Debreceni Egyetem

Informatikai Kar

Alkalmazásfejlesztés Android platformra

Belső témavezető:	***Külső témavezető:***	***Készítette:***
Bátfai Norbert	*Szabó Ákos*	*Kovács Szilárd*
Egyetemi tanársegéd	*Micont Kft.*	*Programtervező informatikus*

Debrecen

2011

Tartalomjegyzék

1. Bevezetés

A multimédiás készülékek, mint a számítógép, televízió, internet és mobiltelefon napjainkban már természetes, mindennap használatos eszközökké váltak. Mivel ezekeket gyakran használjuk, nem feltétlenül vesszük észre, hogy ezek a valaha forradalmi újdonságoknak számító eszközök mennyire befolyásolják mindennapjainkat és mekkora hatásuk van életünk fontosabb területeire.

Ezen készülékek lelke időről időre egyre kisebb és kompaktabb, viszont a teljesítményük növekszik. Elértünk arra a szintre, hogy legtöbbször egyszerűbb egy beágyazott rendszerbe olyan sztenderd és elterjedt hardverelemeket építeni, amelyeket egy operációs rendszer is alapból támogat, így a hardver- és szoftverfejlesztés ideje egyaránt töredékére csökken.

Linux alapú eszközök számtalan termékben fordulnak elő, így az Android operációs rendszer Linux alapokkal, Java támogatással és remek megjelenítés kezelőjével elég gyorsan népszerűvé vált.

Legelterjedtebb Androidos készülékek a mobiltelefonok. Mivel ez az operációs rendszer Linux alapokra épült, így kezdetét veszi a beágyazott rendszerekben való elterjedése is. Így a Java-ban való fejlesztés előnyei - a gyorsabb és biztonságosabb termékek készítése - is kihasználásra kerülnek.

A diplomamunkámban segítséget nyújtó cég a MICONT Kft. saját készítésű intelligens lakásautomatikához tervezett beágyazott rendszereit olyan hardverekkel váltja ki, melyeken képes futni az Android rendszer. Abból indultunk ki, hogy az alapoktól kezdve először egy működő rendszerre, azaz egy mobiltelefonra készüljön el egy olyan alkalmazás, amely az Android rendszer főbb részeit használja ki, úgy, mint a megjelenítés, internet kapcsolat, rendszeresemények, adattárolás. Másrészt fontosnak tartottuk, hogy komplett, mások által is használható legyen a programunk. Ezzel fogjuk bizonyítani magunknak, hogy tudunk az Android rendszerrel termék minőségű alkalmazást fejleszteni és érdemes a jövőben is erre a rendszerre alapozni.

Jelen dolgozatom célja bemutatni az Android platformot, annak részeit és alapvető működését, továbbá a telekomminikációs megoldások közül egy Google Voice szolgáltatásra épülő kliens alkalmazás megvalósítását Android platformon.

2. Android platform

2.1. Mi is az az Android?

Az Android platformot az Open Handset Alliance (OHA) konzorcium, egy multinacionális szövetség hozta nyilvánosságra 2007. november 12.-én. Az OHA a vezető mobil- és technológiai társaságok egy csoportja, melynek eddig összesen nyolcvan hardver, szoftver és telekommunikációs cég a tagja, mint például a Google, LG, Samsung, Vodafone.

Az Android platform az alapoktól kezdve épült fel, hogy lehetőséget biztosítson a fejlesztőknek olyan új és innovatív mobil alkalmazások létrehozására, melyek kihasználják az internet csatlakozásra képes eszközök összes előnyét.

Az Android platform magába foglalja a Dalvik virtuális gépet, hogy maximalizálja az alkalmazások teljesítményét, a hordozhatóságot és a biztonságot, továbbá optimalizálja a memória és a hardver erőforrásokat mobil környezetben.

Az Android SDK gazdag eszköztárával lehetővé teszi a fejlesztőknek az alkalmazások fejlesztését erre a platformra. Ide tartoznak a speciális fejlesztő és hibakereső eszközök, gazdag könyvtárak, egy igazi eszköz emulátor, részletes dokumentáció, minta projektek és útmutatók. A fejlesztők zökkenőmentes munkáját egy Eclipse plugin segíti, mely ezeket az eszközöket integrálja az Eclipse integtegrált fejlesztőesztői környezetéhez.

Az Android előretörésének legnagyobb nyertesei a kétségtelenül készülékgyártók. 2009-ben a világgazdasági válságnak köszönhetően érezhetően csökkentek az eladások, viszont a Gartner Inc., a világ vezető információ- technológiai kutató és tanácsadó társasága szerint a mobil készülékekből 2010-ben világviszonylatban közel 1,6 milliárd db-ot adtak el, amely a 2009-es adatokhoz képest 31,8% növekedést jelent. Ezen belül az okostelefon eladások 2010-ben 72,1%-kal növekedtek az előző évhez képest, az összes 2010-ben eladott mobil készülék 19%-a (297 millió darab) okostelefon volt.

Mobil készülékek eladásai világviszonylatban 2010-ben (ezer darab):

Gyártó cég	2010 eladott mennyiség	2010 piaci részesedés (%)	2009 eladott mennyiség	2009 piaci részesedés (%)
Nokia	461 318,2	28,9	440 881,6	36,4
Samsung	281 065,8	17,6	235 772,0	19,5
LG Electronics	114 154,6	7,1	121 972,1	10,1
Research In Motion	47 451,6	3,0	34 346,6	2,8
Apple	46 598,3	2,9	24 889,7	2,1
Sony Ericsson	41 819,2	2,6	54 956,6	4,5
Motorola	38 553,7	2,4	58 475,2	4,8
ZTE	28 768,7	1,8	16 026,1	1,3
HTC	24 688,4	1,5	10 811,9	0,9
Huawei	23 814,7	1,5	13 490,6	1,1
Többi	488 569,3	30,6	199 617,2	16,5
Összesen	1 596 802,4	100,0	1 211 239,6	100,0

Forrás: Gartner (2011 február)

Az okostelefonok piaca továbbra is a fejlett országokban koncentrálódik, ahol a vevők jövedelme nagyobb és ahol a hálózatok elég gyorsak ahhoz, hogy maradéktalanul ki lehessen használni az okostelefonok tudását. Ez magyarázza azt, hogy 2010. negyedik negyedévében az okos telefon eladások 52,3%-a, valamint az összes mobiltelefon értékesítések közel fele Nyugat-Európában és Észak-Amerikában történt. Az erős verseny 2010 év végére elérte az okostelefon piac legnagyobb szereplőit is.

Okos telefonok eladásai világviszonylatban 2010-ben operációs rendszer szerint (ezer darab):

Operációs rendszer	2010 eladott mennyiség	2010 piaci részesedés (%)	2009 eladott mennyiség	2009 piaci részesedés (%)
Symbian	111 576,7	37,6	80 878,3	46,9
Android	67 224,5	22,7	6 798,4	3,9
BlackBerry	47 451,6	16,0	34 346,6	19,9
iOS	46 598,3	15,7	24 889,7	14,4
Windows	12 378,2	4,2	15 031,0	8,7
Többi	11 417,4	3,8	10 432,1	6,1
Összesen	296 646,6	100,0	172 376,1	100,0

Forrás: Gartner (2011 február)

Az okos telefon piacon az Androidos készülékek eladása 888,8%-al növekedett 2009-hez képest és így piaci részesedés szempontjából a 2010-ben a második helyre került. A Nokia továbbra is a legnagyobb okostelefon-gyártó, hiszen a Symbian szinte teljes egészében az ő készülékeiben található meg.

Az Android több gyártó termékében is megjelenik. A két legnagyobb a Samsung és a HTC - ezek együtt adták 2010-ben az összes Androidos készülék forgalmának csaknem felét. A www.unwiredview.com szerint csak a Samsung az eladások 34%-át tudhatta magáénak, ezzel a Google legfőbb partnerévé vált.

Összességében elmondható, hogy a Google operációs rendszert felhasználó gyártók profitáltak az Android elterjedéséből, mivel a LG, Samsung, HTC okos telefonokból származó forgalma jelentős növekedést mutatott, ráadásul a csúcskészülékeken kívül a középkategóriás termékek eladása is emelkedett a Canalys elemzőcég szerint.

Az elemzőcégek 2011-re is kiélezett versenyt jósolnak a gyártók, valamint a mobil operációs rendszerek között.

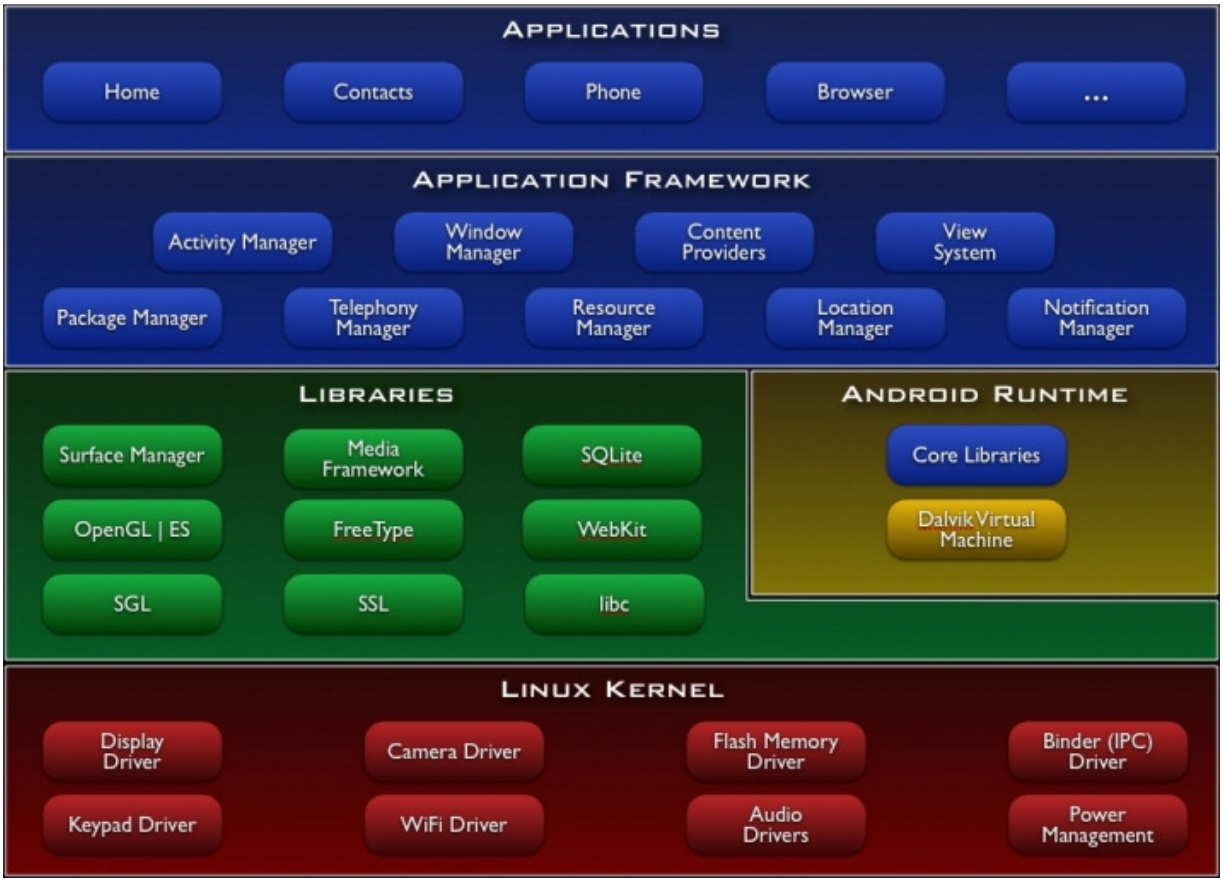
2.1.1. Sajátosságok

- **Alkalmazás keretrendszer:** a komponensek cseréjéhez és újrafelhasználásához
- **Dalvik virtuális gép:** mobileszközökre optimalizálva
- **Integrált böngésző**: nyílt forrású WebKit motorra építve
- **Optimalizált rajzolás:** a 2D rajzoláshoz egyedi grafikus könyvtárat biztosít; a 3D rajzolás az OpenGL ES 1.0 specifikációra épül (opcionális hardver gyorsítással)
- **SQLite:** a strukturált adattároláshoz
- **Média támogatás:** támogatja az összes elterjedt audio, video és képformátumot (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- **GSM Telephony:** (hardver függő)
- **Bluetooth, EDGE, 3G és WiFi:** (hardver függő)
- **Kamera, GPS, iránytű és gyorsulásmérő:** (hardver függő)

- **Gazdag fejlesztői környezet:** magában foglalja a telefon emulátort, a nyomkövetéshez szükséges eszközöket, memória- és teljesítménymérést, továbbá az Eclipse IDE-hez szükséges plugint.

