy az egyes mért adatok érvényesek-e. A hamis paraméter jelenti, hogy érvényes a paraméter, az igaz pedig azt, hogy érvénytelen.

Az osztály több féle konstruktort biztosít gyermekosztályai számára. Létezik a paraméterek nélküli üres konstruktora is. Egy konstruktor az év, hónap és nap paraméterek beállítására, egy, amelyikkel az idő, és a koordináta adatokat lehet beállítani, és ennek egy másik változata, amelyben a nap beállítása nem szerepel. Az osztály megvalósítja az adattagok getter és setter metódusait, és elérést biztosít a két listához is. Hasonlóan a *SensorPlace* osztályhoz ennek az osztálynak is van *getLocation()* metódusa is. Az osztályban definiálva van egy *getDateString()* függvény is, amelyik visszaadja a dátumot szövegként. Ennek a metódusnak az adatbetöltő adathiány kezelésénél van fontos szerepe. Továbbá az osztály megvalósítja a *Comparable* interfészt is, ami dátum szerint rendezi az objektumokat. Az interfész *compareTo(ObservedObject o)* metódusa a *getDateString()* alapján végzi el az összehasonlítást a két obejktum között.

```
public String getDateString() {
    if (this.day >= 10) {
        if (this.month >= 10) {
            return year + "-" + month + "-" + day;
        } else {
            return year + "-0" + month + "-" + day;
        }
    } else {
        if (this.month >= 10) {
            return year + "-" + month + "-0" + day;
        } else {
            return year + "-0" + month + "-0" + day;
        }
    }
}
```

A meteorológiai szenzorok adatait a *MeterologicalData* osztály ábrázolja. Ez az osztály az ősosztályhoz nem ad hozzá semmilyen új adattagot, és a funkcionalitását sem bővíti tovább. Az osztály az ősosztály bizonyos tagváltozóit nem használja, például nincs benne nap adattag, mivel nem olyan mély a mérések felbontása. Konstruktorában hívja az ős konstruktorát, illetve beállítja a szenzor-adatokat tartalmazó listákat, és az adathiány-listát beállítja adathiány nélküli állapotba. A konstruktor itt már a konkrét osztálynak megfelelő adattagokat vár paraméterként, és azokat konvertálja a lista elemivé. Az osztály *getWet()*, *setWet()*, *getTemp()* és *setTemp()* metódusai is hasonlóképpen működnek.

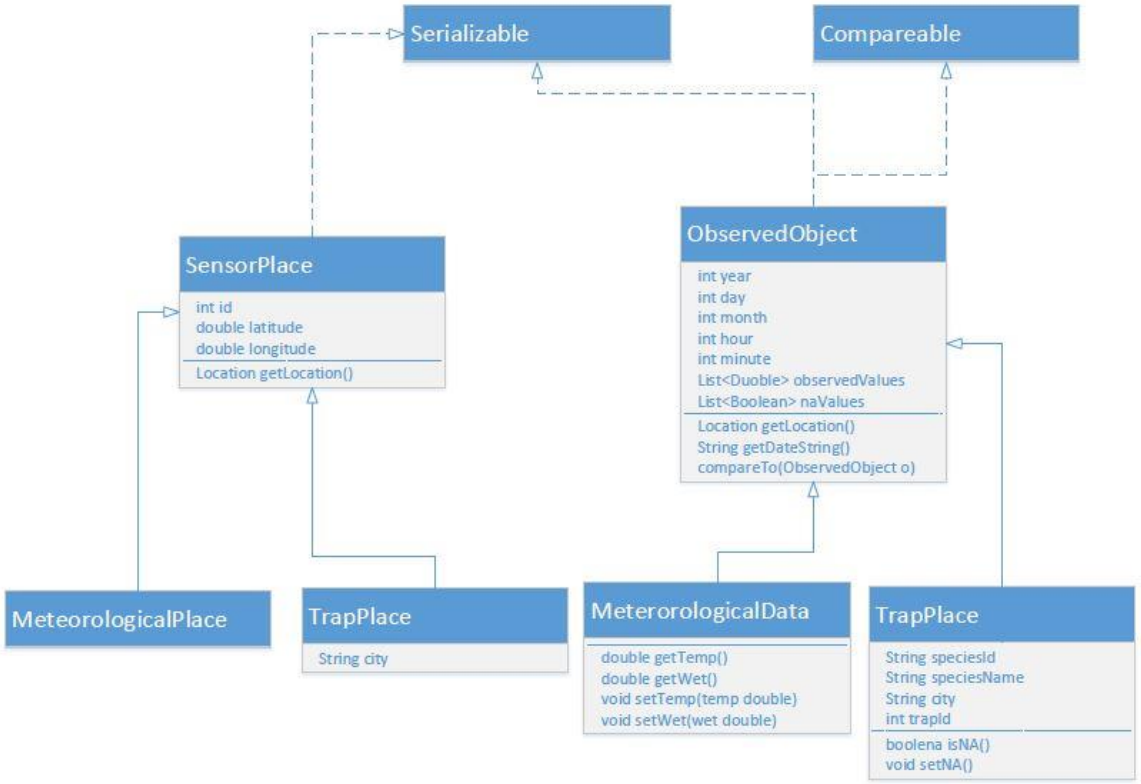
```
public MeteorologicalData(int year, int month, double temp, double wet,
double longitude, double latitude) {
   super(year,month,longitude,latitude);
   ArrayList<Double> observedValues = new ArrayList<Double>();
   observedValues.add(temp);
   observedValues.add(wet);
   super.setObservedValues(observedValues);
   ArrayList<Boolean> naValues = new ArrayList<>();
   naValues.add(false);
   naValues.add(false);
   super.setNaValues(naValues);
}
```

Meg kell még említeni az *isWetNA()*, az *isTempNA()*, a *setWetNA()*, és a *setTempNA()* metódusokat. Ezek közül a lekérdezők visszaadják, hogy érvényes-e az adatmező. Viszont a beállítók mindig igaz, tehát adathiányos állapotra állítják a változókat.

Az *InsectTrap* osztály jeleníti meg az egyes rovarcsapdákat. Ez az osztály is egy *ObservedObject* leszármazott. Ebben az esetben viszont bővül az osztály adattagjainak a száma, viszont itt nem jelennek meg a koordináta attribútumok, mivel az adatforrások sem tartalmaznak ilyet. Ebben az esetben a *trapId* vagy a *city* attribútum mentén lehet összekapcsolni a csapdát, a fogási hellyel. Az osztálynak két tagváltozója van még, amelyekből az egyik a fogott rovar latin fajnevét tartalmazza, a másik pedig egy faj-azonosítószám, szintén szöveges formátumban. A két attribútum között funkcionális függőség áll fenn, az adatforrás miatt maradt benn mind a két attribútum az osztályban.

A konstruktorok, a getterek, és a setterek működése hasonlóan van itt is megvalósítva, mint a MeteorologicalData osztályban. Itt az adathiány kezelésre egy beállító, és egy lekérdező metódus szerepel *setNA()*, és *isNA()* néven. Itt sem lehetséges később a hamis érték beállítása.

A fejezet végén a könnyebb áttekinthetőség kedvéért egy UML diagramot készítettem, amelyik a 7. ábrán látható.



7. ábra - Az entitás osztályok

5.2. Az alkalmazás adatkezelése

Ebben a fejezetben az alkalmazás adatforrásait, illetve adatbetöltő mechanizmusát ismertetem. Először leírom a *DataLoader* interfészt, és annak funkcióit. Majd ismertetem a prototípus által használt *FileDataLoader* osztályt, és egy következő fejezetben a Sensorhub-ot lekérdező *SensorhubDataLoader* osztályt, amit csak a kész verzió használ. Mivel több funkció megegyezik a két osztályban, ezért létrehoztam egy *AbstractDataLoader* osztályt is. Ez az osztály implementálja a *DataLoader* interfészt, a lekérdező osztályok ezt terjesztik ki, és valósítják meg az adatforrás-specifikus funkciókat. Az adatkezelésért a prototípus, és a kész verzióban a *dataloader* csomag osztályai felelnek. Ebben a csomagban található egy *ErtiDataViewer* osztály, ami az *android.app.Application* osztályt terjeszti ki. Ennek köszönhetően felül lehet definiálni benne az alkalmazás indulását az *onCreate()* metódust használva. Ez a metódus a program életciklusának elején a felhasználói felültet megjelenítése előtt fut le. Itt kap helyet a dataLoader inicializálása, a prototípus esetében egy *FileDataLoader* lesz a dinamikus típusa, míg a kész verzióban *SensorhubDataLoader*. A *dataLoader* objektum az alkalmazás többi részéből is elérhető egy statikus getteren keresztül. A prototípus közvetlenül hívja a gettert a grafikus megjelenítésért felelő osztályokból. A kész verzió *AsyncTask* osztályokat használ erre a célra, mivel Android környezetben nem lehet a grafikus szálon hálózati forgalmat generálni, ami a *SensorhubDataLoader* működése szempontjából elengedhetetlen. A háttérszálon futó *AsyncTask* osztályok az *acynctasks* csomagban találhatók, erre a megoldásra majd egy későbbi fejezetben kitérek.

5.2.1. A DataLoader interfész

A *DataLoader* interfész határozza meg azokat az adatlekérési funkciókat, amelyekre az alkalmazásnak szüksége van. Az interfész alapvetően adatlekérő metódusokat definiál, módosító vagy adatfeltöltő funkciókat nem végez. Az interfész az alábbi metódusokat deklarálja:

- *void loadAll()* - az összes adat betöltése
- *void loadAllPlaces(int type)* - az összes adott típusú SensorPlace betöltése
- *void loadAllSensorData(int tpye)* - az összes adott típusú ObservedObject betöltése
- *List<SensorPlace> getAllPlases()* - listában visszadja az összes SensorPlace objektumot
- *List<ObservedObject> getAllObservedObjects()* - listában visszaadja az összes ObservedObject objektumot
- *List<SensorPlace> getPlases(int type)* - listában visszaadja az összes adott típusú SensorPlace objektumot
- *List<ObservedObjet> getObserdedObjes(int type)* - listában visszaadja az összes adott típusú ObservedObject Objektumot
- *List<String> getAllInsectSpecies()* - visszaadja az adatbázisban található összes rovarfaj nevét egy listában
- *List<String> getSpeciesByCity(String city)* - viszaadja az adott városban valaha fogott rovarfajok listáját
- *List<ObservedObject> getInsetTrapBetweenDatesBySpeciesAndPlace(int beginyear, int beginmonth, int beginday, int endyear, int endmonth, int endday, String species, String city, String aggregationtype)* - visszaadja az adott idő intervallumban lévő rovarfogásokat, a kiválasztott fajra, és településre, a megfelelő aggreációtípus szerint

Első ránézésre feleslegesnek tűnhet külön *loadAll(),* illetve *getAll()* jellegű metódusok létrehozása. Ennek főleg a fájl implementációnál van szerepe. A load metódusok beolvassák

fájlból listákba az elemeket, és a *get* metódusok ebből adják vissza az eredményt, így a fájlolvasás csak egyszer történik meg, nem kell minden adatlekéréskor a fájlhoz fordulni.
A *DataLoader* interfész négy *int*, és két *String* típusú statikus konstanst tartalmaz, amik a metódusok paraméterezési biztonságát növelik. Az *int* típus a visszaadandó objektum típusát állítja be.

```
int TRAPPLACE = 1;
int METEOROLOGICALPLACE = 2;
int INSECTTRAP = 3;
int METEOROLOGICALDATA = 4;
```

A *String* típusú konstansok pedig az aggregáció mélységét határozzák meg. A DAY konstans a naponkénti lekérést, a MONTH konstans a havi rovarfogás összegzést jelenti.

```
String DAY = "DAY";
String MONTH= "MONTH";
```

5.2.2. A FiledataLoader osztály

A prototípus verzióban szöveges *.csv* kiterjesztésű fájlok tárolják az adatokat az alkalmazás számára. Ezek természetesen nem ölelik fel az 1965-től napjainkig tartó időszakot, csak az 1995-ös rovarfogás és meteorológiai mérés adatok kerültek be a teszt programban. A rovarcsapdákat is meg kaptam egy külön fájlban, és az is bekerült a programba. A meteorológiai „mérőhelyeket" tartalmazó adatfájl nem létezett ezért azt én generáltam egy szűréssel a mérési adatokat tartalmazó fájlból.

Ez a négy fájl a projektben a */res/raw* mappában található, ez az internal-storage része, ami az alkalmazás saját fájl tárolója, tehát nincs szükség külön engedély az olvasásukra. A *FileDataLoader* egy *ContextWrapper* objektum segítségével fér hozzájuk az *R.java* osztályon keresztül a következő metódushívással:

```
new BufferedReader(new InputStreamReader(contextWrapper.getResources().
openRawResource(R.raw.meteorologicalplace)));
```

Ezen a konstruktoron keresztül érik el a load típusú metódusok a fájlokat, melyekből létre tudják hozni a listákat, amikkel a get típusú metódusok dolgoznak. A get típusú metódusok szolgáltatják az adatokat az alkalmazás többi része számára.

Az osztály talán legfontosabb metódusa a *getInsectTrapBetweenDatesBySpeciesAndPlace(int beginyear, int beginmonth, int beginday, int endyear, int endmonth, int endday, String species, String city, String aggregationType)* metódus, ez az eljárás kéri le az adott időintervallumban lévő rovarfogás adatokat, melyek megfelelnek a fajnév és a település attribútumoknak. Az aggregáció típusától függően vagy minden nap egy külön rekord lesz a visszaadott listában, vagy egy hónap jelent egy lista elemet, amelybe a havi fogásösszeg kerül az adott fajból. Az adott időintervallumba tartozás eldöntéséhez az *AbstractDataLoader* osztálynak van egy *makeDateSring(int year, int mont, int day)* metódusa, amely hasonló, mint az *InsectTrap* osztály *getDateString(int year, int month, int day)* metódusa. A program a *compareTo* metódus segítségével hasonlítja össze a két *String-et*. Ha a rovarfogás által visszaadott *String* érék nagyobb a kezdődátumból generált *String-nél*, és kisebb a végdátumból generál *String-nél,* akkor benne van az intervallumban, egyébként nincs.

```
ArrayList<ObservedObject> its = new ArrayList<>();
for (ObservedObject observedObject : insectTraps) {
    InsectTrap insectTrap = (InsectTrap) observedObject;
    String dateStringBegin = makeDateSring(beginyear,beginmonth,beginday);
    String dateStringEnd = makeDateSring(endyear,endmonth,endday);
    if (insectTrap.getDateString().compareTo(dateStringBegin) >= 0 &&
insectTrap.getDateString().compareTo(dateStringEnd) <= 0) {
        if (insectTrap.getCity().equals(city)) {
            if (insectTrap.getSpeciesName().equals(species)) {
                its.add(insectTrap);
            }
        }
    }
}
```

A metódus egy listába kigyűjti a megfelelő elemeket, viszont itt nem áll meg a feladata. A grafikonrajzoló függvényeknek fontos, hogy minden adatsor tényleg olyan hosszú legyen, mint a kiválasztott időszak, és az adathiányok is le legyenek kezelve. Ha a felhasználó két rovarfajnak az adatait szeretné havi összegzéssel lekérdezni 1995 március-áprilisi időszakban. Akkor a megfelelő előfeldolgozás nélkül, ha az egyik faj adatai mindkét hónapból rendelkezésre állnak, a másikból csak áprilisi adat van, akkor a második faj áprilisi adatai tévesen márciushoz jelenítené meg a program. Egy adatrekordról el tudja dönteni a grafikonrajzoló függvény, hogy annak értéke azért nulla, mert nem fogtak abból a rovarból aznap, vagy abban a hónapban, vagy azért, mert nem állnak rendelkezésre adatok az adott időszakból.