2.1.2. Architektúra

Az alábbi kép az Android operációs rendszer fő komponenseit ábrázolja, melyek később részletes bemutatásra kerülnek.



2.1.3. Alkalmazások

Az Android eleve tartalmaz egy alkalmazás alapcsomagot, többek között egy levelező klienst, SMS programot, naptárt, térképeket, böngészőt, névjegyzék kezelőt, és sok mást. Az alkalmazások Java programozási nyelven készültek.

2.1.4. Alkalmazás keretrendszer

A nyílt fejlesztői környezettel az Android lehetőséget biztosít a fejlesztőknek, hogy a lehető legmagasabb színvonalú és innovatív alkalmazásokat készítsenek. A fejlesztők szabadon kihasználhatják az adott eszköz hardware képességeit, hozzáférnek az összes információhoz, futtathatnak háttérszolgáltatásokat, riasztásokat állíthatnak be, üzeneteket írhatnak ki a fejlécre, és még sok-sok mást.

A fejlesztők teljes hozzáféréssel rendelkeznek a mag-alkalmazások által használt keretrendszer API-khoz. Az alkalmazás architektúra tervezése során elsődleges szempont volt a komponensek egyszerű újrafelhasználhatósága; bármely alkalmazás publikálhatja a saját képességeit, és bármely más alkalmazás igénybe veheti azokat (a keretrendszer által meghatározott biztonsági előírások figyelembe vételével). Ugyanez a mechanizmus teszi lehetővé a felhasználók számára a komponensek cseréjét.

Az alkalmazások hátterében található jópár szolgáltatás és rendszer, mint például:

- Alkalmazásfejlesztéshez használt **Nézetek** egy sokrétű és bővíthető halmaza, ideértve a listákat, táblázatokat, rácsokat, szövegdobozokat, gombokat, vagy akár a beágyazható böngészőt is.
- **Tartalom szogáltatók**, melyeknek segítségével az alkalmazások más alkalmazások adataihoz férhetnek hozzá (például a Névjegyzék), vagy megoszthatják saját adataikat.
- **Erőforrás kezelő**, forráskódon kívüli erőforrások eléréséhez, mint például adott nyelvű (lokalizált) stringek, grafikák, stb.
- **Értesítés kezelő**, aminek segítségével az alkalmazások saját értesítéseket, figyelmeztetéseket jeleníthetnek meg az információs sávon.
- **Tevékenység kezelő**, ami az alkalmazások életciklusát vezérli és egy közös navigációs felületet biztosít.

2.1.5. Könyvtárak

Az Android tartalmaz egy C/C++ könvtárat, amit az Android rendszer több komponense is használ. Ezeket a lehetőségeket az Android alkalmazás keretrendszeren keresztül érhetik el a fejlesztők. A példa kedvéért felsorolunk néhány alapkönyvtárat:

- **C Rendszerkönyvtár** – az alap C rendszerkönyvtár (libc) egy BSD-származtatott implementációja, Linux alapú beágyazott rendszerekre hangolva.
- **Média Könyvtárak** – PacketVideo OpenCORE alapú; a könyvtárak támogatják a legtöbb elterjedt audio- és video- formátum visszajátszását és rögzítését, akárcsak a legtöbb képformátumot, mint például: MPEG4, H.264, MP3, AAC, AMR, JPG, vagy PNG.
- **Felület Manager** – a megjelenítő alrendszerhez történő hozzáféréseket manageli, és folyamatosan kezeli a különböző alkalmazások által előállított 2D és 3D grafikus rétegek akadálytalan megjelenítését.
- **LibWebCore** – korszerű internetböngésző motor, amely egyaránt kiszolgálja az Android és a beágyazott böngészőket.
- **SGL** – a 2D grafikus motor.
- **3D könyvtárak** – OpenGL ES 1.0 API-kra épülő implementáció; a könytárak vagy 3D hardware gyorsítást használnak (ahol elérhető) vagy egy optimalizált 3D szoftveres raszterizert.
- **FreeType** – bitmap és vector betűtípus renderelő.
- **SQLite** – robosztus és pehelysúlyú relációs adatbázismotor, amely minden alkalmazás számára hozzáférhető.

2.1.6. Android futási környezet

Az Android tartalmaz egy alap könytárat, amely megvalósítja a Java programozási nyelv funkcionalitásának túlnyomó részét.

Minden Android alkalmazás saját szálon fut és saját Dalvik virtuális gép példánnyal rendelkezik. A Dalvik-ot úgy írták meg, hogy az eszköz több Virtuális Gépet is tudjon futtatni egyszerre hatékonyan. A Dalvik VM (Virtual Machine – Virtuális Gép) Dalvik Futtatható Fájl (.dex)

formátumot futtatja, amit minimális memóriahasználatra optimalizáltak. A VM regiszter-alapú, és Java nyelvi fordítóval készített osztályokat futtat amelyeket .dex formátumra alakítottak a kapcsolódó “dx” eszközzel.

A Dalvik VM a Linux kernelre támaszkodik az alacsonyszintű funkcionalitással kapcsolatban, mint például a szálkezelés vagy az alacsony szintű memóriakezelés.

2.1.7. Linux kernel

Az Android a 2.6 Linux verzióra épül a rendszerszolgáltatásokkal kapcsolatban, mint a biztonság, memóriakezelés, szálkezelés, hálózatkezelés és meghajtó modellek. A mag ezen felül absztrakciós rétegként is viselkedik a hardver/ és a szoftverréteg között.

2.2. Android alapelemek

Az Android alkalmazások Java programozási nyelven készülnek. Az Android SDK eszközök fordítják le a forráskódot – a szükséges adat és nyersanyag fájlokkal együtt – egy *Android csomaggá*, ami .apk kiterjesztéssel rendelkezik. Egy .apk fájlban található összes kód egyetlen alkalmazásnak minősül, ez az a fájl amit az Android eszközök fel tudnak használni arra, hogy telepítség az alkalmazást.

Miután feltelepülek az eszközre, minden Android alkalmazás a saját biztonsági “homokozójában” (sandbox) él tovább:

- Az Android operációs rendszer egy többfelhasználós Linux rendszer, amiben minden alkalmazás egy külön felhasználóként jelenik meg.
- Alaphelyzetben a rendszer minden alkalmazásnak oszt egy egyedi Linux felhasználó ID-t (ezt az ID-t csak a rendszer használja, az adott alkalmazás nem ismeri a saját ID-jét). A rendszer állítja be a jogosultságokat az alkalmazásban található fájlokhoz, így csak a megadott felhasználó ID-vel rendelkező alkalmazások érhetik el.
- Minden folyamatnak saját virtuális gépe (VM) van, így egy adott alkalmazás kódja elkülönítve fut az összes többi alkalmazásétól.

- Alaphelyzetben minden alkalmazás a saját Linux folyamatát futtatja. Az Android elindítja a folyamatot, amikor az alkalmazás valamelyik komponensét futtatni kell, majd leállítja a folyamatot, amikor már nincs rá szükség vagy a rendszernek más alkalmazások számára memóriát kell felszabadítania.

Íly módon az Android rendszer megvalósítja a *legkisebb jogosultság elvét*. Ez azt jelenti, hogy alaphelyzetben minden alkalmazás csak a számára szükséges komponensekhez fér hozzá, és többhöz nem. Ez egy nagyon biztonságos környezetet teremt, amiben egy alkalmazás nem férhet hozzá a rendszer olyan részeihez amihez nincs jogosultsága.

Ettől függetlenül azonban van rá mód, hogy egy alkalmazás más alkalmazásokkal osszon meg adatokat vagy rendszerszolgáltatásokat érjen el:

- Megoldható, hogy két alkalmazás ugyanazt a Linux felhasználó ID-t használja, így ebben az esetben hozzáférhetnek egymás fájljaihoz. A rendszer erőforrások megfelelő kihasználása érdekében az ugyanolyan felhasználó ID-vel rendelkező alkalmazások futhatnak ugyanazon a Linux folyamaton és megoszthatják ugyanazt a VM-et (az alkalmazásoknak ugyanazzal a tanúsítvánnyal is kell rendelkezniük).
- Egy alkalmazás engedélyt kérhet arra, hogy eszköz adatokhoz férjen hozzá, mint például a névjegyzék, SMS üzenetek, felcsatolható tárolók (SD kártya), kamera, Bluetooth, stb. Minden alkalmazás engedélyt a felhasználónak kell megadnia telepítéskor.

Ezzel le is fedtük az alapokat azzal kapcsolatban, hogy hogyan is létezik egy Android alkalmazás a rendszerben. A dolgozat hátralévő részében bemutatjuk:

- A belső keretrendszer komponenseket, amelyek meghatározzák az alkalmazást.
- A *manifest* fájlt, amiben deklaráljuk az alkalmazás számára szükséges komponenseket és készülék jellemzőket.
- Azokat az erőforrásokat, amelyek elkülönülnek az alkalmazás kódjától, és lehetővé teszik az alkalmazás számára, hogy könnyedén optimalizálja a viselkedését a kölönböző eszköz konfigurációkon.

Az alkalmazás komponensek az Android alkalmazások alap építőkövei. Minden komponens egy pont, amin keresztül a rendszer be tud lépni az adott alkalmazásba. Nem minden komponens tényleges belépési pont a felhasználó számára, ezen felül egyes komponensek egymástól függenek, de mindegyik önálló példányban létezik és egy megszabott szerepet tölt be – minden szerep egyedi építőelem ami segít meghatározni az alkalmazás általános viselkedését.

Négy alkalmazás komponens típust különböztetünk meg. Mindegyik típus egy adott célt szolgál, és adott életciklusa van, ami meghatározza hogyan jön létre a komponens és hogyan szűnik meg.

Az alkalmazás komponensek négy típusa a következő:

- Activity
- Szolgáltatások
- Broadcast receiver-ek
- Tartalom szolgátatók

2.2.1. Activity-k

Egy *tevékenység (Activity)* egyetlen, felhasználói felülettel rendelkező képernyőt jelképez. Például, egy email alkalmazásnak lehet egy tevékenysége, ami az új levelek listáját jeleníti meg, egy másik tevékenység a levélírást valósítja meg, egy másik tevékenység pedig a levélolvasást szolgálja. Bár a tevékenységek együttműködve adnak ki egy teljeskörű felhasználói élményt (levelezőklienst), alapvetően egymástól függetlenek. Ezek alapján egy másik alkalmazás elindíthatja ezen tevékenységek bármelyikét (amennyiben az email alkalmazás engedélyezi). Például, egy kamera alkalmazás elindíthatja a levelező alkalmazás új levél tevékenységét, hogy a felhasználó elküldhesse másoknak az adott fényképet.

Egy tevékenységet az Activity osztályt kiterjesztő alosztályként lehet megvalósítani.

2.2.1.1. Activity-k életciklusa

A tevékenységek életciklusának kezelése callback (visszahívó) metódusok implementálásával létfontosságú egy robosztus és rugalmas alkalmazás fejlesztéséhez. Egy tevékenység életciklusát

közvetlenül befolyásolja a más tevékenységekkel való kapcsolata, feladatai és háttér verme (back stack).

Egy tevékenység alapvetően három állapotban lehet:

- ***Futó (Resumed):*** A tevékenység a kijelző előterén van és birtokolja a felhasználói fókuszt. (Ezt az állapotot „futó” („running”) állapotnak is szokták nevezni)
- ***Felfüggesztett (Paused):*** Egy másik tevékenység van a kijelző előterén felhasználói fókusszal, de ez a tevékenység továbbra is látható. Azaz, egy másik tevékenység látható erre a tevékenységre rárajzolva, de az a tevékenység vagy részlegesen áttetsző vagy nem fedi le a teljes kijelzőt. Egy felfüggesztett tevékenység teljes mértékben élő (az Activity objektum a memóriában marad, megtartja az összes állapot- és taginformációját, és csatlakozva marad az ablakkezelőhöz), de a rendszer elpusztíthatja kritikusan alacsony memóriaszint esetén.
- ***Leállított (Stopped):*** A tevékenységet teljes mértékben lefedi egy másik tevékenység (a tevékenység most a „háttérben” van). Egy leállított tevékenység továbbra is élő (az Activity objektum a memóriában marad, megtartja az összes állapot- és taginformációját, de nem csatlakozik az ablakkezelőhöz). Ugyanakkor már nem látható a felhasználó számára, így a rendszer bármikor elpusztíthatja az objektumot, ha máshol van szükség az általa lefoglalt memóriaterületre.

Ha egy tevékenység felfüggesztett vagy leállított, a rendszer eltávolíthatja a memóriából vagy úgy, hogy megkéri, fejezze be a működését (meghívja a finish() metódusát) vagy egyszerűen leállítja a folyamatát. Amikor egy tevékenységet újra megnyitunk (miután befejeződött vagy elpusztult), újra létre kell hozni.

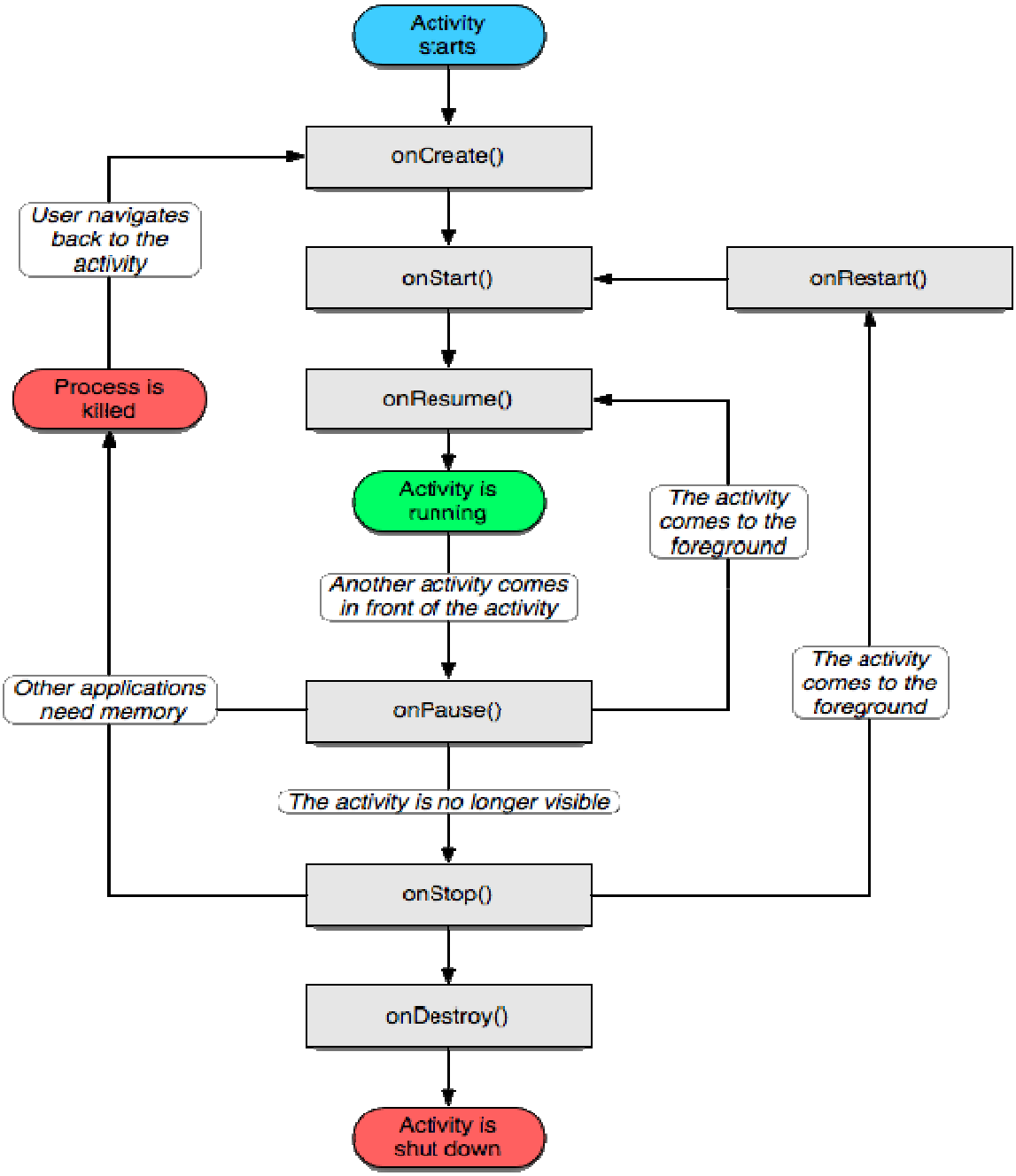
Összevetve ezeket a metódusok határozzák meg egy tevékenység teljes életciklusát. Ezen metódusok megvalósításával három beágyazott ciklust követhetünk nyomon a tevékenység életciklusban:

- A ***teljes életciklusa*** egy tevékenységnek az onCreate() és az onDestroy() metódusok meghívása között zajlik. Az adott tevékenységnek egy „globális” állapot beállítást kellene végrehajtania (például a grafikus elemek elrendezése) az onCreate() metódusban, és minden

erőforrást el kell engednie az onDestroy() metódusban. Például, ha a tevékenység a háttérben futtat egy szálat adatok letöltésére a hálózatról, akkor illik, hogy az onCreate()–ben hozza létre, és az onDestroy()-ban szünteti meg azt a szálat.

- A ***látható életciklusa*** egy tevékenységnek az onStart() és az onStop() metódusok meghívása között zajlik. Ez idő alatt a felhasználó látja a tevékenységet a kijelzőn és kommunikálhat vele. Például, az onStop() meghívásra kerül, amikor egy új tevékenység indul és ez már nem látható. A fenti két metódus között megtarthatjuk a szükséges erőforrásokat ahhoz, hogy megjelenítsük a felhasználó számára a tevékenységet. Például, regisztrálhatunk egy BroadcastReceiver objektumot az onStart() metódusban, hogy nyomon kövessük azokat a változásokat, amelyek hatással vannak a felhasználói felületre (UI – User Interface). Majd felszabadíthatjuk az objektumot az onStop() metódusban, amikor a felhasználó már nem láthatja, amit megjelenítünk. A rendszer többször is meghívhatja az onStart() és onStop() metódusokat egy tevékenység teljes életciklusa során, mivel a tevékenység váltogathat a felhasználó számára látható és nem látható állapotai között.
- Az ***előtér életcilusa*** egy tevékenységnek az onResume() és az onPause() metódusok meghívása között zajlik. Ez idő alatt a tevékenység minden más tevékenység előtt helyezkedik el a kijelzőn és birtokolja a felhasználói fókuszt. Egy tevékenység gyakran kerülhet ki és be az előtérbe – például, az onPause() metódust kerül meghívásra, amikor az eszköz alvó üzemmódba vált, vagy amikor megjelenik egy párbeszédablak. Mivel ez az állapot gyakran változik, a fenti két metódus között kód lehetőleg kicsi legyen a lassú váltások és a felhasználói várakozás elkerülése végett.

Az alábbi ábra bemutatja ezeket a ciklusokat és a tevékenység által bejárható különböző útvonalakat az állapotok között. A téglalapok visszahívó metódusokat szimbolizálnak, amikben megvalósíthatjuk a tevékenység által végrehajtandó utasításokat az egyes állapotok közötti átmenet során.



A fenti visszahívás metódusok megtalálhatóak az alábbi táblázatban is, amely részletesebben leírja az egyes metódusok működését és helyét egy tevékenység teljes életciklusában, illetve, hogy a rendszer megszüntetheti-e a tevékenységet miután a visszahívás metódus futása befejeződik.

Metódus	Leírás	Ezután megszün-tethető?	Következő
onCreate()	A tevékenység létrehozásakor hívódik meg. Itt kell az összes normális statikus beállítást elvégezni – nézetek létrehozása, adatok listákhoz kötése, stb. Ez a metódus megkap egy Bundle objektumot, amely a tevékenység előző állapotát tartalmazza, amennyiben az rögzítésre került (lásd Tevékenység Állapotok Tárolása, fejezet). Minden esetben az onStart() metódus követi.	Nem	onStart()
onRestart()	Egy leállított tevékenység újra elindítása előtt közvetlenül kerül meghívásra. Minden esetben az onStart() metódus követi.	Nem	onStart()
onStart()	Egy tevékenység a felhasználó számára láthatóvá válása előtt közvetlenül kerül meghívásra. Az onResume() metódus követi, ha a tevékenység az előtérbe kerül, vagy az onStop() ha rejtetté válik.	Nem	onResume() vagy onStop()
onResume()	Közvetlenül az előtt kerül meghívásra mielőtt egy tevékenység interaktívvá válna a felhasználó számára. Ezen a ponton a tevékenység a tevékenység-verem tetején van, és felhasználói inputot fogad. Minden esetben az onPause() metódus követi.	Nem	onPause()
onPause()	Akkor kerül meghívásra, mikor a rendszer egy másik tevékenységet készül folytatni (újraindítani). Ennek a metódusnak a tipikus felhasználása az el nem mentett	Igen	onResume() vagy onStop()

Metódus	Leírás	Ezután megszün-tethető?	Következő
	változások elmentése perzisztens adattá, animációk leállítása és más dolgok amik a processzor erőforrást foglalhatják, stb. Bármit is tesz azonban, azt jó ha nagyon gyorsan teszi, mert a következő tevékenység nem kerül ténylegesen folytatásra (újraindításra) amíg ez a metódus vissza nem tér. Az onResume() metódus követi amennyiben a tevékenység visszatér az előtérbe, vagy az onStop() ha a felhasználó számára láthatatlanná válik.		
onStop()	Ez a metódus akkor kerül meghívásra, mikor a tevékenység már nem látható a felhasználó számára. Ez történhet azért, mert a tevékenység objektum megszűnik, vagy mert egy másik tevékenység (akár új, akár már létező) újraindításra került és eltakarja a kijelzőn. Az onRestart() metódus követi, ha a tevékenység visszatér a felhasználó számára interaktív állapotba, vagy az onDestroy() ha a tevékenység végleg megszűnni készül.	Igen	onRestart() vagy onDestroy()
onDestroy()	A tevékenység megszűnésekor kerül meghívásra. Ez a legutolsó hívás amit a tevékenység megkap. Történhet azért, mert a tevékenység befejeződik (valaki meghívta rá a finish() metódust), vagy mert a rendszer átmenetileg megszűnteti a tevékenység példányt, hogy tárhelyet szabadítson fel. A két változat között az isFinishing() metódus segítségével tudunk különbséget tenni.	Igen	semmi

Az "Ezután megszűntethető?" oszlop azt mutatja, hogy a rendszer megszűntetheti-e a tevékenységet kiszolgáló folyamatot bármikor az adott visszahívó metódus lefutása után anélkül,

hogy akár a tevékenység egyetlen további sor kódját is végrehajtaná. Három metódus kapott „Igen” címkét: (onPause(), onStop(), és onDestroy()). Mivel az onPause() az első ebből a hármasból, miután a tevékenység példány létrejön, az onPause() lesz az a metódus amely garantáltan meghívásra kerül mielőtt a folyamatot meg lehet szűntetni. Amennyiben a rendszernek vészhelyzet esetén memóriát kell felszabadítani, akkor az onStop() és onDestroy() metódusok lehet, hogy nem is kerülnek meghívásra. Ebből kifolyólag javasolt az onPause() metódusban elhelyezni minden fontos perzisztens adat (például felhasználói adatok) eltárolására vonatkozó kódot. Ugyanakkor érdemes nagyon jól megválogatni, hogy mit kell az onPause() segítségével eltárolni mert bármilyen hosszú, blokkoló eljárás ebben a metódusban blokkolja a következő tevékenységre való átváltást, és lassítja a felhasználói élményt.

Az "Ezután megszűntethető?" oszlopban „Nem” címkével ellátott metódusok megvédik a tevékenységet kiszolgáló folyamatot a megszűntetéssel szemben attól a pillanattól kezdve, hogy meghívásra kerültek. Így egy tevékenység az onPause() visszatérése és az onResume() meghívása között megszűntethető, de egészen addig védettséget élvez, amíg az onPause() újra meghívásra nem kerül és lefut.

2.2.2. Szolgáltatások

Egy *szolgáltatás* egy olyan komponens, amely a háttérben fut és hosszútávú műveleteket végez, vagy távoli folyamatok számára dolgozik. Egy szolgáltatás nem nyújt felhasználói felületet. Például, egy szolgáltatás zenét játszhat le a háttérben, amíg a felhasználó egy másik alkalmazásban van, vagy adatot tölt le a hálózatról anélkül, hogy a felhasználót akadályozná más tevékenységek használatában. Egy másik komponens, például egy tevékenység, elindíthat egy szolgáltatást és hagyhatja futni, vagy rá is csatlakozhat, hogy kommunikáljon vele.