A metódus az adathiányok kezelését az AbstractDataLoader osztály *createNA(beginyear, beginmonth, beginday, endyear, endmonth, endday, species, city, its, aggregationType)* metódusa felé delegálja. Ez a metódus a *createNAYear(int year, int beginmonth, int beginday, int endmonth, int endday, ArrayList<ObservedObject> observedObjects, String species, String city)* illetve a hasonlóan működő *createNAMont* metódusokat hívja. Ezek segítségével az adatsorok hossza egyforma lesz, és az adathiányokat is tartalmazni fogják. A metódus a futás végén visszatér a rendezett, fogási adatokat tartalmazó listával.

5.2.3. A SensorhubDataLoader osztály

A program másik adatforrása az intézetben futó Sensorhub adattároló platform. Ennek az elérésére szolgál a *SensrohubDataLoader* nevű osztály. Az osztály felhasznál egy *Api* típusú objektumot a működéshez. Az *Api* osztály segítségével lehet kapcsolódni az intézeti Sensorhub-hoz. Az *Api* osztálynak két fajta konstruktora az egyik csak egy *String* Api-kulcsot kér, ezzel megadva, hogy melyik szolgáltatáshoz szeretne csatlakozni. A másik konstruktor kér egy *int* paramétert is, ennek segítségével lehet az intézeten belüli hálózaton elérni a Sensorhub-ot. Erre azért van szükség mivel belső hálózatról más IP-cím alatt érhető el a szolgáltatás, mint kívülről. Mivel a belső hálózat nem mindig működik stabilan, az alkalmazás a külső elérést használja csak.

```
public SensorhubDataLoader(int connectType) {
    if(connectType == OUTER){
        api = new Api("NYMSKKSNSRHBPKYRT");
    }
}
```

A *SensorhubDataLoader* funkcionalitása megegyezik a *FileDataLoader-évelː* Viszont működési filozófiájuk eltért. A *FileDataLoader* először saját listákban letárolja az adatokat, és

azokat kezdi el szűrni a későbbi lekérdező metódusokban. Mivel az internetes kommunikáció lassabb, mint a fájlolvasás, és a Sensorhub jóval több adatot tartalmaz, mint a tesztfájlok, ezért itt nem jöhet szóba az összes adat lekérése működés közben, mindig csak annyi adat töltődik le a kliensre, amennyire szükség van az adott aggregációban. A *SensorhubDataLoader-ben* ezért a *void* visszatérési értékkel rendelkező *load* típusú metódusok, amelyek a listák feltöltését végeznék el, csak üres implementációval vannak megvalósítva. Az adatlekérést mindig a lekérés idejében végzik el a get típusú metódusok.

A rovarfogásokat lekérő *getInsectTrapBetweenDatesBySpeciesAndPlace(int beginyear, int beginmonth, int beginday, int endyear, int endmonth, int endday, String species, String city, String aggregationType)* metódus is másképp működik, mint a fájl implementáció. Itt az *api.getWithNames(getApiTables().get(2), columnsToCheck, names, cols)* metódussal csak azon rovarfogások lekérése történik meg, melyeknél egyezik a település, illetve faj attribútum a megadottakkal. Az előbb említett metódus első paramétere a rovarfogásokat tartalmazó tábla neve, utána kell megadni, hogy melyik oszlopokra fogalmazunk meg feltételt, majd a feltétlek értéket kell megadni, végül a visszaadandó oszlopok jönnek, ez utóbbi három paraméter egy-egy *String* tömb. Mivel a Sensorhub-ban fix hosszúságú szövegként szerepelnek a faj, illetve település adatok, ezért a bemenő paramétereket is arra a hosszra kell konvertálni, a *String* osztály format metódusával, hogy összehasonlíthatók legyenek egymással.

```
String[] cols = {"csapdaazonosito", "ev", "fajkod", "fajnev", "fogasszam",
"honap", "nap", "telepules"};
String[] columnsToCheck = {"telepules", "fajnev"};
String[] names = {city, species};
names[0] = String.format("%1$-" + 20 + "s", names[0]);
names[1] = String.format("%1$-" + 40 + "s", names[1]);
String jsonString = api.getWithNames(getApiTables().get(2), columnsToCheck,
names, cols);
```

Ezután jön az adatok feldolgozása egy *JsonArray* segítségével. A metódus végig iterál a *JsonArray* objektumon, és miden eleméből létrehoz egy *JsonObject-et*. Ennek segítségével már elérhetőek a rekordok attribútumai. A szöveges mezők fix hosszúságúak, és szóköz kitöltést tartalmaznak, ezektől természetesen meg kell szabadulni, hogy ne kerüljenek bele a létrejövő entitásokba, ezért meg kell rajtuk hívni a *trim()* metódust.

Mivel a lekért adatok nincsenek szűrve az idő attribútumokra, ezért nem kerülhet minden entitás a listába. A dátumszűrést itt is, *FileDataLoader-hez* hasonlóan a *makeDateString(int year, int month, int day)* metódus, illetve az *insectTrap.getDateString()* metódusával történik meg. Az adathiányok lekezelése itt is az *AbstractDataLoader* osztály *createNA(beginyear, beginmonth, beginday, endyear, endmonth, endday, species, city, its, aggregationType)* metódusával történik meg.

```
String dateStringBegin = makeDateSring(beginyear, beginmonth, beginday);
String dateStringEnd = makeDateSring(endyear, endmonth, endday);
if (insectTrap.getDateString().compareTo(dateStringBegin) >= 0 &&
insectTrap.getDateString().compareTo(dateStringEnd) <= 0) {
    if (insectTrap.getCity().equals(city)) {
        if (insectTrap.getSpeciesName().equals(species)) {
            its.add(insectTrap);
        }
    }
}
```

Az osztály tartalmaz egy saját metódust is, amelyik a táblaneveket adja vissza, ez a *getApiTables* metódus. A lekéri az api-ban lévő összes táblaadatot az *api.getAvailableTables()* metódus segítségével. Ez visszaadja a táblaneveken kívül a táblák attribútumait is. Az egyes JSON elemeknek a kulcsai a táblák nevei, a metódus végig iterál az összes JSON objektumon, és elkéri azoknak a kulcsait, ezt teszi bele egy *String* típusú listába, és ezzel tér vissza. A többi metódus ennek a listának az elemei segítségével éri el a táblaneveket.

```
private ArrayList<String> getApiTables() {
    ArrayList<String> tables = new ArrayList<>();
    String apitalbes = api.getAvailableTables();
    System.out.println(apitalbes);
    JsonArray jsonArray = (JsonArray) new JsonParser().parse(apitalbes);
    for (int i = 0; i < jsonArray.size(); i++) {
        JsonObject jsonObject = jsonArray.get(i).getAsJsonObject();
        for (Map.Entry<String, JsonElement> entry :
        jsonObject.entrySet()) {
            tables.add(entry.getKey());
        }
    }
    for(String s : tables){
        System.out.println(s);
    }
    return tables;
}
```

5.2.4. Az AbstractDataLoader osztály

Az *AbstarctDataLoader* osztály azért került bevezetésre, mert bizonyos funkciókat ugyanúgy végez el a *FileDataLoader* osztály és a *SensorhubDataLoader* osztály is. Ebből következőleg szükségét láttam egy közös ősosztály bevezetését. Így a közös funkciók ebbe az absztrakt osztályba kerültek bele, és ez az osztály implementálja a *DataLoader* interfészt, viszont metódusait nem valósítja meg, hanem a gyermekosztályokra bízza azt. Az osztály funkcionalitásába tartozik az adathiányok kezelése. Egy metódus mély másolatokat készít az objektumokról, illetve egy a dátumok szöveggé alakításáért felel.

Három adathiány kezelő van az osztályban, ezeknek a feladata, hogy azonos hosszúságú adatsorokat készítsenek a rovarfogás adatokból, és szerepeljenek bennük az adathiányok is. A *createNA(int beginyear, int beginmonth, int beginday, int endyear, int endmonth, int endday, String species, String city, ArrayList<ObservedObject> its, String aggregationType)* metódust hívják a gyermekosztályok *getInsectTrapBetweenDatesBySpeciesAndPlace* metódusai, ez adja tovább a vezérlést a másik két adathiány kezelőnek a *createNAYear*, illetve *createNAMonth* metódusoknak, melyek egy év, illetve egy hónap adathiányait tudják generálni.

A *CreateNA* metódus különböző módon működik napi adatok illetve havi összegek lekérése esetén. Ha napi lekérésről van szó, akkor végig iterál az intervallum összes dátumán, és ha nem talál az adott dátumra fogást a listában, akkor beszúr oda egyet, aminek az *NA* attribútumát igazra állítja ezzel jelezve, hogy arra a napra nincs értékes adat. Azoknál az elemeknél, amik már korábban bekerültek a listában ez az attribútum hamis, jelezve, hogy nincs adathiány, a metódus nem nyúl hozzá a rekordhoz.

Egy időintervallumon való végig iterálás nem egy teljesen triviális feladat, esetszétválasztásra van szükség hozzá. A *CreateNA* metódus az iteráció során végig megy a kezdőévtől a befejező

évig, és a ciklusban vizsgálja a ciklusváltozó, illetve kezdő- és a befejezőév egymáshoz való viszonyát. Ha az aktuális év egyenlő a kezdőévvel, és a kezdő év nem egyenlő a befejező évvel, akkor a kezdő dátumtól, az év végéig kell adathiányt generálni a *createNAYear* metódussal. Ha a jelenlegi év nem egyezik se a kezdő se a befejező évvel, akkor egy teljes évnyi adathiányt kell generálni. Ha az év egyezik a befejező évvel, de a kezdőév nem egyezik a befejezőévvel, akkor az év elejétől a befejező dátumig kell adathiányt generálni. Ha a kezdő és a befejező év megegyezik, akkor az iteráció csak egyszer fut le, ilyenkor a kezdő, és a befejező dátum között kell adathiányt generálni.

```
for (int year = beginyear; year <= endyear; year++) {
    if (year == beginyear && beginyear != endyear) {
        createNAYear(year, beginmonth, beginday, 12, 31, its,
        species, city);
    }
    if (year != beginyear && year != endyear) {
        createNAYear(year, 1, 1, 12, 31, its, species, city);
    }
    if (year == endyear && beginyear != endyear) {
        createNAYear(year, 1, 1, endmonth, endday, its, species,
        city);
    }
    if (beginyear == endyear) {
        createNAYear(beginyear, beginmonth, beginday, endmonth,
        endday, its, species, city);
    }
```

Ehhez hasonló elven működik a *createNAYear* metódus is, amely szintén egy négy irányba elágazó esetszétválasztással hívja a *createNAMonth* metódust, ahol a tényleges adathiány kezelés történik.

A *createNAMonth(int year, int month, int beginday, int endday, ArrayList<ObservedObject> observedobjects, String city, String species)* metódus végzi el a tényleges adathiány generálást a hívási lánc végén. Ha teljes hónapon kell végig iterálni, akkor először meg kell állapítani az aktuális hónap hosszát. Erre szolgál a *getMaxDayInMonth(int year, int month)* metódus, amelyik megmondja az aktuális hónap hosszát figyelembe véve a szökőéveket is február esetében. Az évről, hogy szökőév-e a *new GregorianCalendar().isLeapYaer(int year)* metódussal dönti el. Azt, hogy a hónap végéig szeretnénk iterálni, az *endday=31* paraméterrel adjuk meg, és a metódus ezt csökkenti le a hónap tényleges hosszára, ez lesz a belső ciklus maxday attribútuma. Ha az *endday* kisebb 31-nél, akkor a megadott érték lesz a *maxday* belső változó. A minday, amitől kezdve a belső ciklus iterál, mindig megegyezik a beginday paraméterrel. Ezután a metódus végig iterál az összes dátumon, az intervallumon belül. A dátumot leképezi szöveggé a *makeDateString(int year, int month, int day)* metódus segítségével. Ez a metódus minden dátumból egyforma hosszú string-et készít úgy, hogy a 10-nél kisebb hónap, vagy nap értékek elé nullát szúr be a szövegbe. Ezután a metódus egy *foreach* ciklussal végig nézi, hogy van-e az adott dátumra illeszkedő elem a listában, ha talál ilyet, kilép a belső ciklusból, és igazra állít egy found nevű változót. Ha a belső ciklusból nem léptünk ki, és a found értéke hamis, akkor az adott dátumra létrehoz egy adathiánnyal rendelkező entitást, és hozzáadja a listához.

```
InsectTrap insectTrap = new InsectTrap(null, species, 0, year, month, day,
city, 0);
insectTrap.setNA();
observedobjects.add(insectTrap);
```

Havi összegzés esetén a képlet tovább bonyolódik. A havi összegek generálása, és az adathiány kezelés is a *createNA* metódus keretein belül történik meg. A havi összegeket az alkalmazás az elsejei rekordokban tárolja le. Ez a *FileDataLoader* esetében gondot okozott, mert egy havi összeg lekérés után módosultak az érintett elsejei rekordok. Mivel a Java referenciaként kezeli az objektumokat szükség volt egy mély másolatot készítő metódusra. Ez a metódus az osztály *copy(Object original)* nevű metódusa.

Ezután a metódus adatforrásként a másolat listát használja, és egy harmadik listába gyűjti össze a havi összegeket tartalmazó elemeket. Az összeggenerálás úgy működik, hogy a metódus választ egy kulcs elemet, amibe összegyűjti a kulcs hónapjába tartozó fogások összegét. A kulcs először vagy a lista első elemével van inicializálva, vagy egy új rovarfogás-példánnyal, melynek szöveges attribútumai megegyeznek a bemenő paraméterekkel, a dátuma pedig a kezdő dátum. Ezután történik egy iteráció, amiben a tényleges összegszámítás történik. A ciklusban minden elemre ellenőrzik, hogy egyezik-e a hónap attribútuma a kulcséval, viszont önmaga különbözik-e kulcstól. Ha igen akkor beállítja a kulcs elem tulajdonságait, illetve minden ciklusban a fogás attribútumához, hozzáadja az aktuális elem fogásszámát, és a kulcs nap mezőjét 1-re állítja. Ha a vizsgált elem hónap attribútuma nem egyezik meg a kulcselemével, akkor, ha a lista nem tartalmazza a kulcsot, beszúrja a listába, majd új kulcselemmel folytatja a keresést. Mivel a Sensorhub-ban tárolt adatok nem előrendezettek, fontos, hogy a metódus rendezés után kezdje el az összegzést, mert különben nem biztos, hogy mindegyik elem bekerül a listába.

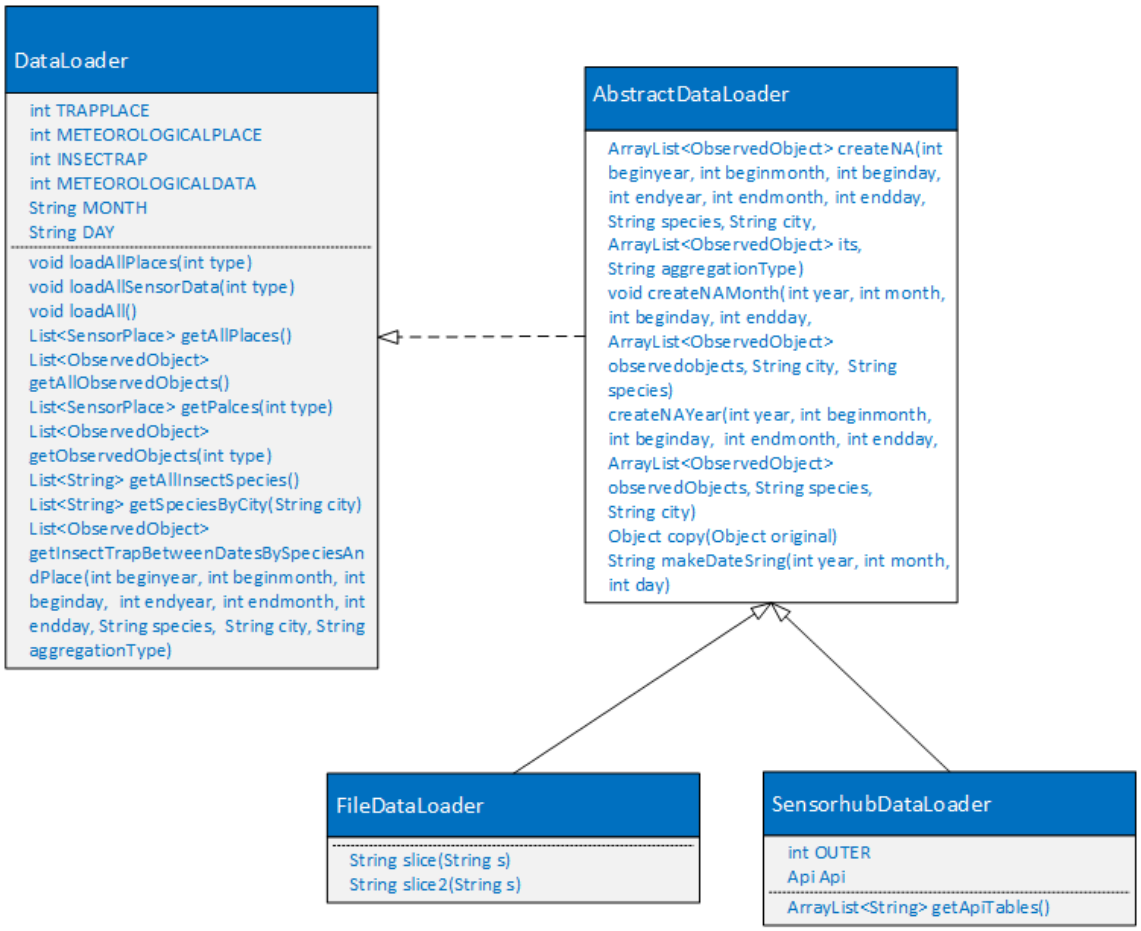
```
ArrayList<ObservedObject> itscopy = (ArrayList<ObservedObject>) copy(its);
InsectTrap key;
if (its.size() > 0) {
    key = (InsectTrap) itscopy.get(0);
} else {
    key = new InsectTrap(null, species, 0, beginyear, beginmonth, beginday,
    city, 0);
    key.setNA();
}
for (ObservedObject observedObject : itscopy) {
    InsectTrap insectTrap = (InsectTrap) observedObject;
    if (key.getMonth() == insectTrap.getMonth() && !key.equals(insectTrap))
    {
        key.setSpeciesId(insectTrap.getSpeciesId());
        key.setSpeciesName(species);
        key.setCity(city);
        key.setYear(insectTrap.getYear());
        key.setMonth(insectTrap.getMonth());
        key.setDay(1);
        int catches = key.getCatches();
        key.setCatches(catches + insectTrap.getCatches());
    } else {
        if(!list.contains(key)){
            list.add(key);
        }
        key = insectTrap;
    }
```

Ezzel még csak a havi összegek összegyűjtése van kész. Itt is szükség van adathiány kezelésre. Ebben az esetben is a korábbi bekezdésekben említett négyirányú esetszétválasztás történik

meg, és a végén a *createNAMonth* metódus segítségével történik az adathiányok keresése, és beszúrása a listába. Majd ez után történik egy újabb rendezés, hogy az adathiányok is biztosan jó helyen legyenek a listában, majd a metódus visszaadja a havi fogásösszegeket tartalmazó listát.

```
for (int year = beginyear; year <= endyear; year++) {
    int begin = 0;
    int end = 0;
    if (year == beginyear && beginyear != endyear) {
        begin = beginmonth;
        end = 12;
    }
    if (year != beginyear && year != endyear) {
        begin = 1;
        end = 12;
    }
    if (year == endyear && beginyear != endyear) {
        begin = 1;
        end = endmonth;
    }
    if (beginyear == endyear) {
        begin = beginmonth;
        end = endmonth;
    }
    for (int month = begin; month <= end; month++) {
        createNAMonth(year, month, 1, 1, list, city, species);
    }
```

A 8. ábrán látható az adatkezelő osztályok osztály diagramja. Az ábrán a szaggatott nyíl interfész-megvalósítást, a teljes nyíl öröklődést jelöl.

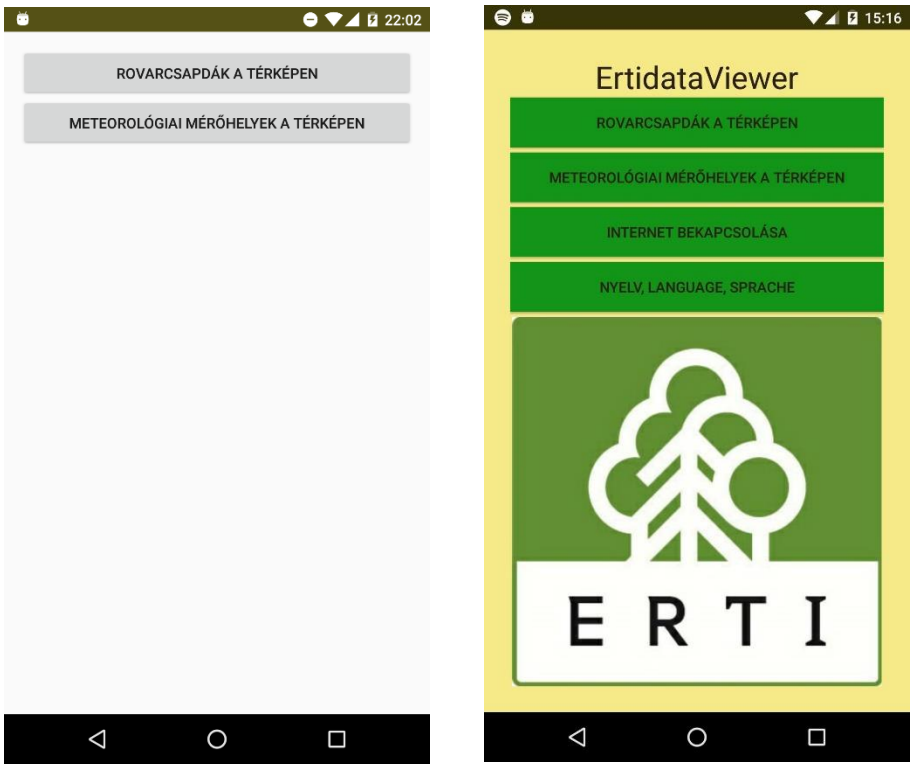


8. ábra - Az adatkezelő osztályok

5.3. Az alkalmazás felhasználói felülete

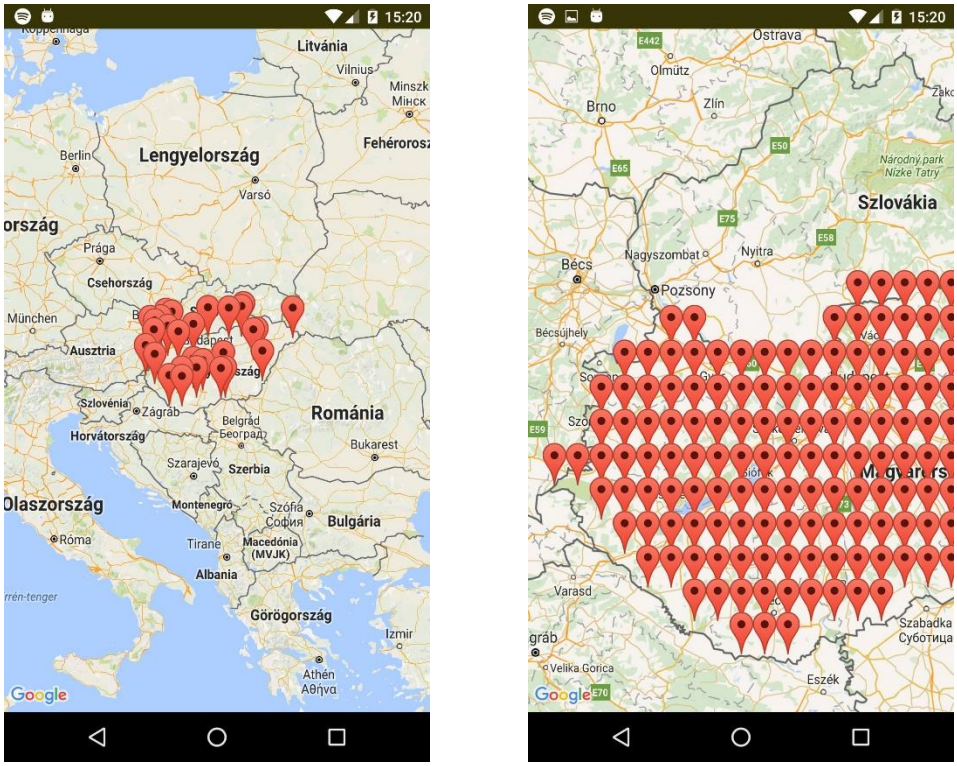
Ebben a fejezetben az alkalmazás felhasználói felületét ismertetem. Először leírom a program felhasználási esteit, és globális áttekintést adok a program működéséről. A későbbi fejezetekben részletesen kitérek a felhasználói felület egyes komponenseire programozói szemszögből, mélyebb betekintést nyújtva azokról.

Az alkalmazás különböző szenzorok térképen való megjelenítését, illetve az ott mért adatok szűrt vagy nyers lekérdezését teszik lehetővé. Az alkalmazás elindításakor a felhasználó a *MainActivity-vel* találkozik először, ami a 9. ábrán látható. Itt két lehetőség közül választhat a prototípus verzióban: Az egyik a rovarcsapdák megjelenítése a térképen, a másik a meteorológiai adatok forráshelyeinek a megjelenítése *Google Maps* segítségével.



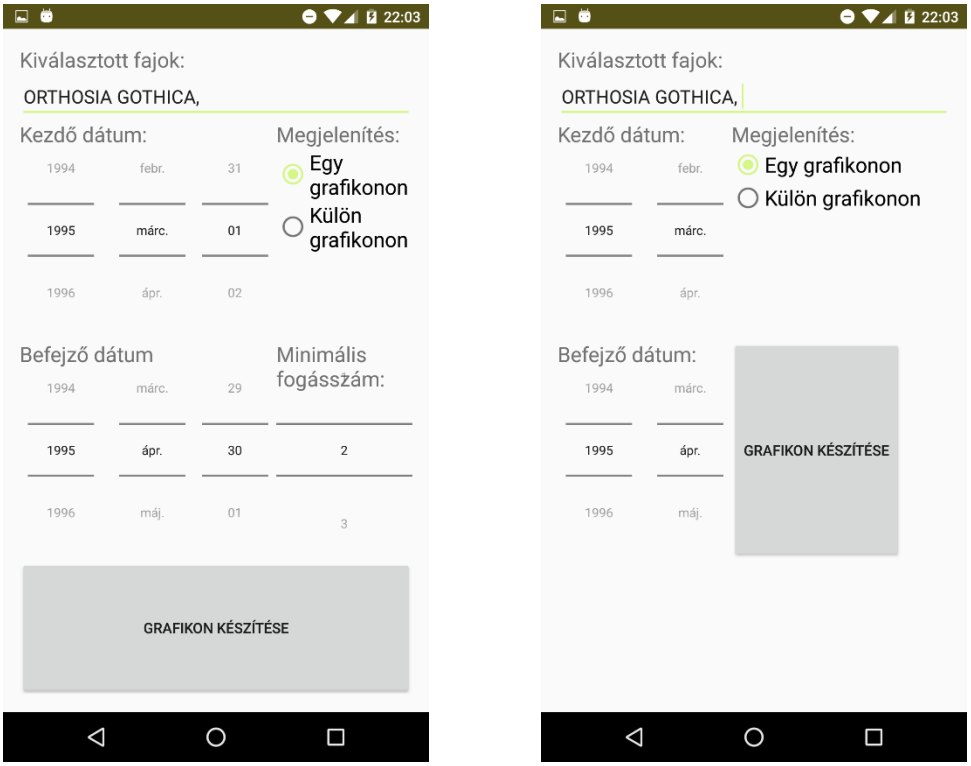
9. ábra - A MainActivity a prototípus, és a vékony kliens verzióban

Az alkalmazásban ugyanaz a *MapsActivity* komponens látja el mindkét megjelenítő funkciót. A felhasználónak a vékonykliens verzióban lehetősége van az internetre csatlakozni egy gomb segítségével, továbbá választhat magyar, angol, vagy német nyelvek közül. A meteorológiai térkép felől nem érhető el a további alkalmazásfunkció a felhasználó számára. A rovarcsapdák esetében a térképen lévő markerek kattinthatók lesznek, és felugrik egy dialógus ablak, amiben a felhasználó kiválaszthatja, hogy napi szintű adatmegjelenítést, vagy havi összegzést kér az adott csapda adatairól. A szenzorok elhelyezkedése a térképen a 10. ábrán látható.