2.2.3. Broadcast receiver-ek

Egy *üzenetszórás fogadó (broadcast receiver)* egy olyan komponens, amely rendszer-szintű broadcast üzenetekre reagál. A legtöbb üzenetszórás (broadcast) a rendszertől ered — például egy ilyen üzenetszórás jelezheti, hogy a kijelző kikapcsolásra került, alacsony az akkufeszültség, vagy fénykép készült. Az alkalmazások is kezdeményezhetnek üzenetszórást — például, hogy

más alkalmazásokkal tudassák, hogy valamilyen adat letöltésre került az eszközre és elérhető vált számukra. Bár az üzenetszórás fogadók nem jelenítenek meg felhasználói felületet, a státusz mezőre (status bar) írhatnak, hogy figyelmeztessék a felhasználót, amikor egy üzenetszórás esemény történik. Gyakoribb azonban, hogy egy üzenetszórás fogadó csak egyszerű „átjáró" (gateway) más komponensekhez, és csak nagyon minimális munka elvégzésére tervezték. Például elindíthat egy szolgáltatást, ami majd elvégzi a megfelelő munkát egy esemény bekövetkeztekor.

2.2.4. Tartalom szolgáltatók

Egy *tartalomszolgáltató (content provider)* alkalmazás adatok egy megosztott halmazát kezeli. Adatot tárolhatunk a fájlrendszerben, egy SQLLite adatbázisban, a weben, vagy bármely más perzisztens tárhelyen amit az alkalmazásunk elér. A tartalomszolgáltatón keresztül más alkalmazások lekérdezhetik vagy akár módosíthatják is az adatokat (amennyiben a tartalomszolgáltató engedi ezt). Például, az Android rendszer rendelkezésre bocsát egy tartalomszolgáltatást, amely a felhasználó névjegyzék adatait kezeli. Ennek segítségével bármely, megfelelő jogosultsággal rendelkező alkalmazás megkérheti a tartalomszolgáltatót (mint a ContactsContact.Data) hogy írja vagy olvassa egy adott személy kapcsolatinformációit.

A tartalomszolgáltatók hasznosak az alkalmazás számára kizárólagos (nem megosztott) adatok olvasására/írására is.
Egy tartalomszolgáltatót a ContentProvider alosztályaként kell megvalósítani, továbbá implementálnia kell API-k egy szabványos halmazát is amely más alkalmazások számára ad lehetőséget tranzakciók végrehajtására.

2.2.5. Intent-ek és intent szűrők

Egy alkalmazás központi komponensei közül hármat — tevékenységek, szolgáltatások és üzenetszórás fogadók — üzenetekkel, *intent*-ekkel (intent – szándék) aktiválhatunk. Az intent üzenet egy olyan megoldás, amellyel késői, futási idejű kötés képezhető az egy alkalmazáson belüli, vagy épp különböző alkalmazások komponensei között. Az intent maga, ami Intent objektum, egy passzív adatstruktúra, ami a végrehajtandó művelet absztrakt leírását tartalmazza — vagy gyakran üzenetszórások esetén, valami megtörténtnek a leírását tartalmazza,

amit a rendszer így közzétesz. A különböző típusú komponensek számára különféle mechanizmusok léteznek az intentek célbajuttatására:

- Egy Intent objektum kerül átadásra a Context.startActivity() vagy Activity.startActivityForResult() metódusoknak, ha egy új tevékenység indítására vagy egy már létező tevékenység valamilyen új műveletének elindítására van szükség. (az objektumot az Activity.setResult() metódusnak is át lehet adni, hogy a startActivityForResult() metódust tevékenység számára adjunk vissza információt.)
- Egy Intent objektum kerül átadásra a Context.startService() metódusnak, hogy elindítsunk egy szolgáltatást vagy egy már létező szolgáltatás számára új utasításokat adjunk. Hasonlóképp, az intent objektumot a Context.bindService() metódusnak is átadhatjuk a hívó komponens és a célszolgáltatás közötti kapcsolat kialakítására. Opcionálisan magát a szolgáltatást is elindíthatja, ha az még nem fut.
- Intent objektumot bármilyen üzenetszóró metódusnak is átadhatunk (például Context.sendBroadcast(), Context.sendOrderedBroadcast(), vagy Context.sendStickyBroadcast()), amit aztán minden érintett üzenetszórás fogadó megkap. Sokféle üzenetszórás a rendszerkódból származik.

A fenti esetek mindegyikében az Android rendszer megkeresi a megfelelő tevékenységet, szolgáltatást vagy üzenetszórás fogadók halmazát, ami az intentre válaszol, és példányosítja is azokat, amennyiben szükséges. Nincs átfedés ezen üzenetközvetítő rendszerek között: Az üzenetszórás intenteket csak üzenetszórás fogadókhoz továbbítja, és soha nem tevékenységekhez vagy szolgáltatásokhoz. A startActivity() metódusnak átadott intent csak egy tevékenységhez kerülhet, és soha nem szolgáltatáshoz, vagy üzenetszórás fogadóhoz, stb.

A következő oldalakon az Intent objektumok leírása található. Azután azokat a szabályokat mutatjuk be, amelyeket az Android alkalmaz az intentek komponensekre való leképezéséhez — hogyan találja meg, hogy melyik intent üzenetet melyik komponensnek kell megkapnia. Azokat az intenteket amelyek nem mondják meg explicit módon, hogy mi a célkomponensük, ez az eljárás teszteli le intent szűrőkkel, hogy megtalálja a lehetséges célpontjaikat.

Intent Objectumok

Egy Intent objektum nem más, mint adatok egy halmaza. Olyan információkat tartalmaz, amelyek az intent objektumot megkapó komponens számára érdekesek (pl. a végrehajtandó művelet vagy feldolgozásra váró adat címe). Ezen felül tartalmaz az Android rendszer számára lényeges információkat is (pl. az intent célpontjául szolgáló komponens kategóriája, vagy a céltevékenység elindítására szolgáló utasítások).

Egy intent szűrő az IntentFilter osztály egy példánya. Mivel azonban az Android rendszernek ismernie kell az adott komponens tulajdonságait mielőtt el tudja azt indítani, az intent szűrőket általában nem a Java kódban inicializálják, hanem az alkalmazás manifest fájljában, (AndroidManifest.xml) <intent-filter> elemekben. (Az egyetlen kivétel ez alól az üzenetszórás fogadókra vonatkozó szűrők, amelyeket dinamikusan regisztrálhatunk a Context.registerReceiver() hívásként, IntentFilter osztály példányosításval.)

Egy szűrő egyaránt tartalmaz egy Intent objektum műveletére, adatára és kategóriájára vonatkozó mezőket. Egy implicit intentet mindhárom területen megvizsgálunk a szűrővel. Ahhoz, hogy a szűrőt birtokló komponens megkapja az Intent objektumot, mindhárom teszten át kell mennie. Ha csak az egyiken is elbukik, az Android rendszer nem fogja átadni az Intent objektumot a komponensnek — legalábbis nem az adott szűrő alapján. Ugyanakkor, mivel egy komponensnek több szűrője is lehet, attól hogy az intent objektum fennakad a komponens egyik szűrőjén, még átmehet egy másikon.

A következőkben részletesen bemutatjuk a három tesztet:

Művelet teszt

A manifest fájlban található <intent-filter> elem a műveleteket <action> alelemekként sorolja fel. Például:

```
<intent-filter ... >
   <action android:name="com.example.project.SHOW_CURRENT" />
   <action android:name="com.example.project.SHOW_RECENT" />
   <action android:name="com.example.project.SHOW_PENDING" />
   ...
</intent-filter>
```

Ahogy a példa is mutatja, míg egy Intent objektum csak egyetlen műveletet nevez meg, egy szűrő akár többet is megnevezhet. A lista nem lehet üres; egy szűrőnek legalább egy <action> elemet tartalmaznia kell, vagy minden intent objektumot blokkolni fog.
Ahhoz, hogy az Intent objektum átmenjen ezen a teszten, a műveletének meg kell egyeznie a szűrőben felsorolt műveletek egyikével. Amenyiben az objektum vagy a szűrő nem ad meg tevékenységet, az eredmény az alábbiak szerint alakul:

- Ha a szűrő nem tartalmaz műveleti listát, az intent objektum művelete semmivel sem tud megegyezni, így minden intent objektum el fog bukni a teszten. Egyetlen intent objektum sem fog átmenni a szűrőn.
- Másrészt, ha az Intent objektum nem ad meg műveletet, akkor automatikusan átmegy a teszten — amennyiben a szűrő legalább egy műveletet tartalmaz.

Kategória teszt

Az <intent-filter> elem kategóriákat is felsorol alelemként. Például:

```
<intent-filter ... >
   <category android:name="android.intent.category.DEFAULT" />
   <category android:name="android.intent.category.BROWSABLE" />
   ...
</intent-filter>
```

Ahhoz, hogy egy intent objektum átmenjen a kategória teszten, az objektumban található összes kategóriának meg kell egyeznie a szűrőben megadott kategóriák valamelyikével. A szűrő tartalmazhat további kategóriákat, de az intent egyetlen kategóriáját sem hagyhatja ki.
Elvileg így egy kategóriák nélküli Intent objektumnak mindig át kellene mennie a teszten, a szűrő tartalmától függetlenül. Ez javarészt igaz is. Azonban, egyetlen kivétellel, az Android minden, a startActivity() metódusnak átadott implicit intentet úgy kezel, mintha azok legalább egy kategóriával rendelkeznének: "android.intent.category.DEFAULT" (a CATEGORY_DEFAULT constans). Így az implicit intent objektumok fogadására hajlandó tevékenységek szűrőinek tartalmaznia kell az "android.intent.category.DEFAULT" elemet. (Az "android.intent.action.MAIN" és "android.intent.category.LAUNCHER" beállításokat tartalmazó szűrők kivételek. Ezek olyan tevékenységeket jelölnek amelyek új feladatok kezdenek és az inditóképernyőn jelennek meg. Tartalmazhatják az "android.intent.category.DEFAULT" értéket is a kategória listában, de nem szükséges.)

Adat teszt

Akárcsak a műveletek és kategóriák esetében, az adat specifikációk is alelemként jelennek meg a szűrőben. És csakúgy, mint a fenti két esetben, itt is több alelem szerepelhet a listában, vagy akár nulla is. Például:

```
<intent-filter ... >
   <data android:mimeType="video/mpeg" android:scheme="http" ... />
   <data android:mimeType="audio/mpeg" android:scheme="http" ... />
   ...
</intent-filter>
```

Minden <data> elem egy URI-t és egy adat típust (MIME média típus) ad meg. Az URI minden részének külön attribútuma van — scheme, host, port, and path :

scheme://host:port/path

Például, az alábbi URI-ban,

content://com.example.project:200/folder/subfolder/etc

a scheme (séma) "content", a host az "com.example.project", a port "200", és a path (útvonal) pedig "folder/subfolder/etc". A host és a port együtt adja meg az URI hatáskörét; amennyiben nincs host megadva, akkor a port figyelmen kívül marad.

Az attribútumok mindegyike opcionális, de egymástól nem függetlenek: Ahhoz, hogy egy hatáskör értelmes legyen, a sémát is meg kell adni. Ahhoz, hogy az útvonal értelmes legyen, a sémát és a hatáskört egyaránt meg kell adni.

Amikor egy Intent objektum URI-ja összehasonlításra kerül egy szűrő URI specifikációjával, voltaképpen a szűrőben ténylegesen megadott URI attribútumok kerülnek összehasonlításra. Például, ha a szűrő csak a sémát adja meg, akkor a megfelelő sémával rendelkező összes URI át fog menni a teszten. Ha a szűrő egy sémát és hatáskört ad meg, de útvonalat nem, az összes URI át fog menni a teszten, ha megfelelő sémával és hatáskörrel rendelkezik, útvonaltól függetlenül. Ha a szűrő sémát, hatáskört és útvonalat egyaránt megad, akkor csak az ugyanolyan sémával, hatáskörrel és útvonallal rendelkező URI-k fognak átmenni. Ugyanakkor, egy útvonal specifikáció tartalmazhat joker karaktereket is, ha csak részleges útvonal illeszkedésre van szükség.
A <data> elem type attribútuma adja meg az adat MIME típusát. Ez jóval gyakoribb a szűrőkben, mint az URI használata. Az Intent objektum és a szűrő egyaránt használhat "*" joker karaktert az altípus mezőben — például "text/*" vagy "audio/*" — altípus egyezésekhez.