10. ábra - A rovarcsapdák, és a meteorológiai mérőhelyek a térképen

Ezután egy mindkét esetben egy a *SelectTrapPlaceAdapterActivity* osztályból származó képernyő komponens jelenik meg. Napi megjelenítés esetében a *SelectTrapPlaceDayActivity* nevű *Activity-re* kerül a vezérlés. Itt lehetősége van a felhasználónak kiválasztani a megjelenítendő fajokat. Ebből a program legfeljebb ötöt támogat a prototípusban, a vékony kliens verzióban csak hármat. A felhasználó beállíthatja a kezdő, és a befejező dátumot, amik között szűrni szeretne. Illetve megadhat fogási limitet, ekkor csak azok az a napok jelennek meg a grafikonon, amikor az adott fajból többet fogtak be a limitnél az. Illetve egy rádiógomb segítségével be lehet állítani, hogy egy vagy külön grafikonon jelenjelek meg a fajok adatai. A havi menü választása esetén is egy hasonló képernyő jelenik meg, amit a *SelectTrapPlaceMonthActivity* nevű osztály valósít meg. Itt is hasonlóan meg kell adni a fajneveket, illetve a kezdő, és befejező dátumokat, viszont ebben az esetben nem dolgozik a program limitekkel. Minden nap minden fogási adata belekerül az aggregációba, és a grafikonon havi bontásban jelennek meg a fogások. Ebben az esetben is lehetőség van egy grafikonos és külön grafikonos megjelenítésre is. A 11. ábrán láthatók a napi, illetve a havi fogás paraméterezésére szolgáló Activity-k.

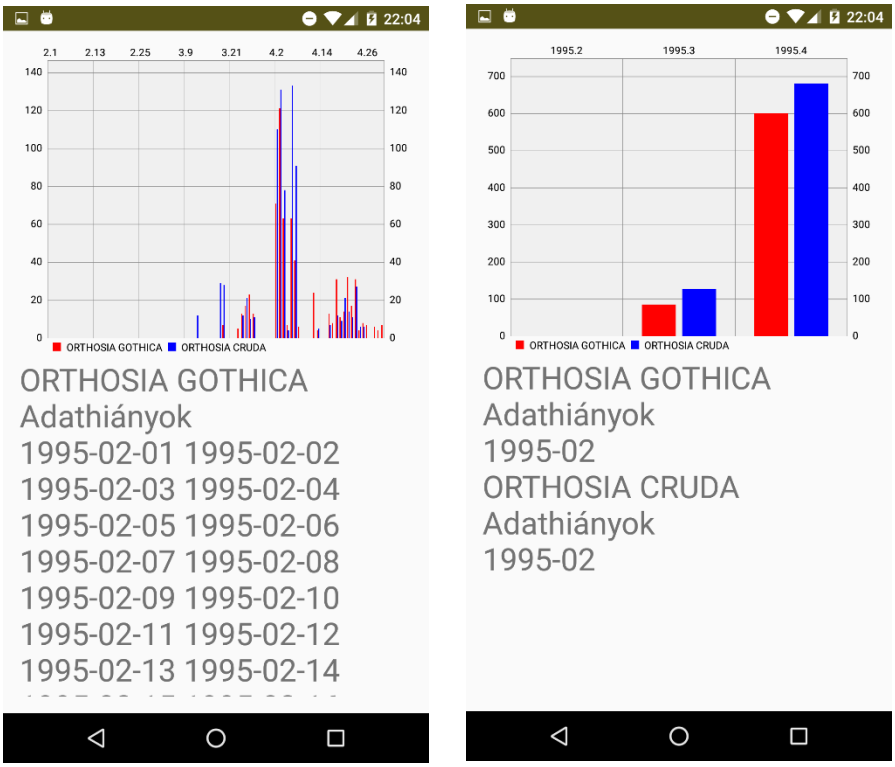


11. ábra - A havi, és a napi nézet a prototípus verzióban

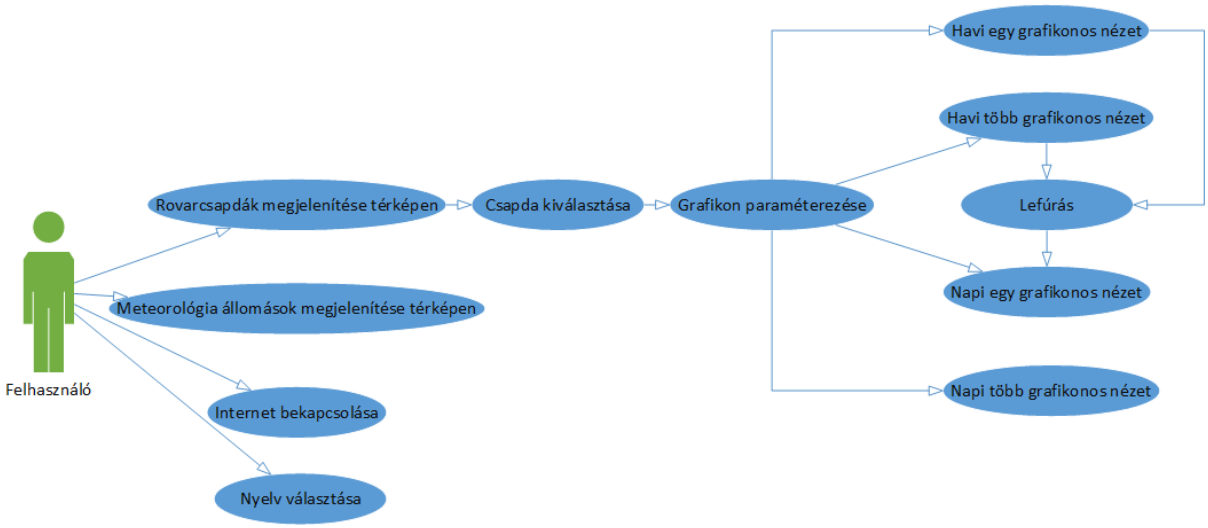
A grafikonok megjelenítésért felelő komponensek a *BarChartAdapterActivity* osztály leszármazottjai. Az implementáció során igyekeztem úgy eljárni, hogy főbb funkciókat egységesen az ősosztály tudja ellátni, és a gyermekosztályoknak csak a konkrét felhasználói felület megvalósítására kelljen koncentrálnia, ilyen elven működnek a grafikonparaméterezésért felelő osztályok is. Az egy grafikonon történő megjelenítésért az *OneBarchartActivity* felel. Itt a képernyő tetején lehet látni oszlopdiagramban az összes faj fogási adatait, napi, vagy havi bontásban. A grafikon alatt a program fajonként felsorolja az adathiányokat tartalmazó rekordokat, illetve napi nézet esetén a limit alatti értékeket. Külön grafikon esetében is fölül jelennek meg a grafikonok a *MoreBarChartActivity* segítségével. A grafikonok közül egyszerre csak egy látható az ablakban, le kell görgetni a többi megtekintéséhez a prototípus verzió esetén. A vékony kliens verzióban az összes grafikon látható egyszerre. A grafikonok alatt ebben az esetben is az adathiányokat, illetve limit alatti értékeket megjelenítő lista található.

A program havi nézet esetében biztosítja a felhasználó számára a lefúrás lehetőségét. Ha a felhasználó rákattint a grafikonon valamelyik havi adatot megjelenítő oszlopra, akkor a program felteszi neki a kérdést, hogy szeretne-e lefúrást végezni. Nemleges válasz esetén az a dialógus eltűnik, és tovább lehet böngészni a grafikont. Igenlő válasz esetén a kiválasztott faj adott hónapjának fogásadatai megjelennek napi nézetben egy *OneBarChartActivity* segítségével.

A 12. ábrán látható két rovarfaj adatainak lekérése egy grafikonon a prototípus verzióban. A 13. ábra mutatja be az alkalmazás felhasználási eseteit, azt hogy melyik képernyőről merre navigálhat tovább a felhasználó.



12. ábra - A napi, és a havi adatok megjelenítése grafikonon a prototípus verzióban



13. ábra - Az alkalmazás felhasználási esetei

5.4. A prototípus verzió

Az alkalmazás két lépcsőben készült el. Először elkészítettem egy prototípust, amely fájlból olvassa be az adatokat. Amikor elkészült a Sensorhub, és a hozzá tartozó lekérdező api, akkor álltam át egy vékony kliens verzió elkészítésére. A két program funkcionalitása nagyjából megegyezik, viszont létezik egy-két kisebb módosítás és funkció, ami csak a vékony kliens verzióba került bele. Ebben a fejezetben a prototípust ismertetem. A vékony kliens verzióról szóló fejezetben pedig az ott bevezetett módosításokat, és újdonságokat fogom bemutatni.

5.4.1. Az alkalmazás indulása

Az *ErtiDateViewer* osztály kiterjeszti az *android.app.Application* osztályt. Ennek köszönhetően az alkalmazás működésére vonatkozó funkciókat felül lehet benne definiálni. Az *onCreate()* metódusa az alkalmazás indulásakor hívódik meg. Ebben a metódusban, így meg lehet adni, hogy mi történjen, amikor elindul az alkalmazás, még mielőtt a felhasználói felület megjelenne. Ebben a metódusban történik a *FiledataLoader* inicializálása, és a fájl adatok betöltése. Az osztály egy statikus gettert, ad a *dataloader-hez,* ezen keresztül éri el a többi osztály az adatokat. Ez az osztály így egyfajta komponenstárként is funkcionál.

A *MainActivity* reprezentálja az alkalmazás kezdő felületét. Ezzel találkozik először a felhasználó. A rendszerrel közölni kell, hogy ez az alkalmazás belépési pontja. Ezt az *AndroidManifest.xml-ben* lehet megtenni *Intent-filterek* segítségével.

```
<activity
  android:name=".activities.MainActivity"
  android:configChanges="orientation">
  <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

A *MainActivity* ebben a verzióban két gombot tartalmaz. Az egyikkel a rovarcsapdákat, a másikkal a meteorológiai mérő helyeket lehet térképen megjeleníteni. Mivel ugyanaz az *Activity* felel mindkét adattípus megjelenítéséért, ezrét egy intent-ben át kell adni, hogy melyiket szeretnénk a két típus közül listázni. Majd ezután indítható el az új *Activity*.

```
trapPlacesButton.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
      Intent intent = new Intent();
      intent.putExtra("type", DataLoader.TRAPPLACE);
      intent.setClass(MainActivity.this, SensorPlaceMapsActivity.class);
      startActivity(intent);
  }
});
```

5.4.2. A térkép megjelenítése

A *SensorPalceMapsActivity* felel a térkép megjelenítéséért, és helyezi el a térképen a szenzorokat. Ehhez a *Google-Maps API*-t használja fel. Ahhoz, hogy használhassuk ezt a szolgáltatást a https://console.developers.google.com oldalon regisztrálni kell az alkalmazásunkat. Itt kapunk egy API kulcsot, amit be kell illesztenünk az AndroidManifest.xml állományba. Ezután a térkép megjelenítés használhatóvá válik.

```
<meta-data
  android:name="com.google.android.geo.API_KEY"
  android:value="@string/google_maps_key" />
```

A *SensorPlaceMapsActivity* ősosztálya *FragmentActivity*, ennek köszönhetően egy fragmens segítségével tudja a térképet megjeleníteni. A térkép betöltése egy *FragmensManager* példány segítségével történik meg, erőforrás azonosító alapján az *Activity onCreate* metúdusában.

```
SupportMapFragment mapFragment = (SupportMapFragment)
getSupportFragmentManager().findFragmentById(R.id.sensoprplacemap);
```

Ebben a metódusban történik meg az adatok lekérése is, a kiválasztott típusnak megfelelően. Ha típus értéke egí, akkor a rovarcsapdák, ha értéke kettő, akkor a meteorológiai mérőpontok jelennek meg a térképen.

Az osztály megvalósít két interfészt is, az egyik az *OnMapReadyCallback*, aminek az *onMapReady(GoogleMap googlemap)* metódusa akkor fut le, amikor a térkép már megjelent, és lehet rá rajzolni. Ebben a metódusban kerülnek fel GPS koordináták a szenzorokat megjelenítő *Markerek* a térképre. Ha rovarcsapdáról van szó, akkor a *Marker* címének beállítjuuk a csapda települését, különben a Marker nem kap címet. A meteorológiai állomások megjelenítésénél a használati eset itt véget is ér, nem érhetőek el további funkciók. A metódus végén a térkép közepe Magyarország földrajzi középpontjának, Pusztavacsnak a koordinátáit veszi fel, és ráközelít Magyarországra. Ebben az esetben kisebb kijelzős készülékeken a legkeletibb csapda nem fér rá a kijelzőre, ezért a vékony kliens verzió, egy keletebbre eső középpontot használ.

A rovarcsapdák megjelenítése esetén a *Marker-ekre* kattintva lehet lekérdezni a csapdák fogási adatait. Ennek megvalósítására szolgál a *GoogleMap.OnMarkerClickListener* interfész és annak az *onMarkerClick(Marker marker)* metódusa. A metódus elején egy *Toast* üzenetben megjelenik a csapda települése. Ezután egy *AlertDialog* épül fel, amivel ki lehet választani, hogy napi megjelenítést, vagy havi összegzést kérünk a csapda adatairól. A kiválasztott elemtől függően elidul a *SelecTrapPlaceDayActivity*, vagy a *SelectTrapplaceMonthActivity*, ahol fel lehet paraméterezni a megjelenítendő grafikonokat.

```
AlertDialog.Builder builder = new AlertDialog.Builder(context);
builder.setTitle("Válassz ki az agrregáció típusát");
builder.setItems(options, new DialogInterface.OnClickListener() {
  @Override
  public void onClick(DialogInterface dialog, int which) {
      if (which == 0) {
          Intent intent = new Intent();
          intent.putExtra("trapplacecity", marker.getTitle());
          intent.setClass(context, SelectTrapPlaceDayActivity.class);
          startActivity(intent);
      }
      if (which == 1) {
          Intent intent = new Intent();
          intent.putExtra("trapplacecity", marker.getTitle());
          intent.setClass(context, SelectTrapPlaceMonthActivity.class);
          startActivity(intent);
      }
  }
});
AlertDialog dialog = builder.create();
dialog.show();
```

5.4.3. A grafikonok paraméterezése

Miután a felhasználó kiválasztott egy csapdát, az után meg kell adnia, hogy az adott csapdából milyen adatokat szeretne lekérdezni. A grafikonok paraméterezésért három osztály felel. A *SelecTrapPlaceAbstractActivity* terjeszti ki az *Activity* osztályt. Saját XML erőforrás leíróval nem rendelkezik, mert a felületet a két gyermekosztálya valósítja meg az aggregáció típusának megfelelően, ez az osztály azokat a funkciókat fogja össze, amelyeket a két gyermekosztály azonos, vagy hasonló módon valósít meg.

Ha minden nap adatait le szeretné kérni a felhasználó, akkor a *SelectTrapPlaceDayActivity* jelenik meg. Ez az osztály csak a felhasználói felület megjelenítéséért, illetve az egyes view elemekben lévő a felhasználó által bevitt adatok kinyeréséért felel. A validáció feladatát a *SelectTrapPlaceAbstractActivity* látja el.

Az osztály *oncreate(Bundle savedInstanceState)* metódusában először megtörténik a felület leíró erőforrás elkérése. Ezután az átadott intentből kiveszi a program a kiválasztott település nevét. Majd megtörténik a fajokat tartalmazó tömb lekérése. A metódus hátralévő nagyobb részében pedig a *view* elemek inicializálása, illetve eseménykezelőjük beállítása történik meg.

A felhasználói felület tetején a megjelenítendő fajok kiválasztására van lehetőség. Ezt a funkciót az alkalmazás egy *MultiAutoCompleteTextView* segítségével oldja meg. Ha a felhasználó elkezdi begépelni a kiválasztott faj nevét, akkor egy lista alapján felajánlja neki az így kezdődő elemeket egy listában, amiből a felhasználó választhat. A multi előtag itt arra utal, hogy több elem bevitelét is támogatja szemben egy hagyományos *AutoCompletTextView* objektummal. A *MultiAutoCompleteTextView* adapterként az összes faj listáját használja, elválasztó karakterként pedig vesszőt. Ezeknek a beállításáért a következő sorok felelnek:

```
multiAutoCompleteTextView = (MultiAutoCompleteTextView)
findViewById(R.id.multiAutoCompleteTextView);
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
R.layout.multi_auto_complete_text_view, spices_array);
multiAutoCompleteTextView.setAdapter(adapter);
multiAutoCompleteTextView.setTokenizer(new
MultiAutoCompleteTextView.CommaTokenizer());
```

Ezután történik a kezdődátum beállításáért felelő *DatePicker* beállítása. Mivel a legtöbb mérésnek a legkorábbi dátuma 1995.03.01, ezért ezt állítottam be rajta alapértelmezett dátumként, meg kell jegyezni, hogy a Java nullától számozza a hónapokat, így itt eggyel alacsonyabb értéket kell beállítani. A *DatePicker* támogat *Calendar*, illetve *Picker* nézetet is. Mivel nem minden mobil készüléken elég a hely, ezért ebben az esetben a *picker* mód a támogatott. Az adatok betöltése a *Datepicker-ről* egy *OndateCangeListener* segítségével történik. Ez az interfész egy metódust definiál, ez az*: onDateChanged(DatePicker view, int year, int monthOfYear, int dayOfMonth)*. Ez a metódus akkor fut le, ha a felhasználó megprögeti a datepicker-t, ilyenkor az *Activity beginyear* tagváltozója felveszi a year paraméter értékét, a *beginmonth* a korrekció miatt a *monthOfYear* értékénél eggyel nagyobbat vesz fel, a *beginday* pedig egyenlő lesz a *dayOfMonth* paraméterrel.

Hasonló elven működik az időintervallum végét beállító *datePickerEnd* nevű változó is. Ez kezdőértékként 1995.04.30-at vesz fel.

Ezután egy *NumberPicker* segítségével lehet beállítani egy limit értéket. Ha egy nap a fogás értéke nem éri el a limit értékét, akkor az nem fog megjelenni a grafikonon. A *NumberPicker* értékének a változását egy *OnValueChangeListener* figyeli. Az interfész megvalósít egy *onValueChange(NumberPicker picker, int oldVal, int newVal)* metódust, amely a *Picker* tekerésekor fut le. A *newVal* paraméter lesz az Activity limit tagváltozójának az értéke.

A felhasználónak lehetősége van kiválasztani, hogy egy grafikonon szeretné az összes rovarfaj adatait látni, vagy mindegyiket külön-külön grafikonon. Erre egy *RadioGroup* szolgál. Alapértelmezetten az egy grafikonos verzió van kiválasztva a *radioButton.toggle()* metódus meghívásával. A *RadioGroup* értékének lekérése egy *OnChangeListener-rel* történik. Ha az első gomb, van kiválasztva, akkor az activity graficontype változója igaz értéket vesz fel, és egy grafikonos megjelenítést használunk. Ha a második van kiválasztva, akkor a logikai változó értéke hamis, és külön grafikonos változatot használunk.

```
radioGroup.setOnCheckedChangeListener(new
RadioGroup.OnCheckedChangeListener() {
  @Override
  public void onCheckedChanged(RadioGroup group, int checkedId) {
      if (checkedId == radioButton.getId()) {
          graficontype = true;
      }
      if (checkedId == radioButton1.getId()) {
          graficontype = false;
      }
  }
});
```

A metódus végén a *ChartButton* nevű gomb beállítása, és eseménykezelője található. A gomb megnyomásának a hatására meghívódik a *SelectTrapPlaceAbstractActivity makeDiagram* metódusa, az első paraméter a grafikon típusát jelenti. Mivel az alkalmazás csak oszlopdiagrammot támogat, ezért ennek értéke mindig bar. A második paraméter az aggregáció típusát jelentő integer paraméter, ami jelen esetben *SelectTrapPlaceAbstractActivity.DAY,* tehát nulla értéket vesz fel. Mivel a *SelecTrapPlaceDayActivity* a *SelectTrapPlaceAbstractActivity* tagváltozóit használja, ezt nem kell átadni paraméterként a grafikon kirajzolásához.

A felhasználónak lehetősége van havi összegzést kérnie a kiválasztott fajokról. Ha ezt szeretné, akkor a *SelectTrapPlaceMonthActivity* jelenik meg a térkép nézet után. Ez hasonlóan működik a napi nézethez. A fajok kiválasztása, illetve a grafikon típusválasztás ugyanúgy történik meg

mind a két nézet esetében. Itt a dátum kiválasztása másképp működik, mert csak évet és hónapot kell megadni egy *Datepicker* segítségével. Az androidos *DatePicker LinearLayout-okból* épül fel. Ennek köszönhetően le lehet vágni belőle a nap beállításáért felelő *NumberPicker-t.*

```
datePickerBegin = (DatePicker) findViewById(R.id.datePickerBegin);
LinearLayout pickerParentLayout = (LinearLayout)
datePickerBegin.getChildAt(0);
LinearLayout pickerSpinnersHolder = (LinearLayout)
pickerParentLayout.getChildAt(0);
NumberPicker picker = (NumberPicker) pickerSpinnersHolder.getChildAt(2);
picker.setVisibility(View.GONE);
datePickerBegin.init(beginyear, beginmonth, 0, new
DatePicker.OnDateChangedListener() {
          @Override
          public void onDateChanged(DatePicker view, int year,
          int monthOfYear, int dayOfMonth) {
              beginyear = year;
              beginmonth = monthOfYear + 1;
          }
      }
);
```

Itt is hasonló módon a napi nézethez, korrigálni kell eggyel a hónap értéket. Ebben az esetben is a *SelectTrapPlaceActivity* tagváltozóit állítja be a megfelelő metódus, mind a kezdő, mind a befejező dátum esetén. A gomb eseménykezelőjében is a *makeDiagram* metódus hívódik meg, egy "bar", illetve egy *SelectTrapPlaceAbstractActivity.MONTH* azaz egy értékű paraméterrel.

A *SelectTrapPlaceAbstractActivity* felel a két gyermekosztály által szolgáltatott adatok feldolgozásáért validálásáért, illetve átadásáért a grafikonrajzoló *Activity* osztályoknak. Az osztály slice metódusa szedi szét fajnevekre a *MultiCompleteTextView-ban* lévő szöveget vesszők mentén, és ebből egy *String* generikusú *ArrayList-et* készít.

A *makeDiagram* metódus dolgozza fel a felhasználó által megadott paramétereket. Először ellenőrzi, hogy nem haladja-e meg a bevitt fajok száma az ötöt, ha igen, akkor *Toast* üzenetet dob fel, amiben közli a felhasználóval a hibát.

Ha a fajlista hossza nem nulla, akkor egy ciklus végig iterál a fajlistán. És a fájladatbázishoz fordul, hogy lekérdezze onnan a paramétereknek megfelelő fogásokat.

```
if(aggregationtype==DAY) {
      insectTraps=((ArrayList) ErtiDataViewer.getDataLoader().
      getInsectTrapBetweenDatesBySpeciesAndPlace(
      beginyear, beginmonth, beginday, endyear, endmonth, endday,
      speciesList.get(i), trapplacecity, FileDataLoader.DAY));
}
if(aggregationtype==MONTH){
      insectTraps = (ArrayList) ErtiDataViewer.getDataLoader().
      getInsectTrapBetweenDatesBySpeciesAndPlace(
      beginyear,beginmonth,1,
      endyear,endmonth,31,speciesList.get(i),trapplacecity,
      FileDataLoader.MONTH);
}
```

Eztuán ellenőrzi, hogy van-e értékes, adathiányt nem tartalmazó rekord a visszaadott adathalmazban. Ha talál ilyen rekordot, akkor az osztály *ArrayList<ArrayList> obserovedObjectCollections* nevű listájához, hozzáadja a visszaadott listát, mint egy elemet. Ha a felhasználó nem adott meg fajokat, akkor egy *Toast* üzenetben figyelmezteti a program erre

a hibára, és ekkor sem történik grafikonrajzolás. Ezután a program ellenőrzi, hogy van-e olyan grafikon, amelyik kirajzolható, tehát, hogy nem üres-e a listákat tartalmazó lista. Ha a lista üres, akkor figyelmezteti a felhasználót a hibára egy *Toast* üzenetben. Ha a listának van értékes eleme, akkor egy intentbe beleteszi a listákat tartalmazó listát, a limit értékét, illetve az aggregáció típusát.

Most következik a grafikon típus ellenőrzése. Ha a graficontype paraméter értéke igaz, akkor egy grafikonra kell az összes fajt kirajzolni. Ehhez elindul egy *OneBarChartActivity*. Ha a grafikontype értéke hamis, akkor először ellenőrizni kell, hogy van-e, egynél több grafikon, ha nincs, akkor *OneBarChartActivity* indul el ebben az esetben is. Ha a grafikonok száma meghaladja az egyet, akkor *MoreBarChartActivity* indul el, ami külön grafikonokra rajzolja ki az egyes fajok adatait. A tesztelés során gondot jelentett, hogy ugyanazt a lekérést lefuttatva mindig az előzőhöz hozzáadódott az új értéke, és a sokadik lekérések már fals eredményeket mutathattak. Ezt a hibát azzal küszöböltem ki, hogy a metódus a végén törli a listákat tartalmazó lista tartalmát.

5.4.4. A grafikonok rajzolása

A grafikonok rajzolására az Android-ban nincsen beépített megoldás. Viszont létezik több nyílt forráskódú eszközkönyvtár is erre a célra. Én ezek közül az *MPAndroidChart* nevű megoldást választottam. Ez több fajta grafikon típust is támogat, például kördiagramot, oszlopdiagramot és vonaldiagramot is. A félév elején a vonaldiagrammal kísérleteztem, de ezzel a megoldással az adathiányok lekezelése nehézkessé vált. A félév végére áttértem tehát az oszlopdiagramok használatára. Ahhoz, hogy az *MPAndroidChart* [6] eszközkönyvtárat használni tudjam le kellett töltenem egy *mpandroidchartlibrary-2-1-6.jar* nevű állományt, majd bemásolni az alkalmazáson belül az */app/libs* mappába, és az Android Studio-ban hozzá kellett adnom a *buid-path-hez.*

A grafikonok rajzolását is három osztály végzi el hasonlóan a paraméterezéshez. A *BarchartAbstractActivity* osztály az *Activity* gyermekosztálya, viszont XML felület leíróval nem rendelkezik. Ez az osztály végzi el a grafikonok elkészítéséhez szükséges előfeldolgozásokat, itt valósul meg az adathiányok listázása, illetve havi nézet esetén a lefúrás a napi nézetbe.

A *OneBarChartActivity*, és a *MoreBarchartActivity* a *BarChartAbstractActivity* osztályból öröklődik. Ez a két osztály csak a felhasználói felület létrehozásáért felel, minden más funkciót delegál az ősosztály felé.

Az *MPAndroidChart* eszközkönyvtár több osztályt felhasznál az oszlopdiagramok készítéséhez, most ezeket szeretném ismertetni. A *BarChart* osztály több leszármazottja a *View* osztálynak, így ha oszlopdiagramot szeretnénk rajzolni a programmal, akkor egy ilyen elemet kell belerakni az *Activity* erőforrás leíró *XML* fájljába.

```
<com.github.mikephil.charting.charts.BarChart
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:id="@+id/chart"
    />
```

A többi osztály objektumai mind egy *BarChart* objektumnak lesznek a komponensei. A *BarData* objektumok reprezentálnak minden olyan adatot, ami felkerül a grafikonra. Jelen esetben a vízszintes tengelyen lévő feliratokat tartalmazó lista, illetve egy vagy több

BarDataSet objektum. Egy *BarDataSet* objektum írja le az egy adatsorhoz tartozó oszlopok adatait. Ha egy grafikonra több adatsort szeretnénk felrajzolni, akkor több *BarDataSet-et,* is hozzáadhatunk. Ez történik a *OneBarchartActivity-ben*, míg a *MoreBarChartActivity-ben* csak egy *Dataset*-et tartalmaz egy grafikon. A BarDataSet-en belül egy-egy oszlopot egy-egy *BarEntry* objektum testesít meg. A *BarEntry* konstruktora két paramétert vár, az első egy float érték, ez határozza meg az oszlop magasságát, értékét, a második paraméter egy integer érték, ez pedig az oszlop indexe.

A *OneBarChartActivity* egy grafikonra rajzolja ki az összes fajnak az adatait. Az *oncreate(Bundle savedInstanceState)* metódus elején egy intentből elkéri a listákat tartalmazó listát, amely a fogási adatsorokat tartalmazza. A minimális fogás értékét, az összesítés típusát, illetve lefúrás esetén fontos egy colorindex lekérése is, hogy a lefúrt grafikon színei megegyezzenek a fúrás előtti adatsor színeivel.

```
final Intent intent = getIntent();
observedObjectCollections = (ArrayList<Collection>)
intent.getSerializableExtra("observedobjectcollections");
limit = intent.getIntExtra("limit", 0);
aggregationtype = intent.getIntExtra("aggregationtype",0);
int colorindex = intent.getIntExtra("colorindex",0);
```

Ezután a metódus végigmegy az összes fogási listán, és létrehozza hozzá a megfelelő *BarDataSet* objektumokat. A cikluson belül abban az esetben, ha lefúrásról van szó, akkor a havi bontásban használt színnel azonos színt állít be a grafikonhoz, egyébként a colors nevű szín konstansokat tartalmazó tömb aktuális elemét állítja be színnek.

```
ArrayList<BarDataSet> dataSets = new ArrayList<>();
for (int i = 0; i< observedObjectCollections.size(); i++){
  BarDataSet dataSet = getDataSet(((ArrayList)
  observedObjectCollections.get(i)));
  if(colorindex!=0){
      dataSet.setColor(getColors().get(colorindex));
  } else {
      dataSet.setColor(getColors().get(i));
  }
  dataSets.add(dataSet);
}
```

Ezután még egyszer végig iterál a listán, és lekéri az X tengelyhez tartozó feliratokat. A felirat listákból, és az adathalmazokból létrehozza a *BarData* objektumot, majd a grafikonhoz. A metódus legvégén beállítja az adathiányok kiírására szolgáló *textView-t*, és kiírja rá az esetleges adathiányokat, és limit alatti értékeket, az ősosztály *writeNA* metódusának a meghívásával.

A *MoreBarChartActivity* osztály *oncreate(Bundle savedInstaceState)* metódusának elején szintén a fajlisták a limit és az aggregáció típus lekérésével kezdődik. Mivel a lefúrás célja nem ez lehet ez az *Activity*, mivel mindig egy grafikon keletkezik a fúrás után, ezért nem kerül elkérésre a colorindex paraméter.

Az intentből származó adatok elkérése után lefut a *createCharts()* metódus, amely egy *BarChart* generikusú *ArrayList-tel* tér vissza. Ez a metódus végig iterál az *observedObjectCollections* listán, amely a fogási adatsorokat tartalmazza. A ciklus elején létrehoz egy *BarChart* objektumot, majd létrehoz egy *BarDataSet-et,* aminek az ősosztály *getDataSet* metódusának a meghívásával ad értéket. Majd beállítja a szín lista aktuális elemét az adatsor színének. Létrehoz egy *BarData* objektumot, aminek a konstruktorában lekéri az X-

tengely feliratait, és átadja neki a *BarDataSet-et* is, és ezt állítja be a *BarChart* adatforrásának. Ha a napnál hosszabb időszakról készült az aktuális grafikon, akkor le lehet fúrni napi nézetbe. A *drillDown* első paramétere a kiválasztott grafikon, a második egy integer paraméter, ami a grafikon pozíciója a listában. A ciklus végén a grafikon listához hozzáadódik a grafikon. A metódus végül visszaadja a grafikonok listáját.

```
private ArrayList<BarChart> createCharts() {
  ArrayList<BarChart> charts = new ArrayList<>();
  for (int i = 0; i < observedObjectCollections.size(); i++) {
      BarChart chart = new BarChart(getApplicationContext());
      BarDataSet dataSet = getDataSet((ArrayList)
      observedObjectCollections.get(i));
      dataSet.setColor(getColors().get(i));
      BarData barData = new BarData(getXValues(((ArrayList)
      observedObjectCollections.get(i)),aggregationtype), dataSet);
      chart.setDescription("");
      chart.setData(barData);
      if(aggregationtype>DAY){
          super.drillDown(chart, i);
      }
      charts.add(chart);
  }
  return charts;
}
```

A grafikonok egy *ListView-ban* foglalnak helyet, aminek az elemei az egyes grafikonok. Ahhoz, hogy a grafikonok meg tudjanak jelenni a listanézetben, szükség van egy adapterosztályra. Itt ezt a funkciót a *ChartAdapter* osztály látja el.

```
ArrayList<Chart> charts = (ArrayList) createCharts();
ListView listView = (ListView) findViewById(R.id.chartList);
ChartAdapter adapter = new ChartAdapter(getApplicationContext(),
R.layout.graficon_bar_row, charts);
listView.setAdapter(adapter);
```

A *ChartAdater* osztály az *ArrayAdapter* gyermekosztálya. Négy metódust kell megvalósítani. a public int *getCount()* metódus visszaadja a grafikonok számát, a *public Object getItem(int position)* metódus visszaadja a pozíciónak megfelelő grafikont a listából. A *public long getItemId(int position)* visszadja long típusúként a paraméterként kapott integer értéket.

A *getView* metódus veszi a grafikonlista pozíciónak megfelelő elemét, beállítja a magasságát 300 dp-re, majd visszatér a grafikonnal.

A konstruktorban, át kell adni neki a megfelelő kontextust, egy *Layout* elrendezés azonosítót, illetve a grafikonokat tartalmazó listát.

A ListView beállítása után történik az adathiányokat tartalmazó *TextView* beállítása, illetve az adathiányok és limit alatti értékek kiírása rá az ősosztály *writeNaData* metódusának a meghívásával.

Érdemes még pár szót szólni az *Activity-hez* tartozó XML fájl felépítéséről. Azt szerettem volna elérni, hogy a grafikonok is görgethetőek legyenek egy kisebb ablakban, illetve az egész képernyő tartalma legörgethető legyen. Ezt úgy sikerült megoldanom, hogy a grafikont tartalmazó listát, és az alatta lévő szövegmezőt is egy *ScrollView-ba* ágyaztam bele. Ha a

felhasználó a grafikonon görget, akkor le fel tud lapozni az egyes grafikonok között, ha pedig kívül görget, akkor az *Activity* teljes tartalmát tudja fel-le görgetni.

```
<ScrollView
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:id="@+id/scrollview">
  <RelativeLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:id="@+id/relativelayout">
      <ListView
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:id="@+id/chartList"></ListView>
      <TextView
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:id="@+id/naTextViewMoreChart"
          android:textSize="30dp"
          android:scrollbars="horizontal"
          android:layout_below="@+id/chartList"/>
  </RelativeLayout>
</ScrollView>
```

A *BarChartAbstractActivity* végzi a grafikonok elkészítését a rovarfogás adatokból, illetve itt található az adathiányok kiírásáért felelő *writeNaData* metódus, és a lefúrást végző drilldown metódus is.

A grafikonok felépítése a legkisebb komponensek felől halad az egyre nagyobbak felé. Ezért először az *addEntries(ArrayList<ObservedObject> observedObjects)* metódus fut le, amelyik a grafikon oszlopait hozza létre a rovarfogás adatokból. A metódus végig iterál a fogásokat tartalmazó listán, ellenőrzi, hogy az adott elemhez tartozó fogás mennyiség nagyobb-e a megadott limitnél, ha igen, akkor hozzáadja az entries listához, ha nem akkor nullát ad a listához hozzá. Erre azért van szükség, hogy az adatsorok hosszai mindig egyezőek legyenek

```
for (int i = 0; i < observedObjects.size(); i++) {
  InsectTrap insectTrap = (InsectTrap) observedObjects.get(i);
  if (insectTrap.getCatches() > limit) {
      BarEntry entry = new BarEntry(insectTrap.getCatches(), i);
      entries.add(entry);
  } else {
      BarEntry entry = new BarEntry(0, i);
      entries.add(entry);
  }
```

A *getXValues(ArrayList<ObservedObject> observedObjects, int aggregationtype)* metódus hozza létre az x-tengely feliratait, és tárolja el egy *String* alapú listában. Ha napi adatokról van szó, akkor az adott elemhez tartozó felirat az elem hónap és nap attribútuma lesz, ha havi bontásról van szó, akkor az adott elem év és hónap attribútuma lesz.

```
for (int i = 0; i < observedObjects.size(); i++) {
  if (aggregationtype == DAY) {
      values.add(observedObjects.get(i).getMonth() + "." +
      observedObjects.get(i).getDay());
  }
  if (aggregationtype == MONTH) {
      values.add(observedObjects.get(i).getYear() + "." +
      observedObjects.get(i).getMonth());
  }
}
```

A *getDataSet(ArrayList<ObservedObject> observedObjects)* metódus hozza létre a teljes oszlop halmazt, az oszlopokat tartalmazó listából, ezt a listát az *addEntries* metódus meghívásával állítja elő. Ellenőrzi, hogy a visszaadott lista tartalmaz-e elemeket, és ha igen akkor létrehozza az adathalmazt, és feliratnak a rovarfogás lista első elemének fajnevét állítja be. A *dataset.setDrawValues(false)* függvényhívással beállítja, hogy az oszlopok tetején ne jelenjenek meg a fogásértékek. Ezután visszatért a *dataSet-tel.*

```
ArrayList<BarEntry> entries = addEntries(observedObjects);
if (entries.size() != 0) {
  dataSet = new BarDataSet(entries,
  ((InsectTrap) observedObjects.get(0)).getSpeciesName());
}
dataSet.setDrawValues(false);
return dataSet;
```

A *writeNaData()* metódus arra szolgál, hogy a felhasználót tájékoztassa arról, hogy bizonyos napokhoz rendelt értékek miért nullát vesznek fel. Három okból lehet nulla egy oszlop értéke, egyrészt azért, mert nem volt ott fogás az adott napon, a kiválasztott fajból másrészt azért, mert a fogásszám nem haladta meg a limit értéket, a harmadik lehetőség pedig, hogy adathiány van az adott napon. A *writeNaData()* azokat a napokat vagy hónapokat sorolja fel, amelyek az utóbbi két kategóriába esnek. Havi összeg esetében a limitek nem jönnek a képbe. A metódus végig iterál a listán, és eldönti, hogy tartalmaz-e adathiányt, majd egy második ciklussal eldönti, hogy vannak-e limit alatti értékek. Ha vannak adathiányok, akkor szóközzel elválasztva feltünteti őket, napi nézetben, kiírja a teljes dátumot, havi nézetben pedig csak az év hónap attribútumokat írja ki.

A limit alatti értékeket csak napi nézet esetén jeleníti meg, ha szerepelnek alacsony értékek a listában. Végig iterál a listán, és a limit alatti fogásoknak kiírja a dátumát, és hogy hány fogás volt aznap.

A *drillDown(final Chart chart, final int chartindex, final int graficontype)* metódus első paramétere a grafikon, amelyikről le akarunk fúrni, a második pedig a kiválasztott grafikon indexe. A második paraméterre azért van szükség, hogy a lefúrással létrejött grafikon oszlopainak színei megegyezzenek az eredeti grafikon oszlopainak színeivel. A harmadik paraméter megadja, hogy a metódust a *OneBarchartActivity-ről*, vagy a *MoreBarCahrtActivity-ről* hívjuk éppen. Ahhoz, hogy a grafikon egy oszlopát ki tudjuk választani a grafikonhoz egy *OnChartValueSelectedListenert* kell rendeli. Ennek az egyik metódusa az *onNothingSelected*, ez a metódus csak üres implementációval rendelkezik. A másik metódus az *onValueSelected(Entry entry, int i, Highlight highlight)*, ebből az *Entry* a kiválasztott oszlop, az int az adatsor indexe, a *Highligt-tal* pedig a kiválasztott oszlop szín kiemelését lehet szabályozni. A lefúrás csak olyan oszlopra van értelmezve, aminek az értéke nem nulla. Ezért a metódus először ezt ellenőrzi. Utána elkéri a kiválasztott oszlophoz tartozó fajlistát. Egy

grafikonos listánál ez az adatsor indexe, külön grafikonos nézet esetén pedig a grafikon indexe alapján történik.

```
if(graficontype == ONECHART){
  insectTraps = (ArrayList<InsectTrap>) observedObjectCollections.get(i);
}
if (graficontype == MORECHART){
  insectTraps =
  (ArrayList<InsectTrap>)observedObjectCollections.get(chartindex);
}
```

Ezután az *Entry* indexének megfelelő *InsectTrap* objektumnak elkéri a fajnév, év, hónap, és település attribútumait. Majd ennek a fajnak az adataival lekéri a kiválasztott hónapot napi bontásban a *dataloader-en* keresztül. Mivel egy fogáslistákat tartalmazó listát kell átadni paraméterként ezért bele kell tenni a fogásokat tartalmazó listát egy ilyen listába. Mivel a kiválasztott hónap minden napjának adataira kíváncsiak vagyunk, ezért a limit értéke természetesen nulla. Ezután létrejön egy drillintent nevű *Intent*, ennek az extra paraméterei lesznek a lista, a limit, illetve az aggregáció típus, ami természetesen napi nézet lesz. Az intentben kell átadni a színnek az indexét, amit a colors tömbben elfoglal. Ez egy grafikonos lista esetén, az adatsor indexe, külön grafikonos nézet esetén a grafikon indexe. Az intent célja egy *OneBarChartActivity* minden esetben. A felhasználó felé egy *AlartDioalog-ot* jelenít meg a rendszer a kattintás hatására. Megkérdezi a felhasználót, hogy szeretne-e lefúrni, közli vele a kiválasztott faj nevét, illetve a kiválasztott időszakot. A felugró *AlertDialog* igen gombjára kattintva megtörténik a lefúrás megjelenik az új *Activity*, a nem gombra kattintva a felhasználó tovább böngészheti a grafikont.

5.5. Áttérés a vékony kliens verzió használatára

A félév első felében egy prototípus verziót készítettem el, ez a verzió fájlból olvassa az adatokat, és a felhasználói felülete sincs optimalizálva. A félév végén ezt a verziót vettem alapul a vékony kliens verzió megalkotásához. Ez a verzió már a Sensorhub-ból olvassa ki a szükséges adatokat, és a felhasználói felülete is kiforottabb. Továbbá támogat még egy pár olyan funkciót, amellyel nem rendelkezik a prototípus. Ilyen funkciók például a német és az angol nyelvű felhasználói felület, illetve a táblagépek képernyőinek magasabb fokú kihasználása. Illetve vannak kisebb apróságok is, mint például előszűrés a fajnevekre település alapján, és több információ megjelenítése a grafikonokról.

5.5.1. A Sensorhub architektúra támogatása

A legfontosabb változtatás a prototípushoz képest az volt, hogy bevezettem a Sensorhub-ot, mint adatforrást. A Sensorhub-hoz történő csatlakozást a *SensorhubDataLoader* osztály végzi el, amely felhasznál egy Pintér Tamás által írt lekérdező API-t. A Sensorhub lekérdezéséhez internet kapcsolatra van szükség. Az Android operációs rendszer nem támogat internet kapcsolatot igénylő műveleteket a grafikus szálon. A prototípus verzió adatlekérő műveletei ezen a szálon történtek meg, a vékony kliens alkalmazásban ezen változtatni kellett. Ezt a változtatást meg lehetett volna oldani a *Thread* osztály példányaival is, viszont az Android kínál erre egy szofisztikáltabb megoldást az *AsyncTask* [1] osztály használatán keresztül. A programozás során három *AsycTask* osztályt vezettem be, most ezeknek a felépítését, és használatát szeretném ismertetni.

Saját *AsyncTask* osztály létrehozásához az *AsycTasck<Params, Progress, Result>* osztályt kell kiterjeszteni. Az osztály három generikussal rendelkezik, mind a három helyre írható tetszőleges objektum típus. Az első helyre írt típus a bemenő paramétereket határozza meg, amivel a *Task* dolgozik. A második paraméter megadja azt a típust, amit a felhasználó számra publikálunk, amíg tölt a *Task*, ilyen lehet például egy egész szám, ami a betöltődés állapotát mutatja meg százalékban. A harmadik helyre írt típus megadja az *AsyncTask* által szolgáltatott érték adattípusát. Az osztálynak öt fontosabb életciklus metódusa van. Az *onPreExecute()* függvény a fő szálon fut le, itt lehet például megjeleníteni egy *ProgressBar* objektumot, amivel meg lehet jeleníteni a *Task* töltésének haladási állapotát. A *doInBackground(Params…)* metódus a háttér szálon fut le, ez az *AsyncTask* osztály legfontosabb metódusa, itt történik meg az adatok lekérdezése, amiért *Task-ot* létrehoztuk. A *pulbishProgress(Progress…)* metódussal lehet a töltési állapotról adatokat közölni a felhasználó felé, az *onProgressUpdate(Progress…)* metódus felel az állapotjelző frissítéséért, ezek a metódusok is a fő szálon futnak. Az *onPostExecute(Result…)* függvény a *doInBackGround* függvény után fut le, és annak visszatérési értéket kapja meg paraméterül. Mivel nagyon gyorsan egy másodpercen belül megjelenik a lekérdezett eredmény a programban, ezért nem tartottam szükségesnek *ProgressBar* létrehozását egyik *Task* osztályomban sem. Mindenütt csak a *doInBackground* metódust definiáltam felül. Az *Activity-k* felől a *task.execute().get()* hívással lehet a *Task* által szolgáltatott eredményt elkérni. A metódus során nem ellenőrzött kivételek léphetnek fel, amiket kötelező lekezelni.