Az adat teszt egyaránt összehasonlítja az URI-t és az adattípust is az Intent objektum és a szűrő között. A szabályok a következők:

a. Az az Intent objektum, amely nem tartalmaz sem URI-t sem adat típust csak abban az esetben megy át a teszten, ha az sem tartalmaz semmilyen URI-t vagy adat típust.

b. Az az Intent objektum, amely tartalmaz URI-t de nem tartalmaz adat típust (és nem is lehet kikövetkeztetni az URI-ból valamilyen típust) csak abban az esetben megy át a teszten, ha az URI-ja megegyezik a szűrőben megadott URI-k valamelyikével és a szűrő szintén nem tartalmaz egyetlen típus specifikációt sem. Ez az eset fog bekövetkezni a mailto: és tel: típusú URI-knál, amelyek nem hivatkoznak tényleges adatra.

c. Az az Intent objektum, amely tartalmaz adat típust, de nem tartalmaz URI-t csak abban az esetben megy át a teszten, ha a szűrő ugyanazt az adattípust felsorolja, és nem tartalmaz semmilyen URI-t.

d. Az az Intent objektum, amely egyaránt tartalmaz URI-t és adat típust (vagy az adat típus kikövetkeztethető az URI-ból) csak abban az esetben megy át a teszten, ha a szűrő felsorolja magában az adott adat típust. Átmegy a teszt URI részén akkor is ha az URI-ja megegyezik a szűrőben felsorolt URI-k valamelyikével, vagy egy content: vagy file: URI-t tartalmaz és a szűrő nem tartalmaz egyetlen URI-t sem. Más szóval, egy komponensről azt feltételezi a rendszer, hogy támogatja a content: és file: adatokat ha a szűrője csak adatípusokat sorol fel.

Ha egy intent több tevékenység vagy szolgáltatás szűrőjén is átmegy, akkor a felhasználó számára is érkezhet kérdés, hogy melyik komponenst szeretné aktiválni. Ha egyetlen célpont sem található, kivétel keletkezik.

Általános esetek

Az adat tesztben leírt utolsó szabály (d. szabály) azt az elvárást tükrözi, hogy a komponensek képesek lokális adatok fogadására fájlból vagy tartalomszolgáltatótól. Így a szűrőiknek elég ha csak egy adattípust adnak meg, és nem szükséges explicit megnevezniük a content: és file: sémákat. Ez egy tipikus eset. Mint például az alábbi <data> elem, ami azt mondja meg az Androidnak, hogy a komponens képes egy tartalomszolgáltatótól képadatokat fogadni és megjeleníteni:

```
<data android:mimeType="image/*" />
```

Mivel a legtöbb elérhető adatot tartalomszolgáltatók nyújtják, talán a leggyakoribb szűrők azok, amelyek adattípust meghatároznak, de URI-t nem.

Egy másik gyakori konfiguráció, ha a szűrő sémát és adat típust tartalmaz. Az alábbi <data> elem például azt mondja meg az Androidnak hogy a komponens képes a hálózatról video adatok fogadására és megjelenítésére:

```
<data android:scheme="http" android:type="video/*" />
```

Gondoljuk át például, hogy mit csinál egy böngésző alkalmazás, amikor a felhasználó rákattint egy linkre a megjelenített weboldalon. Először megpróbálja megjeleníteni az adatot (ami sikerülne is, ha a link HTML oldalra mutatna). Ha nem tudja megjeleníteni az adatot, összeállít egy implicit intent objektumot a séma és adat típussal, majd megpróbál elindítani egy olyan alkalmazást, ami el tudja végezni a munkát. Ha nincs jelentkező, akkor megkéri a letöltés kezelőt, hogy töltse le az adatot. Az egy tartalomszolgáltató fennhatósága alá helyezi az adatot, így tevékenységek egy feltehetőleg nagyobb csoportja (azok, amelyeknek a szűrői csak adattípusokat neveznek meg) válaszolhat a feladatra.

2.3. AndroidManifest.xml

Mielőtt az Android rendszer elindíthat egy tevékenység komponenst, tudnia kell, hogy a komponens létezik azáltal, hogy felolvassa az alkalmazás AndroidManifest.xml fájlját (a "manifest" fájlt). Az alkalmazásunknak minden komponensét deklarálnia kell ebben a fájlban, aminek az alkalmazás könyvtárának gyökerében kell elhelyezkednie.

A manifest egy sor további dolgot is ellát az alkalmazás komponenseinek deklarálásán túl, mint például:

- Leírja az alkalmazás számára szükséges felhasználói jogosultságokat, mint például az Internet elérés, vagy névjegyzék-olvasási engedély.
- Deklarálja az alkalmazás számára szükséges minimum API szintet, az alkalmazás alapján használt API-k alapján.
- Deklarálja az alkalmazás számára szükséges hardver és szoftver paramétereket, mint például a kamera, bluetooth szolgáltatások, vagy multitouch kijelző.
- Az alkalmazás számára szükséges API könyvtárakat (az Android keretrendszer API-kon túl), mint például a Google Maps könyvtár.

Komponensek deklarálása

A manifest fájl elsődleges feladata a rendszer tájékoztatása az alkalmazás komponenseiről. Például a manifest fájl a következőképp deklarálhat egy tevékenységet:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application android:icon="@drawable/app_icon.png" ... >
    <activity android:name="com.example.project.ExampleActivity"
              android:label="@string/example_label" ... >
    </activity>
    ...
  </application>
</manifest>
```

Az <application> elemben az android:icon attribútum az alkalmazás által használt ikont lokalizálja.

Az <activity> elemben az android:name attribútum az Activity osztályból származtatott alosztály teljes minősített nevét tartalmazza, az android:label attribútum pedig a tevékenység a felhasználó által látható címkét adja meg.

Egy alkalmazás összes komponenseit az alábbiak szerint kell deklarálni:

- <activity> elemek az tevékenységekhez
- <service> elemek a szolgáltatásokhoz
- <receiver> elemek az üzenetszórás fogadókhoz
- <provider> elemek a tartalomszolgáltatókhoz

Azok a tevékenységek, szolgáltatások és tartalomszolgáltatások, amelyeket meghivatkozunk a forráskódunkban, de nem deklarálunk a manifest fájlban nem láthatóak a rendszer számára, így ebből kifolyólag soha nem futhatnak le. Ugyanakkor az üzenetszórás fogadókat deklarálhatunk a manifest fájlban vagy dinamikusan a forráskódban is (mint BroadcastReceiver objektumok) amiket registerReceiver() metódus hívásával regisztrálhatunk a rendszerben.

2.4. Felhasználói felület

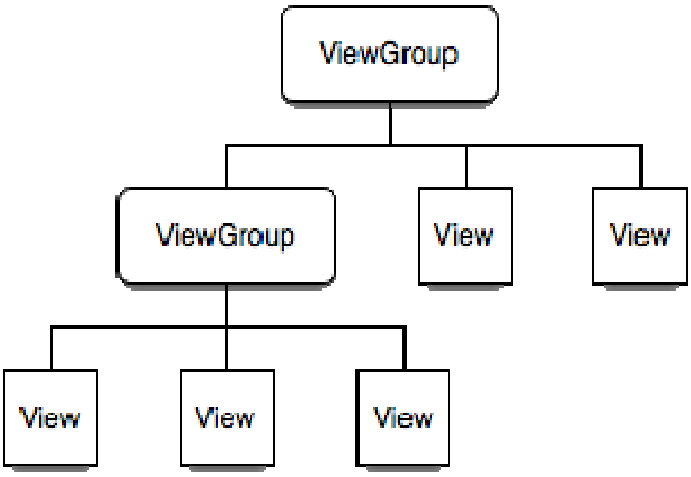
Egy Android alkalmazás esetén a felhasználói felület View és ViewGroup objektumból épül fel. Sokféle típusú nézet és nézetcsoport létezik, melyeknek mindegyike a View osztály leszármazottja.

Az Android platform alap építőkövei a View objektumok. A View osztály a „widget”-eknek nevezett alosztályok őse, amik teljesen implementált UI (User Interface – Felhasználói Felület) objektumok, mint egy szövegmező vagy gomb. A ViewGroup osztály pedig a „layout”-oknak (elrendezéseknek) nevezett alosztályok őse, amelyek különféle elrendezési stratégiákat kínálnak, mint a lineáris, tabuláris, relatív elrendezés, stb.

Egy View objektum egy olyan adatstruktúra, aminek mezői az elrendezési paramétereket és a kijelző egy megadott téglalap alapú területének tartalmát tartalmazzák. Egy View objektum kezeli a saját méretezését, elrendezését, rajzolását, fókusz kezelését, görgetését, billentyű/érintés kezelését a kijelző azon téglalap alapú területén, amelyet elfoglal. A felhasználói felület objektumaként a View ugyanakkor a felhasználó és az eszköz közötti interakciós pont.

2.4.1. Megjelenítő elemek hieachiája

Az Android platformon egy Tevékenység (Activity) felhasználói felületét View és ViewGroup csomópontokkal határozhatunk meg, ahogy ezt a lenti ábra is mutatja. Ez a hierarchia fa lehet nagyon egyszerű vagy akár nagyon bonyolult is az adott alkalmazástól függően. Könnyedán felépíthetünk egyet az Android előre definiált widgetei és layoutjai, valamit a saját magunk által létrehozott View-k segítségével.



Ahhoz, hogy az adott view hierarchia fát hozzácsatoljon a kijelzőhöz, a Tevékenységnek meg kell hívnia a setContentView() metódust, amiben átadja a gyökér csomópont referenciáját. Az Android rendszer megkapja ezt a referenciát, majd ezt felhasználva méretezi és rajzolja ki a fát. A hierarchia gyökér csomópontja megkéri a gyermek csomópontjait, hogy rajzolják meg önmagukat – akik azután meghívják a saját gyermekcsomópontjaikat, és így tovább. A gyerek csomópontok kérhetnek pozíciót és méretet a szülőn belül, de a végső döntés a szülőobjektum kezében van az egyes gyerekek méretéről. Az Android inorder bejárással értékeli ki az egyes elemeket (a hierarchia fa tetejétől kezdve), így hozza létre a View-kat és csatolja hozzá a szülő(k)höz. A kirajzolási folyamat inorder jellege miatt ha vannak olyan elemek a fában, amelyeknek a területe/pozíciója átfedi egymást, akkor a legutoljára kirajzolandó objektum el fogja takarni az összes alatta lévő objektumot az adott területen.

Elrendezés

A legáltalánosabb módja az definiálásának és a view herarchia megadásának egy XML fájl segítségével történik. Az XML egy ember által olvasható formátumot nyújt, mint a HTML. Minden elem az XML-ben vagy egy View vagy egy ViewGroup objektum (vagy annak leszármazottja). A View objektumot a fa levélelemei, a ViewGroup objektumok pedig a fa ágai (lásd fenn a View Hierarchia ábrát).

Egy XML elem neve a megfelelő Java osztály nevét tükrözi. Így egy <TextView> elem egy TextView objektumot, míg egy <LinearLayout> elem egy LinearLayout viewgroup objektumot hoz létre a UI-n. Amikor betöltünk egy elrendezést, az Android rendszer az elrendezés elemeinek megfelelően inicializálja ezeket a futási-idejű objektumokat.

Például, egy egyszerű függőleges elrendezés egy text view-al és egy gombbal a következőképpen néz ki:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:orientation="vertical" >
    <TextView android:id="@+id/text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Hello, I am a Button" />
</LinearLayout>
```

Figyeljük meg, hogy a LinearLayout elem egyaránt tartalmazza a TextView és Button elemeket is. Lehetőségünk van egy másik LinearLayout (vagy bármilyen más view group típus) beágyazására is itt, ha növelni szeretnénk a view hierarchiát és összetettebb elrendezéseket készíteni.

Felhasználói felületek építésével kapcsolatosan részletesebb információkat talál az Elrendezések Deklarálása c. fejezetben.

Tipp: View és ViewGroup objektumokat Java kódból is kirajzolhatunk az addView(View) metódusokkal, amiknek segítségével dinamikusan adhatunk a nézethez új View és ViewGroup objektumokat.

A view objektumaink elrendezéséhez többféle módszer is létezik. A különböző view groupok segítségével végtelen számú lehetőségünk van a gyermek view és view groupok elrendezésére.