```
SensorPlaceListTask listTask = new SensorPlaceListTask();
try {
  sensorPlaces = (ArrayList) listTask.execute(type).get();
} catch (InterruptedException e) {
  e.printStackTrace();
} catch (ExecutionException e) {
  e.printStackTrace();
}
```

A bevezetett *AsyncTask* osztályok közül a program életciklusában legelőször a *SensorPlaceListTask* kap szerepet. Ez az osztály tölti be a Sensorhub felől a térképen megjelenítendő pontokat. A *Params* típusa *Integer*, ha ennek értéke egy, akkor a rovarcsapdákat, ha kettő, akkor a meteorológiai mérőhelyeket kéri le az osztály, és ezek jelennek meg a térképen, a *SensorPlaceMapsActivity* segítségével. A vékony kliens verzióban a *MainActivity* keretein belül történik meg a szenzorok lekérése, és *Intet-ben* kerül átadásra a *SensorPlacesMapsActivity* számára. A prototípus verzióban a *SensorPlacesMapsActivity* végzi el ezt a lekérést is.

A *SpeciesListTask* bemenő paramétere *String*, a visszatérő értéke *String[]*. A bemenő paraméter a térképen kiválasztott város neve, ezzel hívja meg a dataloader *getSpeciesByCity(String city)* metódusát. A vékony kliens verzió esetében a fajnevek lekérése nem a grafikon paraméterező *Activity-kben* történik meg, hanem már a *SensorPlaceMapsActivity-ben.* Egy Intent-ben adóik át a csapdában valaha fogott fajok listája, vagy ha a lista üres, meg sem jelenik a grafikon paraméterező felület, ez nagyobb biztonságot ad a felhasználónak, az alkalmazás használata közben.

```
@Override
protected String[] doInBackground(String... params) {
  if(ErtiDataViewer.getDataLoader()!=null) {
      String city = params[0];
      ArrayList<String> species =
     (ArrayList)ErtiDataViewer.getDataLoader().getSpeciesByCity(city);
      Collections.sort(species);
      String[] speciesArray = new String[species.size()];
      speciesArray = species.toArray(speciesArray);
      return speciesArray;
  } else {
      return null;
  }
}
```

A harmadik *AssyncTask* osztály az *ObservedObjectListTask*. Ennek a segítségével lehet lekérni a rovarfogásokat a grafikonok paraméterezése után. Ez a *Task* fut le a *SelectTrapPlaceAbstractActivity makeDiagram* metódusában, illetve a *BarChartAbstractActivity drillDown* metódusában.

```
public ObservedObjectListTask(int beginyear, int beginmonth, int beginday,
int endyear, int endmonth, int endday, String species, String city, String
aggregationtype) {
  this.beginyear = beginyear;
  this.beginmonth = beginmonth;
  this.endyear = endyear;
  this.endmonth = endmonth;
  this.beginday = beginday;
  this.species = species;
  this.endday = endday;
  this.city = city;
  this.aggregationtype = aggregationtype;
  ErtiDataViewer.changeDataLoader();
}
```

A fenti konstruktorban látható egy *ErtiDataViewer.changeDataLoader()* függvényhívás. Ez ellenőrzi, hogy rendelkezésre áll-e internet kapcsolat a készüléken. Ha igen, akkor ellenőrzi, hogy van-e már *SensorhubDataLoader* példány inicializálva, és ha nincs, akkor létrehoz egyet. Ha nincs kapcsolat, akkor null-ra állítja a *DataLoader* objektumot. Ez a függvényhívás mindhárom *Task* konstruktorában megtalálható.

Az *AsyncTask* osztályok *doInBackground* metódusai ellenőrzik, hogy null-e a *DataLoder* értéke, ha nem, akkor elvégzik a megfelelő adatlekérést. Ha viszont az érték null, akkor a *doInBackground* metódus is null értékkel tér vissza. Az *AsyncTask-okat* felhasználó osztályok ellenőrzik, hogy az *execute().get()* hívás értéke nem null-e, ha az érték nem null, akkor megjeleníti a lekért adatokat, különben egy *Toast* üzenetben megjelenik, a "Nem sikerült az adatok letöltése, ellenőrizze az internetkapcsolatot felirat."

5.5.2. Különböző képernyő méretű készülékek támogatása

A munkám során a fejlesztést, és a tesztelést saját mobiltelefonomon, egy Nexus 5X típusú készüléken kezdtem el, így a felhasználói felületet is arra terveztem meg. Amikor nekiálltam tesztelni kisebb képernyőjű telefonokon az alkalmazást, akkor kiderült, hogy a *Datepicer-ek* nem minden esetben férnek el megfelelő módon azoknak a képernyőin, ezért szükségessé vált a *SelectTrapPlaceDayActivity*, és a *SelectTrapPlaceMonthActivity* átalakítása. Úgy döntöttem, hogy az *Activity-ken* elhelyezett *DatePicker-eket* csak táblagépeken fogom megjeleníteni, kisebb képernyőjű készülékeken, csak *TextView-kat* jelenítek meg a dátumok bevitelére, és a szövegmezőre kattintva jelenítek meg olyan dialógus ablakokat, amikkel a dátum megadható.

Hogy ezt meg tudjam valósítani minősített felhasználói felület leíró erőforrás fájlokat hoztam létre. A *layout-xlarge* mappában is elhelyeztem egy-egy felület leíró erőforrás fájlt a *DayFragment,* és a *MonthFragment* számára. Az Android operációs rendszer futásidőben választja ki, hogy a minősítés nélküli, vagy a minősített könyvtárban lévő felületet fújja fel.

Az *Activity-k* tetején lévő "Kiválasztott fajok:" felirat, és az alatta lévő *MultiAutoCompleteTextView* megegyezik, ezért csak ennyi került az *Activity*-k felület leíró fájljaikba. A felület többé részét dinamikusan helyezi fel a program *Fragment-ek* [7] segítségével. A Fragment elhelyezéséhez egy *FragmentManager* objektumra van szükség. A *Fragment* elkérése egy *FragmantTransaction* keretén belül zajlik le, a tranzakció indítása a *manager.beginTransaction()* függvényhívással történik. A *FragmantManager* osztály tudja *Tag* alapján elkérni a rendszertől a megfelelő *Fragment-et*. A *Fragment* elkérése után ellenőrizni kell, hogy a visszakapott *Fragment* értéke nem null-e, mert ha igen, akkor létre kell egyet hozni az osztály konstruktorával. Ezután a *transaction.add(R.id.frame_layout,*

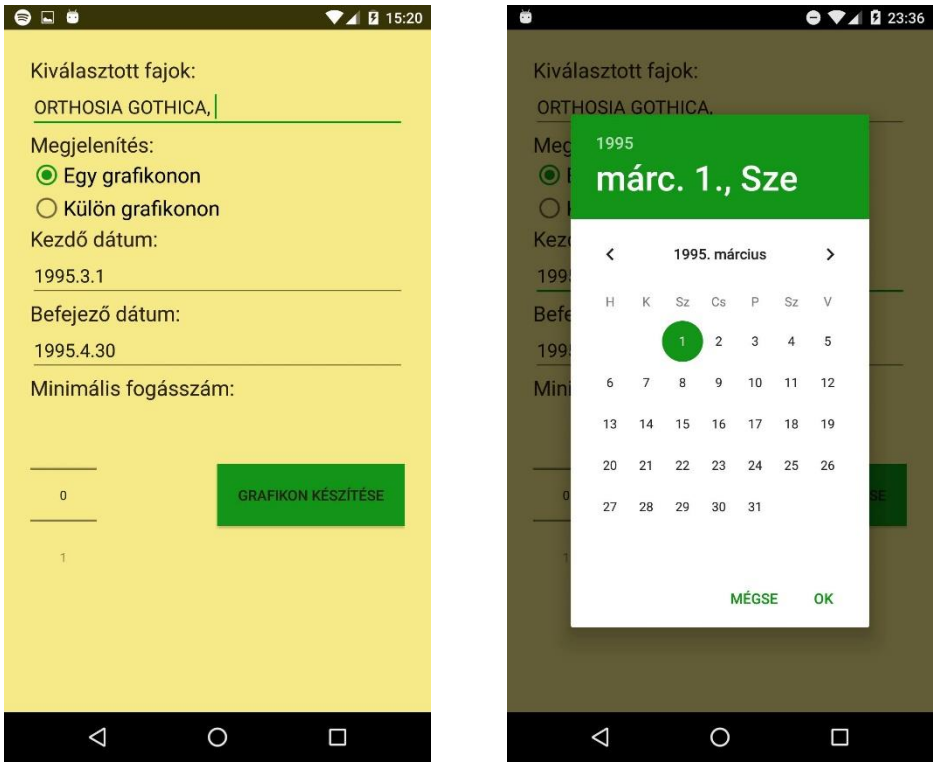
dayFragment, dayFragment.TAG) függvényhívással megadjuk, hogy az *Activity* melyik *View* eleméhez akarjuk a *Fragment-et* hozzáadni, ez itt egy *FrameLayout*, utána meg kell adni a *Fragment* objektumot, majd a végén a *Fragment Tag-jét*. A tag hasonló azonosító funkciót lát el *Fragment-ek* esetében, mint *View-k,* vagy *Layout-ok* esetén az id-k. A hívás után a tranzakciót le kell zárni egy *transaction.commit()* metódussal.

```
FragmentManager manager = getSupportFragmentManager();
FragmentTransaction transaction = manager.beginTransaction();
DayFragment dayFragment = (DayFragment)
manager.findFragmentByTag(DayFragment.TAG);
if (dayFragment == null) {
  dayFragment = new DayFragment();
}
transaction.add(R.id.frame_layout, dayFragment, dayFragment.TAG);
transaction.commit();
```

A programban két fajta *Fragment* osztály létezik, az egyik a napi nézetet támogató *DayFragment,* a másik a havi nézetet támogató *MonthFragment.* A *Fragment-ek* megfelelő működéséhez két életciklus függvényt kell felüldefiniálni. Az egyik az *onAttach(Context context)* metódus. Ez arra szolgál, hogy referenciát szerezzen a *Fragment* az *Activity-re,* és azt eltárolja egy *Activity* típusú tagváltozóban. Ez a metódus akkor fut le, amikor az *Activity-re* csatolódik a *Fragment*. Az *onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)* metódusban lehet létrehozni a *Fragment* felületét. A *LayoutInflater* segítségével lehet felfújni a megfelelő *View* elemet, A *ViewGroup* megadja azt a visszaadott *View* szülőkomponensét, ami jelen esetben null, a *Bundle* paraméter segítségével pedig a *Fragment* elmentett állapotát lehet helyreállítani.

A metódus a futás elején referenciát szerez a *Fragment* felületét leíró erőforrásról a *View view = inflater.inflate(R.layout.day_fragment, null)* függvényhívással. A *View* egyes elemeit a *view.findViewById* hívással lehet megszerezni. Mivel a dátum beállító felület elemek különböznek a telefonra, illetve a táblagépre szánt verziónál ezért a *View* elem megjelenítése előtt ellenőrizni kell, hogy nem null-e az elem, és csak abban ez esetben felfújni, ha nem null. A különbség a táblagépre, és a telefonra szánt elrendezés között csak abban jelenik meg, hogy a telefonon *TextView-k* jelennek meg, táblagépen pedig naptár nézetek. A naptár nézetet a prototípus esetében ismertettem, most a szöveg mezőkhöz tartozó felugró ablakokat mutatom be. A 14. és a 15 ábra a napi nézetet paraméterező *Activity-t* mutatja be mobiltelefonon, illetve táblagépen megjelenítve.

```
editTextBegin = (EditText) view.findViewById(R.id.editTextBeginYear);
editTextBegin.setText(activity.getBeginyear() + "." +
activity.getBeginmonth() + "." + activity.getBeginday());
final DatePickerDialog.OnDateSetListener datePickerDialogBegin = new
DatePickerDialog.OnDateSetListener() {
  @Override
  public void onDateSet(DatePicker view, int year, int monthOfYear, int
  dayOfMonth) {
      monthOfYear++;
      activity.setBeginyear(year);
      activity.setBeginmonth(monthOfYear);
      activity.setBeginday(dayOfMonth);
      editTextBegin.setText(activity.getBeginyear()+"."+
      activity.getBeginmonth() + "." + activity.getBeginday());
  }
};
editTextBegin.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
      InputMethodManager methodManager = (InputMethodManager)
      getActivity().getSystemService(Context.INPUT_METHOD_SERVICE);
      methodManager.hideSoftInputFromWindow(editTextBegin.getWindowToken(),
      0);
      new DatePickerDialog(getActivity(),
      datePickerDialogBegin, 1995, 2, 1).show();
  }
});
```



14. ábra - A napi nézet és a dátum kiválasztására szolgáló DatePickerDialog

15. ábra - A napi nézet táblagépre készült verziója

Az *EditText* inicializása után beállítja a program a hozzá tartozó szöveget, ami az *Activity-ben* eltráolt kezdődátum 1995.03.01. Ezután létrejön egy *DatePickerDialog.* Ennek meg kell valósítani az *OnDateSetListener-jét,* ami egy *onDateSet* metódust tartalmaz. Itt a dátum váltoásának megfelelően aktualizálódnak az Activity adattagjai, illetve az *EditText* szövege, oda kell figyelni arra, hogy a hónap értéket eggyel korrigálni kell. Az *EditText-hez* hozzárendel a program egy *OnClickListener-t,* itt az *onClick* metódusban a program referenciát szerez a bevitelt támogató rendszerszolgáltatásról. Majd letiltja a képernyő billentyűzet megjelenítését. Erre azért van szükség, hogy a felhasználó ne tudjon értelmetlen adatokat bevinni. Ezután a program létrehoz, és megjelenít egy *DatePickerDialog-ot*, az előbbi *DatePicker* objektum segítségével. Ennek a dátumát szintén 1995. március 1-jére állítja be.

A telefonra készült *MonthFragment* verzió négy *EditText-et* tartalmaz a kezdőév, kezdőhónap, befejezőév és befejezőhónap beállítása számára. Ezeknek az *inputType* attribútuma *number* típus, tehát csak számot lehet bevinni értékként a billentyűzettel. Itt ezért ellenőrizni kell, hogy a felhasználó az 1-12 intervallumból választott-e hónap értéket. Ha nem innen választott, akkor figyelmezteti a program erre egy *Toast* üzenet segítségével. Ez a folyamat már a *chartButton ActionListener-ében* történik, amikor a felhasználó lekéri a grafikont.

A különböző képernyő méretek támogatása még egy helyen jelenik meg a vékony kliens alkalmazásban. A térkép nézetet megvalósító *SensorPlaceActivity onMapReady(GoogleMap googleMap)* függvényében. Itt történik meg a térképre a szenzorok koordinátáinak kirajzolása, a térkép középpontozása, és a közelítés beállítása. A prototípus verzióban a legkeletibb szenzorok kilógtak a térképről, és Magyarország se fért rá teljesen. Ezért több változtatást is eszközöltem. Pusztavacsot, mint térkép középpontot leváltottam egy tőle észak-keletebbre lévő koordinátára, és a zoomolási szint megállapításánál figyelembe vettem a képernyő tájolását, illetve méretét.

```
if ((getResources().getConfiguration().screenLayout &
      Configuration.SCREENLAYOUT_SIZE_MASK) ==
      Configuration.SCREENLAYOUT_SIZE_SMALL) {
  Log.e("debug", "normal");
  if ((getResources().getConfiguration().orientation ==
Configuration.ORIENTATION_LANDSCAPE)) {
      LatLng latLng = new LatLng(47.0, 19.4);
      mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(latLng, 5f));
  }
  if (getResources().getConfiguration().orientation ==
Configuration.ORIENTATION_PORTRAIT) {
      LatLng latLng = new LatLng(47.0, 19.4);
      mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(latLng, 4f));
  }
}
```

A külső elágazásban történik a képernyő méretének ellenőrzése, itt a *getResources().getConfiguration().screenLayout*, és a hívás az eszköz adatait kéri le, a másik a *Configuration.SCREENLAYOUT_SIZE_SMALL* pedig azt ellenőrzi, hogy a visszaadott érték kis képernyőt jelent-e. Hasonlóképpen történik meg a fekvő, és az álló helyzet kiválasztása. A *LatLng latLng = new LatLng(47.0, 19.4*) hívás állítja be a térkép középpontját. A *mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(latLng, 5f))* pedig a térkép közelítési szintjéért felel. Minél nagyobb benne a float változó értéke annál jobban ráközelít a program a térképre. Minél nagyobb a képernyő annál nagyobb számot adok meg közelítési szintnek, így a térkép mindig a megfelelő közelségi szinten látszik. Fontos megemlíteni, hogy az *onMapReady(GoogleMap googleMap)* függvény lefut akkor is, amikor a felhasználó elforgatja a készüléket, így mindig aktualizálódik a közelítés a fekvő, és az álló mód között. Ehhez hasonló elágazások segítségével választódik ki a normal, large, és x-large felülethez megfelelő zoomolás is.

5.5.3. A többnyelvű felhasználó felület kialakítása

Fontosnak tartottam, hogy az alkalmazás több nyelven is elérhető legyen a felhasználók számára. Ezért a magyar mellett elkészítettem a felhasználói felület angol és német nyelvű verzióját is. Az Android operációsrendszer magas szinten támogatja a több nyelv kezelését az alkalmazásokon belül. Ehhez az szükséges, hogy a felhasználói felület szöveges elemeit egy string.xml fájlban tároljuk el, és ne a forráskódba legyenek beleégetve. A *strings.xml* a */res/values* mappában tárolódik. A values mappáknak lehet minősítéseket adni, a *values-hu* mappa a magyar, a *values-en* az angol, a *values-de* mappa pedig a német nyelvű felhasználói felület elemeit tartalmazza.

```
<resources>
  <string name="internetconnect">Szeretne csatlakozni az
internetre?</string>
  <string name="yes">Igen</string>
  <string name="no">Nem</string>
```

```
<resources>
  <string name="internetconnect">Would you like connect to the
internet?</string>
  <string name="yes">Yes</string>
  <string name="no">No</string>
```

```
<resources>
  <string name="internetconnect">Möchten Sie mit dem Internet
verbinden?</string>
  <string name="yes">Ja</string>
  <string name="no">Nein</string>
```

A forráskód felől a name attribútum alapján lehet hivatkozni a *String* erőforrásokra, az *R.string.stringname* hívás segítségével. A legtöbb helyen elég csak ennyit írni, mivel *String* paraméter helyett nagyon sok beépített Adroid-specifikus osztály a szöveget String erőforrás azonosító alapján is be tudja állítani, nem muszáj konkrét *String-et* megadnunk. Arra viszont ügyelni kell, hogy *TextView-ra* nem szabad közvetlenül kiírni egy *String* erőforrást *R.string.stringname* formában, mivel ilyenkor a *String* erőforrás integer azonosítójának értéke fog megjelenni a képernyőn. A helyes megjelenéshez a *String string = getResources.getString(R.string.stringname)* hívást kell először elvégezni, majd a *String* objektumot megjeleníteni a *TextView-n.*

A nyelvi beállításokat a program mindig futásidőben dönti el, és megpróbálja a készülék nyelvének megfelelő nyelvi fájlt betölteni, és az alapján felépíteni a felhasználói felületet. Nagyon fontos, hogy a minősítés nélküli values könyvtárban lévő *srings.xml* fájl-ban is legyenek valamilyen elsődlegesen támogatott nyelv szövegadatai. Ez azért fontos, mert, ha a felhasználó egy nem támogatott nyelven használja a telefonját, akkor nem jelennének meg a szövegek a képernyőn, hanem csak a *String* objektumok azonosítói látszódnának. Mivel Magyarországra, magyar felhasználók számára készült elsősorban az alkalmazás ezért, ez az elsődleges nyelv is a magyar lett.

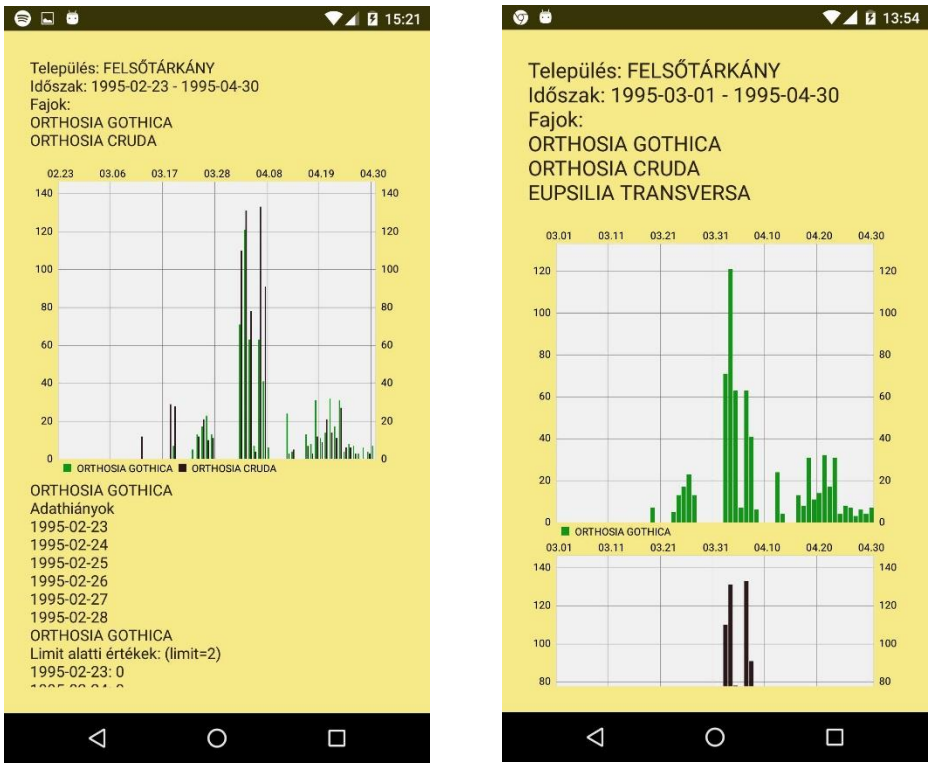
A felhasználó használni tudja más nyelven is az alkalmazást, mint amilyen nyelv a telefonjára be van állítva. Ezt a funkciót a *MainActivity languageButton-jára* kattintva lehet elérni. A gombnyomás után egy *AlartDialog* jelenik meg, amelyik felsorolja a felhasználónak a program nyelveit: Magyar, English, Deutch. A kiválasztott elemre kattintva a program lekéri a telefon nyelvi beállításokat, majd átállítja azokat a kiválasztott nyelvre, és újratölti a kiválasztott nyelven a *MainActivity-t.* Fontos megjegyezni, hogy a nyelvi beállításokat a készülék nem menti el, tehát ha nem a készülék nyelvén akarja a felhasználó használni a programot, akkor induláskor mindig ki kell választania a neki tetsző nyelvet. Az alkalmazás a Google Maps API segítségével jelenít meg térképeket, viszont ennek a térképnek a nyelvét nem befolyásolja a felhasználó nyelv választása az alkalmazásban, ez a készülék nyelvétől függ.

5.5.4. További változtatások és újítások a prototípushoz képest

Az alkalmazásnak további kisebb változtatásai is vannak a két verzió között, amelyek külön-külön nem tesznek ki egy-egy alfejezetet. Ezeket a módosításokat szeretném most itt ismertetni.

A program felhasználói felülete egy jellegzetes erdész színezetet kapott, a gombok színe sötét zöld lett, a háttérszín pedig sárga. A grafikonok oszlopai piros, és kék színek helyett zöld és barna árnyalatokban jelennek meg. Mivel nem fér ki háromnál több faj név a grafikon jelmagyarázatában, ezért ez a verzió csak három fajt enged meg bevinni a felhasználónak.
A grafikonok megjelenítésében is történtek kisebb változások, például a grafikonok fölött megjelenik egy leírósáv egy *TextView-ban*, ami leírja a kiválaszotott település nevét, a kiválasztott időszakot, illetve a kiválasztott fajokat sorolja fel, itt azok a fajok is megjelennek, amelyekre nincs találat az adott időszakban, és ilyenkor a "Nincs megjelenítendő adat" felirat jelenik meg a faj mellett, kötőjellel elválasztva. A grafikonrajzoló *Activity-k* felépítése is megváltozott ebben a verzióban. Itt az egész *Activity* tartalma egy *ScrollView-ba* van foglalva, és le lehet görgetni benne a legaljáig. A *MoreBarChartActivity ArrayAdapter és ListView* helyett *LiearLayout* segítségével teszi egymás alá a grafikonokat. Ez a nézet is legörgethetővé vált, és felhasználóbarátabb lett. A 16 ábrán látható a vékony kliens verzió napi grafikon megjelenítése egy és több grafikonos nézetben.

```
ArrayList<Chart> charts = (ArrayList) createCharts();
LinearLayout linearLayout = (LinearLayout) findViewById(R.id.chartList);
for(int i=0; i<charts.size(); i++){
  Chart chart = charts.get(i);
  linearLayout.addView(chart);
}
```



16. ábra - A fogások megjelenítése, egy illetve több grafikonos nézetben a vékony kliens verzióban

A Sensorhub-ra történő kapcsolódás miatt a vékony kliens alkalmazás internet kapcsolatot igényel a működéséhez. Az internet kapcsolat állapotát az *ErtiDataViewer.isOnlinine()* metódusa ellenőrzi minden lekérdezés előtt.

```
public static boolean isOnline() {
  ConnectivityManager connectivityManager = (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
  NetworkInfo activeNetworkInfo =
connectivityManager.getActiveNetworkInfo();
  return activeNetworkInfo != null && activeNetworkInfo.isConnected();
}
```

Ha az alkalmazás indulásakor nincs internet kapcsolat, akkor a program feldob egy *AlertDialog-ot*, "Szeretne csatlakozni az intertetre?" szöveggel. Ha a felhasználó igenlő választ ad, akkor a program egy intent segítségével a rendszer wifi beállítás paneljára irányítja a felhasználót, ahol bekapcsolhatja a wifit. Az *Activity onResume()* metódusában meghívódik az *ErtiDataViewer.changeDataLoader()* metódusa, ami ellenőrzi, hogy sikerült-e csatlakozi az internetre, és ha igen helyreállítja a kapcsolatot a Sensorhub-bal. Fontos megjegyezni, hogy az *onResume()* metódus az alkalmazás indulásakor is lefut az onCreate metódus előtt, ezért nem szükséges az *ErtiDataViewer* osztályban inicializálni az adatbázis kapcsolatot. Az internet bekapcsolására a felhasználónak manuálisan is van lehetősége a *MainActivity-n* lévő *internetButton* segítségével. Ennek az *OnClickListenerjében*, ugyanez az *AlertDialaog* fut le, és ugyanúgy a wifi beállításokhoz lehet eljutni vele, mint az alkalmazás indulásakor felugró ablakról. A kapcsolódás újraellenőrzése is hasonlóan az *onResume()* metódusban történik.

6. Összefoglalás

Ebben a félévben megterveztem és elkészítettem egy androidos mobilalkalmazást, amelyik képes az ERTI rovarcsapdáit és az OMSZ meteorológiai mérőhelyeit megjeleníteni térképen Google Maps segítségével. A program képes a kiválasztott rovarcsapda fogási adatait rovarfajokra és időszakra vonatkozó szűrések után grafikonon megjeleníteni. A félév során a legtöbb új tapasztalatot a grafikonok megjelenítésében szereztem, ez a tudás a későbbiekben hasznosnak bizonyulhat.

Eleinte a program helyi adatokon dolgozott, de a félév végére sikerült elkészíteni egy vékony kliens verziót is, amelyik az Informatikai és Gazdasági Intézetben futó Sensorhub környezetet használja adatforrásként.

A félév során csoportban dolgoztam együtt több hallgatótársammal és az intézet több oktatójával, ahol hasznos tapasztaltokkal gazdagodtam ezen a területen. Pödör Zoltán tanár úr felkérésére az alkalmazás prototípus verziójának bemutatásával részt vettem az április 6-án Győrben megrendezett Nyugat-dunántúli Regionális Innovációs Kiállítás és Tanulmányi Vásáron.

Az alkalmazás továbbfejlesztése számos irányban megtörténhet. A félév során nem került sor a meteorológiai adatok feldolgozására. Ezeknek az adatoknak a segítségével statisztikai összefüggéseket lehetne keresni az időjárás változása, és egyes rovarok fogási adatai között. Meg lehet valósítani olyan szűrési funkciót is, amelyik például több csapda fogási adatait jeleníti meg egy fajra vonatkozóan. Az ERTI dolgozói egy olyan funkciót is hasznosnak látnának, amely segítségével a fogási helyen tudják rögzíteni a befogott rovarok számát, és realtime betölteni akár a Sensorhub-ba, vagy más adatbázisokba. További hasznos funkció lenne az erdészek szerint különböző fájl formátumokba például csv-be, txt-be vagy pdf-be tudja az alkalmazás a szűrt adatokat exportálni.

Irodalomjegyzék

[1] Ekler Péter, Fehér Marcell, Forstner Bertalan, Kelényi Imre. Android-alapú szoftverfejlesztés. SZAK Kiadó Kft. 2012

[2] SensorHUB - An IoT Driver Framework for Supporting Sensor Networks and Data Analysis. BME AUT. https://www.aut.bme.hu/Pages/Research/SensorHUB. *(Utolsó megtekintés: 2016. április 25.)*

[3] Ikvahidi Ádám. Grafikus mobil megjelenítési funkciók kialakítása a BME/NymE SensorHub környezethez kapcsolódva. 2015

[4] Balogh Tamás, Ujj Tamás István. ICT Framework for Supporting Connected Road Vehicles. BME-VIK TDK dolgozat. 2014

[5] Charts Google Developers. https://developers.google.com/chart/ *(Utolsó megtekintés: 2016. április 20.)*

[6] PhilJay/MPAndroidChart. GitHub. https://github.com/PhilJay/MPAndroidChart/wiki *(Utolsó megtekintés: 2016. április 20.)*

[7] Dave MacLean, Satya Komatineni. Android Fragments. Friendsof Apress. 2014