Az Android által kínált előre definiált view groupok (amelyeket elrendezéseknek nevezünk) közé tartozik a LinearLayout, RelativeLayout, TableLayout, GridLayout és mások. Ezek mindegyike egyedi elrendezési paramétereket kínál, amelyek segítségével definiálhatjuk a gyermek view-k pozícióját és elrendezési struktúráját.

Widget-ek

A widget egy View objektum, amely kapcsolódási pontot nyújt a rendszer és a felhasználó közötti interakcióra. Az Android egy sor leimplementált widget-et kínál, például gombok, szövegbeviteli mezők, jelölőnégyzetek, stb. formájában, amikkel gyorsan fel tudunk építeni egy felhasználói felületet. Néhány összetettebb, az Android által kínált widget például a dátum kiválasztó, óra, vagy kicsinyítés/nagyítás funkciók. Természetesen nem csak az Android által kínált widget-eket használhatjuk. Ha egyedi elemekre van szükségünk, saját View objektum létrehozásával, vagy a már létező widget-ek kiterjesztésével és kombinálásával megalkothatjuk a saját widget-einket.

Az Android által kínált összes widget megtekintéséhez az android.widget csomagot érdemes átnézni.

UI Események

Miután hozzáadtunk néhány View-t/widget-et a felhasználói felülethez, nyílván szeretnénk tudni róla, amikor a felhasználó használja őket, hogy különféle műveleteket végezhessünk azok hatására. Hogy értesítést kapjunk a UI eseményekről, az alábbi két dolog egyikére van szükségünk:

- **Definiáljunk egy esemény figyelőt és csatoljuk hozzá a View-hoz.** Ez a gyakoribb módja az eseményekre való figyelésnek. A View osztály tartalmaz beágyazott interfészek egy gyűjteményét (On<valami>Listener nevekkel), amelyek mindegyikének van egy visszahívó metódusa is On<valami>() néven. Például, View.OnClickListener (a View-n történő kattintások kezeléséhez), View.OnTouchListener (a View-n történő érintőképernyő események kezeléséhez), és View.OnKeyListener (a View-n történő eszköz gombnyomások

kezeléséhez). Így ha azt akarjuk, hogy a View-nk értesítve legyen ha „rákattintanak” (például kiválaszt a felhasználó rajta egy gombot), implementáljuk le az OnClickListener interfészt és definiáljuk az onClick() visszahívó metódusát (ahol megadhatjuk a kattintás eseményre történő utesításokat), és csatoljuk hozzá a View-hoz a setOnClickListener() metódussal.

- **Definiáljunk felül egy már létező visszahívó metódust a View-ban.** Erre akkor lehet szükségünk, ha saját View osztályt implementáltunk és bizonyos eseményekre szeretnénk figyelni azon belül. Lekezelhető példaeseményként meg lehet itt említeni a képernyő megérintését (onTouchEvent()), a trackball mozgatását (onTrackballEvent()), vagy amikor az eszköz egyik gombját lenyomja a felhasználó (onKeyDown()). Ennek segítségével a saját View-nkon belül minden eseményre külön meghatározhatjuk a megfelelő viselkedést és eldönthetjük, hogy az adott eseményt esetleg továbbítjuk-e egy másik gyermek View-nak. Hangsúlyozzuk, hogy ezek a visszahívó metódusok a View osztályhoz tartoznak, így az egyetlen esélyünk, hogy definiálhassuk őket az, ha saját komponenst hozunk létre.

Menük

A menü egy másik fontos része az alkalmazás felhasználói felületének. A menük egy megbízható interfészt kínálnak a felhasználónak, amiken keresztül elérheti az alkalmazás különböző funkcióit és beállításait. Legáltalánosabb alkalmazás menü az eszköz MENU gombjának megnyomásával érhető el. Ezen túl azonban lehetőségünk van Környezetfüggő Menük (Context Menu) létrehozására is, amelyeket akkor jeleníthetünk meg amikor a felhasználó lenyomva tartja a gombot/ujját egy elemen.

A menüket szintén a View hierarchiával lehet strukturálni, de ez esetben nem mi definiáljuk a stuktúrát. Ehelyett az onCreateOptionsMenu() vagy onCreateContextMenu() visszahívó metódusokat definiáljuk a Tevékenységünkhöz, majd megadjuk azokat az elemeket, amiket a menüben szerepeltetni szeretnénk. A megfelelő időben az Android automatikusan létrehozza a szükséges View hierarchiát a menühöz és kirajzolja az egyes menüelemeket.

A menük szintén kezelik a saját eseményeiket, így nem szükséges eseményfigyelőket regisztrálni a menüelemekhez. Amikor egy menüelem kiválasztásra kerül, az

onOptionsItemSelected() vagy onContextItemSelected() metódus kerül meghívásra a keretrendszer által.

És akárcsak az alkalmazás elrendezés esetében, itt is lehetőségünk van XML fájlban megadni az egyes menüelemeket.

2.4.2. UI definiálás XML-ben

Az elrendezés az a szerkezet, amin keresztül a felhasználó kapcsolatba léphet egy Tevékenységgel. Ez definiálja a felhasználó számára megjelenő elemek elhelyezését. Egy elrendezést kétféleképpen deklarálhatunk:

- **UI elemek deklarálása XML-ben.** Az Android egy egyértelmű XML nyelvezetet kínál, amely összhangban van a View osztályokkal és alosztályokkal, mint például a widget-ek és elrendezések.
- **Elrendezés elemek példányosítása futási időben**. Az alkalmazásunk futás közben kódból is képes View és ViewGroup objektumokat létrehozni (és a tulajdonságaikat változtatni).

Az Android keretrendszer elég rugalmas ahhoz, hogy a fenti két módból egyiket, vagy akár egyszerre mindkettőt felhasználjuk az alkalmazásunk felhasználói felületének kialakításához és kezeléséhez. Például, megadhatjuk XML-ben az alkalmazás alapvető elrendezéseit, beleértve az egyes elemeket és tulajdonságaikat, amelyek majd megjelennek a képernyőn. Ezek után írhatunk olyan kódrészeket az alkalmazásba, amelyek futási időben változtatják az XML-ben deklarált elemek paramétereit.

Az XML-ben történő UI megadás azzal az előnnyel jár, hogy jobban el lehet különíteni az alkalmazás megjelenítési rétegét a kód többi részétől. A UI leírások az alkalmazás kódján kívül, egy külön fájlban vannak, ami azt jelenti, hogy anélkül lehet megváltoztatni, hogy a forráskódot módosítani, és emiatt újrafordítani kellene. Például, megadhatunk az egyes képernyő helyzetekhez, méretekhez vagy nyelvekhez külön-külön XML elrendezést. Ezen felül az XML-ben deklarált elrendezéseknek az is az előnye, hogy könnyebben tudjuk olvasni a UI struktúráját,

így könnyebben tudjuk az esetleges hibákat is javítani. Ebből kifolyólag most az XML-ben történő felhasználói felület elrendezés deklarálásra helyezzük a hangsúlyt.

Általánosságban elmondható, hogy a UI deklaráláshoz használt XML nyelvezet szorosan követi az osztályok és metódusok elnevezéseit, ahol az elemek nevei az osztályok neveinek, míg az attribútumok nevei a metódusok neveinek felelnek meg. Ez a megfelelés sokszor annyira direkt, hogy sokszor egyértelműen meg tudjuk mondani, hogy melyik XML attribútum melyik osztálymetódusnak, vagy épp melyik XML elem melyik osztálynak felel meg. Ugyanakkor figyeljünk rá, hogy a nyelvezetben nem minden esetben azonos a két név. Bizonyos esetekben előfordulnak kisebb eltérések. Például az EditText elemnek van egy text attribútuma, amely az EditText.setText() metódusnak felel meg.

UI leírás XML-ben

Az Android XML nyelvezetével gyorsan megtervezhetjük a különféle UI elrendezéseket és tartalmazott képernyő elemeket, hasonló módon a HTML-ben való weboldal tervezéshez — beágyazott elemek sorozatával.

Minden layout fájlnak pontosan egy gyökérelemet kell tartalmaznia, ami vagy egy View vagy egy ViewGroup objektum kell, hogy legyen. Miután definiáltuk a gyökérelemet, további layout objektumokat vagy widgeteket adhatunk hozzá gyermek elemként, így építve fel a View hierarchiát amely meghatározza az elrendezésünket. Az alábbi példában egy olyan XML elrendezés látszik, amely a LinearLayout elrendezést használja fel egy TextView és egy Button tartalmazására:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="vertical" >
   <TextView
      android:id="@+id/text"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hello, I am a TextView" />
   <Button
      android:id="@+id/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hello, I am a Button" />
</LinearLayout>
```

Miután megadtuk az elrendezésünket XML-ben, mentsük el a .xml kiterjesztésű fájt az Android projekt res/layout/ könyvtárában.

2.5. Alkalmazás erőforrások

Lehetőleg minden esetben szervezzünk ki minden erőforrást – például képek, stringek - az alkalmazásunk kódjából külön helyre, hogy függetlenül kezelhessük őket a kód többi részétől. Az erőforrások kiszervezésével arra is lehetőség nyílik, hogy alternatívákat biztosítsunk az alkalmazásunknak a különféle eszközkonfigurációkhoz, például más nyelvek vagy képernyőméretek figyelembe vételével. Ez egyre fontosabb dolog, mivel egyre többféle Android eszköz jelenik meg a piacon. Ahhoz, hogy különféle konfigurációkkal is kompatibilis legyen az alkalmazásunk, rendszereznünk kell az erőforrásokat a projekt res/ könyvtárában alkönyvtárak segítségével, amelyek típusonként vagy konfigurációnként csoportosítanak.

Minden erőforráshoz megadhatunk egy *alapértelmezett* és több *alternatív* erőforrást az alkalmazásban:

- Az alapértelmezett erőforrásokat eszközkonfigurációtól függetlenül fogja használni az alkalmazás, vagy ha nincs az adott konfigurációhoz megadott megfelelő alternatíva.

- Az alternatív erőforrásokat adott konkrét konfigurációhoz tervezzük. Az erőforrás csoportot tartalmazó könyvtár nevét kiegészítve egy konfigurációs jelzővel megadjuk, hogy milyen konfiguráció esetén használja fel az alkalmazásunk a könyvtárban lévő alternatív erőforrásokat. Például, míg az alapértelmezett UI elrendezést a res/layout/ könyvtárban tároljuk, egy másik UI elrendezést is megadhatunk arra az esetre, ha a képernyőt elforgatják vízszintes síkra (landscape orientation), amit a res/layout-land/ könyvtárban fogunk elmenteni. Az Android automatikusan a megfelelő erőforrásokat fogja felhasználni úgy, hogy megpróbálja illeszteni az erőforrás könyvtárak neveit az eszköz aktuális konfigurációjához.

1. Ábra. Két eszköz az alapértelmezett erőforrásokat használja.

2. Ábra. Két eszköz, az egyik alternatív erőforrásokat használ.

Az 1. Ábra azt mutatja, hogyan lehet alapértelmezett erőforrások egy gyűjteményét különböző eszközökön felhasználni, ha nincs elérhető alternatíva. A 2. Ábra pedig azt példázza, hogy ugyanaz az alkalmazás hogyan képes alternatív erőforrások segítségével különböző konfigurációkon is helyesen megjelenni.

2.5.1. Erőforrás tipusok

Ebben a fejezetben röviden felsorolunk néhány erőforrás típust, amiket az alkalmazásunkhoz felhasználhatunk.

- Animáció erőforrások: Előre meghatározott animációkat definiálnak.
  A tween típusú animációkat a res/anim/ könyvtárban tároljuk, és az R.anim osztályon

keresztül érjük el. A frame típusú animációkat a res/drawable/ könyvtárban tároljuk és az R.drawable osztályon keresztül érjük el.

- Színkezelő erőforrások: Színkezelő erőforrásokat definiálnak, amelyek az aktuális View állapot alapján változnak. A res/color/ könyvtárban tároljuk és az R.color osztályon keresztül érjük el.
- Kirajzolható erőforrások: Különböző XML vagy bitmap alapú grafikákat definiálnak. A res/drawable/ könyvtárban tároljuk és az R.drawable osztályon keresztül érjük el.
- Elrendezés erőforrások: Az alkalmazás felhasználói felületének elrendezését definiálják. A res/layout/ könyvtárban tároljuk és az R.layout osztályon keresztül érjük el.
- Menü erőforrások: Az alkalmazás menük tartalmát definiálják. A res/menu/ könyvtárban tároljuk és az R.menu osztályon keresztül érjük el.
- String erőforrások: Stringeket, String tömböket definiálnak (formázással és stílusokkal együtt). A res/values/ könyvtárban tároljuk és az R.string, R.array, és R.plurals osztályokon keresztül érjük el.
- Stílus erőforrások: A UI elemek kinézetét és formátumát definiálják. A res/values/ könyvtárban tároljuk és az R.style osztályon keresztül érjük el.

További erőforrások:
Különböző értékeket (pl. boolean, integer, dimenzió, szín, tömb) definiálnak. A res/values/ könyvtárban tároljuk, de egyedi R alosztályokon keresztül érjük el őket (pl. R.bool, R.integer, R.dimen, stb.).

2.6. Adattárolás

Az Android különböző lehetőségeket kínál az alkalmazás perzisztens adatainak eltárolására. A megfelelő megoldást az adott igények alapján választjuk ki, mint pl. az adatot csak az alkalmazás érhesse el, vagy más alkalmazások (vagy a felhasználó) is, mennyi helyre van szüksége az adatoknak, stb.

Az alábbi adattárolási lehetőségeink vannak:

- Megosztott preferenciák (SharedPreferences): Privát primitív adatok tárolása kulcs-érték párokkal.
- Belső tár: Privát adatok tárolása eszköz belső tárjában.
- Külső tár: Publikus adatok tárolása a megosztott külső tárban.
- SQLite adatbázisok: Strukturált adatok tárolása privát adatbázisban.
- Hálózati kapcsolat: Adatok tárolása a weben saját hálózati szerverrel.

Az Android lehetőséget kínál arra is, hogy a privát adatainkat is megosszuk más alkalmazásokkal – tartalomszolgáltatók segítségével. A tartalomszolgáltató egy opcionális komponens, amely írás/olvasás jogokat kínál az alkalmazásunk adataira, az általunk beállított megkötések figyelembevételével. Részletekért lásd a 2.2.4. Tartalomszolgáltatók című fejezetet.

Megosztott preferenciák használata

A SharedPreferences osztály egy általános keretrendszert kínál, amelynek segítségével elmenthetünk és visszaolvashatunk primitív adattípus kulcs-érték párokat. A SharedPreferences segítségével bármilyen primitív adatot elmenthetünk: boolean, float, int, long, és string. Ezek az adatok megmaradnak két indítás között.

A SharedPreferences objektum megfogásához az alábbi két módszer egyikét kell használnunk:

- getSharedPreferences() – Ezt használjuk akkor, ha több preferencia, névvel azonosított preferenciafájlra van szükségünk. A nevet az első paraméterben adjuk át a metódusnak.
- getPreferences() – Ezt használjuk akkor, ha a Tevékenységünknek egy preferencia fájlra van szüksége. Mivel ez lesz az egyetlen preferencia fájl a Tevékenységnek, nem adunk meg nevet.

Például:

```
// Restore preferences
SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
boolean silent = settings.getBoolean("silentMode", false);
setSilent(silent);

// Save preferences
SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
SharedPreferences.Editor editor = settings.edit();
editor.putBoolean("silentMode", mSilentMode);

// Commit the edits!
editor.commit();
```

Belső tár használata

Fájlokat közvetlenül az eszköz belső tárjába is menthetünk. Alapértelmezésként a belső tárba mentett fájlok kizárólagosak az alkalmazás számára, és más alkalmazások nem érhetik el (és a felhasználó sem). Amikor a felhasználó eltávoltítja az alkalmazást az eszközről, ezek a fájlok törlődnek.

Fájl létrehozása és írása a belső tárban:

1. Hívjuk meg az openFileOutput() metódust a fájl nevével és műveleti móddal. Ez visszaad nekünk egy FileOutputStream objektumot.
2. A fájlba írni a write() metódussal tudunk.
3. Az adatfolyam lezárásához a close() metódust használjuk.

Például:

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

A MODE_PRIVATE műveleti mód úgy hozza létre a fájlt (vagy írja felül, ha már létezik ilyen nevű fájl), hogy kizárólag az alkalmazásunk fogja tudni elérni. További műveleti módok: MODE_APPEND, MODE_WORLD_READABLE, és MODE_WORLD_WRITEABLE.

Külső tár használata

Minden Android-kompatibilis eszköz támogat egy megosztott „külső tárt”, amit fájlok tárolására használhatunk. Ez lehet akár egy hordozható tárolóeszköz (mint például SD kártya) vagy egy belső (nem hordozható) tár is. A külső tárra mentett fájlok minden alkalmazás számára elérhetőek és a felhasználói is módosíthatja őket, ha USB tárként felcsatolja a számítógépre az Androidos eszközét.

Fájlok elérése külső táron

Ha Level 8 vagy magasabb szintű API-t használunk, akkor a getExternalFilesDir() metódussal nyithatunk meg File objektumot, amely a külső tár mappát képviseli, ahova menthetjük a fájljainkat. Ez a metódus egy type paramétert fogad, ami meghatározza a kívánt alkönyvtár típusát, mint például a DIRECTORY_MUSIC vagy DIRECTORY_RINGTONES (ha null paramétert adunk át, az alkalmazás gyökérkönyvtárát érjük el). Ez a metódus létre is hozza a kívánt könyvtárat, amennyiben szükséges. A könyvtár típusának megadásával biztosítjuk, hogy az Android média scannere megfelelően kategorizálja be a rendszerben található fájlokat (például a csengőhangokat csengőhangként fogja azonosítani és nem zeneként). Ha a felhasználó eltávolít egy alkalmazást, a hozzá tartozó könyvtár és tartalma szintén törlődik.

Ha Level 7 vagy alacsonyabb szintű API-t használunk, akkor a getExternalStorageDirectory() metódussal nyithatunk meg File objektumot, amely a külső tár gyökerét fogja képviselni. Ezek után az alábbi könyvtárba írjuk ki az adatokat:

/Android/data/<*csomag_név*>/files/

Adatbázisok használata

Az Android teljeskörű támogatást nyújt az SQLite adatbázisokhoz. A létrehozott adatbázisokat név alapján bármely osztály el tudja érni az alkalmazáson belül, de nem elérhetőek alkalmazáson kívül.

SQLite adatbázis létrehozásának az ajánlott módja az SQLiteOpenHelper osztály alosztályának implementálásával történik, amin belül felüldefiniáljuk az onCreate() metódust, amiben futtathatjuk a táblákat létrehozó SQLite parancsokat. Például:

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {
   private static final int DATABASE_VERSION = 2;
   private static final String DICTIONARY_TABLE_NAME = "dictionary";
   private static final String DICTIONARY_TABLE_CREATE =
            "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
            KEY_WORD + " TEXT, " +
            KEY_DEFINITION + " TEXT);";


   DictionaryOpenHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }


   @Override
   public void onCreate(SQLiteDatabase db) {
      db.execSQL(DICTIONARY_TABLE_CREATE);
   }
}
```

Az általunk definiált konstruktorral létrehozhatunk egy SQLiteOpenHelper implementáció példányt. Az adatbázisban való íráshoz/olvasáshoz használjuk a getWritableDatabase() és getReadableDatabase() metódusokat. Mindkettő egy SQLiteDatabase objektumot ad vissza, ami az adatbázist képviseli és SQLite műveletekhez biztosít metódusokat.

SQLite lekérdezéseket az SQLiteDatabase query() metódusokkal futtathatunk, amelyek különböző lekérdezés paramétereket fogadnak. Ilyenek például a lekérdezendő tábla neve, projekció, szelekció, oszlopok, csoportosítás, stb. Összetett lekérdezések esetén, amelyek oszlop

aliasokat igényelnek, az SQLiteQueryBuilder osztályt érdemes használnunk, ami jópár kényelmes metódust kínál lekérdezések felépítéséhez.

Minden SQLite lekérdezés egy Cursor objektummal tér vissza, amely a lekérdezés eredményeként előállt sorokra mutat. Minden esetben a Cursor objektummal leszünk képesek navigálni az adatbázis lekérdezés eredménysorai között és írhatjuk/olvashatjuk a sorokat és oszlopkat.

Hálózati kapcsolat használata

A hálózatot is felhasználhatjuk (amennyiben elérhető) adatok tárolására és visszaolvasására a saját web-alapú szolgáltatásainkon. Hálózati műveletekhez az alábbi két csomag osztályait kell használnunk:

- java.net.*
- android.net.*

3. Egy alkalmazás Android Platformon: Voice+

A Google 2007-ben szerezte meg a GrandCentral telekommunikációs céget, melyet átkeresztelt Google Voice-ra. Sok ember soha nem használta ezt a szolgáltatást, mert miután a Google tulajdonába került a cég befagyasztotta a használatát 21 hónapra.

2009 márciusában hozzáadott új funkciókkal a Google Voice szolgáltatás újra elindult:

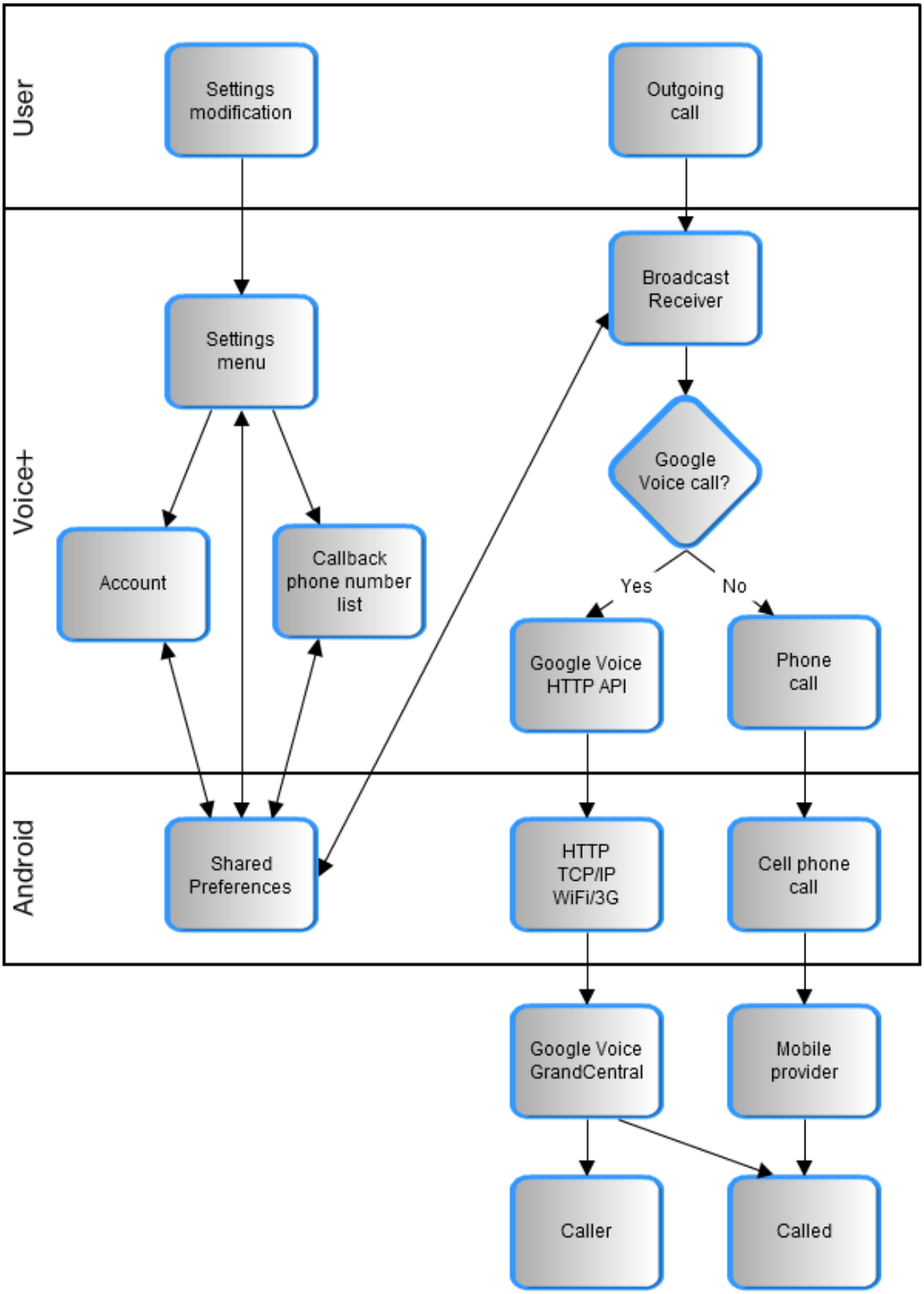
- minden Google felhasználó kaphat egy virtuális telefonszámot mellyel felhívható bármilyen mobil és vezetékes telefonszám.
- szöveges üzenetek kezelése
- hangposta szolgáltatás
- alacsony költségű konferencia és nemzetközi hívások

A Google Voice szolgáltatás be lett integrálva a Google levelezőrendszerébe is, amivel a felhasználók a kontaktlistából felhívhatnak IP alapú hanghívással egy másik felhasználót, vagy az egyenleg feltöltése után akár alacsony költségű nemzetközi mobil vagy vezetékes telefonhívást indíthatnak.

A célom egy olyan Google Voice kliens fejlesztése volt Android platformra, amivel a felhasználók az egész világon a lehető legegyszerűbb módon képesek internet hívást kezdeményezni.

Továbbá célom volt egy olyan alkalmazás kiválasztása, ami az Android rendszernek olyan alap részeit mutatja be, ami más számára is hasznos információval szolgálhat: BroadcastReceiver, Activity, saját készítésű megjelenítő elem, HTTP használata, RegExp kifejezések.

3.1. Voice+ architektúrája



A Voice+ alkalmazás 2 fő komponensből áll:

- Settings menu: ahol a Google account-tal kapcsolatos beállításokat végezhetjük el.
- Kimenő hívás integráció: hívás megkezdésekor eldönthetjük, hogy Google Voice vagy normál GSM hívást szeretnénk indítani.

A Google Voice szolgáltatás igénybevételéhez szükség van egy Google account-ra és egy aktiválás a Voice szolgáltatásra.

3.2. Voice+ felhasználói felülete

Azok a felületek, melyeken keresztül a felhasználó beállíthatja a Google account paramétereit és láthatja a hívás közbeni státuszokat.

3.2.1. Voice+ funkciói

- Beállítások megtekintése
- Google account kezelése
- Visszahívás telefonszámainak megjelenítése
- Telefonszám lista megjelenítése hívás indításakor
- Hivás státusz

3.2.1.1. Beállítások megtekintése

Az alkalmazás elindításakor ezzel a képernyővel találkozunk. Itt be-/kikapcsolhatjuk a kimenő hívás integrációt az Android rendszerbe, beléphetünk a Google account és telefonszámok menüjébe.

3.2.1.2. Google account kezelése

Google account megadása után automatikusan átnavigál a visszahívás telefonszámainak a beállításaira.

3.2.1.3. Visszahívás telefonszámainak megjelenítése

Itt adhatjuk meg azokat a visszahívás telefonszámokat, amelyek megjelennek hívás indításkor.

3.2.1.4. Telefonszám lista megjelenítése hívás indításakor

A hívás indításakor megjelenik először egy felugró menü, melyben fel vannak sorolva milyen hívást lehet kezdeményezni. A lista elején vannak az általunk beállított visszahívás telefonszámok, alatta a normál GSM hívás lehetőségei.

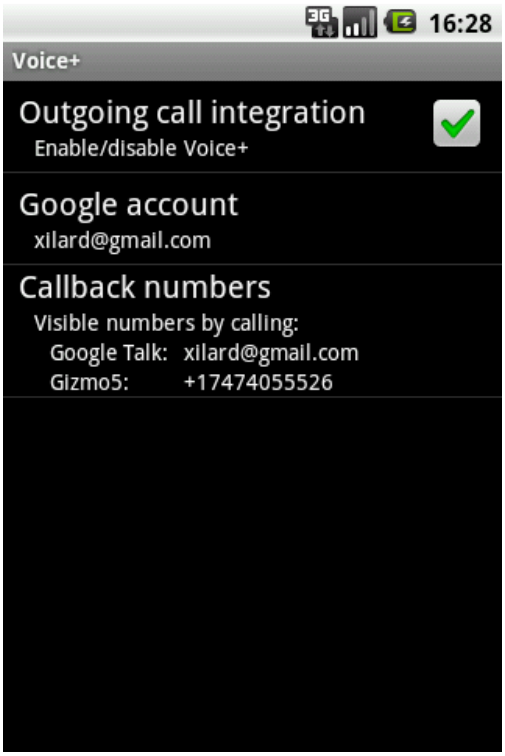
3.2.1.5. Hívás státusz

A hívás megkezdésekor megjelenik egy státuszablak és egy hívást befejező gomb arra az esetre, ha a visszahívás nem következik be.

3.3. Megvalósítás

3.3.1. Beállítások

Az alkalmazás elindításakor a beállítás menüvel találkozunk.



A beállítás menü egy ListActivity-ből származtatott Activity-vel van megvalósítva. A lista elemeit dinamikusan töltjük fel, így eltérhetünk az alapértelmezett megjelenítéstől és saját kinézetet definiálhatunk.

```
public class Settings extends ListActivity {
    GVCallerSettingsListAdapter adapter;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        adapter = new GVCallerSettingsListAdapter(this);
        setListAdapter(adapter);
    }
```

```
    @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        adapter.click(this, position);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        adapter.notifyDataSetChanged();
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

Mivel a lista sorainak tartalma egyedi, a lista adaptere egy BaseAdapter-ből származtatott saját GVCallerSettingsListAdapter adapterosztály. Ezeknek az egyedi soroknak a kinézete xml-ben van leírva a layout-ok között. Az adapterbe a getView() függvény megvalósításával töltjük be a lista sorainak kinézetét. A click() esemény figyelve tudunk reagálni a kiválasztott listaelemre.

3.3.1.1. Kimenő hívás integráció

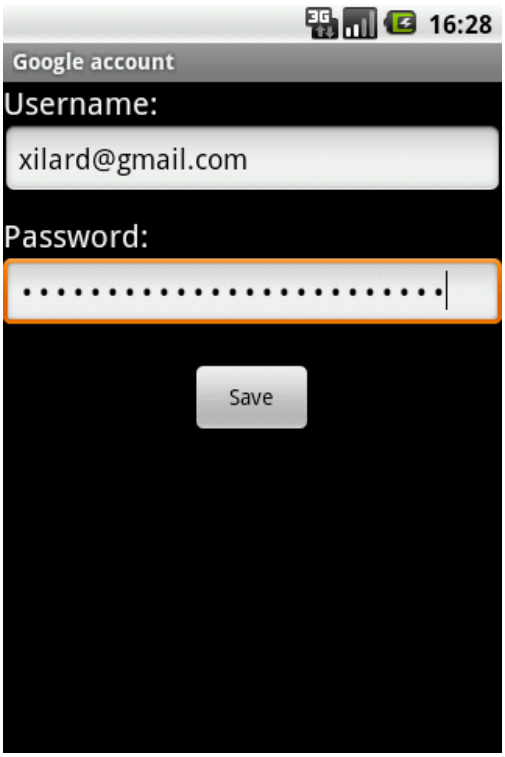
Az első listaelem a kimenő hívás integrációjának kapcsolója, ez nem csinál mást, mint lementi az állapotot a SharedPreferences-be amikor megváltozik.

```
@Override
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
    SharedPreferences settings = context.getSharedPreferences(Constants.PREFERENCE_NAME,
MODE_PRIVATE);
    SharedPreferences.Editor editor = settings.edit();
    editor.putBoolean(Constants.IS_OUTGOING_CALL_INTEGRATION, isChecked);
    editor.commit();
}
```

Majd ha kimenő hívás történik, akkor ugyaninnen fogjuk beolvasni ezt a beállítást.

3.3.1.2. Google account

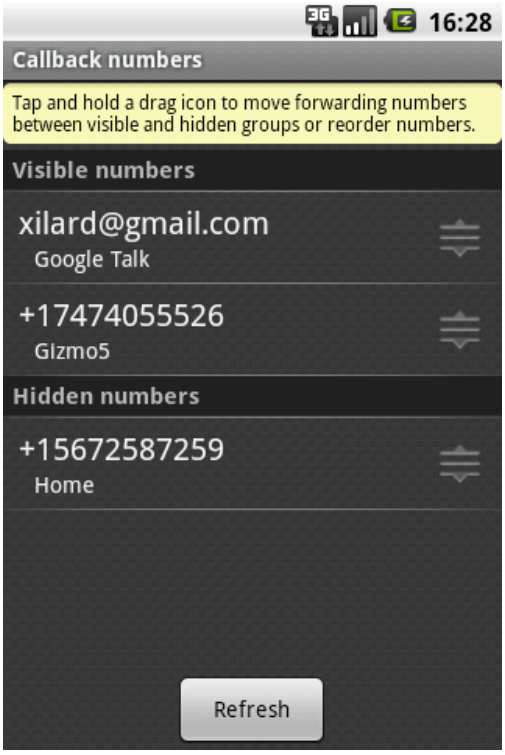
Megadjuk a bejeletkezéshez szükséges felhasználónevet és jelszót:



3.3.1.3. Visszahívás számai

A harmadik listaelemen belül a visszahíváshoz való telefonszámokat rendezgethetjük. A számok a Google fiókból töltődnek le. Ezeket a számokat Voice web-es felületén kellett megadnunk, nevük *„forwarding numbers”* és 2 féle funkciót töltenek be:

- Egyik funkciója a hívásátirányítás: Akkor, ha nem vagyunk elérhetőek a Voice szolgáltatáson keresztül, akkor hova továbbítsa a rendszer a hívást.
- Másik funkciója a visszahívás: Mi ezt a funkciót használjuk most ki. Amikor telefonálunk akkor először a Google GrandCentral felhív minket és miután felvettük tárcsázza azonnal a hívottat.



Ez is egy egyedi lista, ráadásul ennek a sorait még cserélgetni is lehet. A jobb szélen lévő icon megérintése után fel/lehúzható az elem. Szintén a BaseAdapter kiterjesztésével készítettem el ezt a listát.

```
public class SortableListViewAdapter extends BaseAdapter {
    private ArrayList<SortableListItem> listItems;
    private LayoutInflater layoutInflater;
    private float scale;

    public SortableListViewAdapter(Context context, ArrayList<SortableListItem> listItems) {
        layoutInflater = LayoutInflater.from(context);
        this.listItems = listItems;
        scale = context.getResources().getDisplayMetrics().density;
    }

    @Override
    public int getCount() {
        return listItems.size();
    }

    @Override
    public Object getItem(int position) {
```

```
        return listItems.get(position);
    }

    @Override
    public long getItemId(int position) {
        return listItems.get(position).getId();
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        if (convertView == null) {
            convertView = layoutInflater.inflate(R.layout.sortablelistviewitem_template, null);
        }

        LinearLayout template = (LinearLayout) convertView;
        SortableListItem listItem = listItems.get(position);

        if (listItem.isHeader())
        {
            View view = template.findViewById(R.id.separator);
            if (view == null) {
                view = layoutInflater.inflate(R.layout.sortablelistviewitem_separator, null);
                template.removeAllViews();
                template.addView(view, new LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.WRAP_CONTENT));
            }

            TextView text = (TextView) view.findViewById(R.id.item_title);
            text.setText(listItem.getTexts()[0]);

            template.setMinimumHeight(0);
        }
        else if (listItem.isEmpty())
        {
            template.removeAllViews();
            template.setMinimumHeight((int)(58 * scale));
        }
        else
        {
            View view = template.findViewById(R.id.settings);

            if (view == null) {
                view = layoutInflater.inflate(R.layout.sortablelistviewitem_template_row, null);
                template.removeAllViews();
```

```
                template.addView(view, new LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.WRAP_CONTENT));
            }

            String[] texts = listItem.getTexts();

            TextView text = (TextView) view.findViewById(R.id.item_title);
            text.setText(texts[0]);

            text = (TextView) view.findViewById(R.id.item_subtitle);
            text.setText(texts[1]);

            template.setMinimumHeight(0);
        }
        template.requestLayout();

        return convertView;
    }

    public void setItem(int position, SortableListItem listItem) {
        listItems.set(position, listItem);
    }
}
```

A SortableListViewAdapter konstruktorában adható át a SortableListItem lista mely megjelenik a lista elemeiként. Ebben van leírva, hogy egy elem fejléc-e vagy telefonszám és milyen szöveget kell megjeleníteni az adott listaelemben. Csak olyan listaelem helyezhető át, ami nem fejléc.

3.3.2. Hívás indítása

A telefonálást a telefon bármelyik programjából kezdeményezhetjük (telefonkönyv, tárcsázó, egyéb telepített programból), mivel az AndroidManifest.xml-ben meg van határozva intent-filter-rel, hogy minden kimenő hívásra fusson be az alkalmazásunkba. Mégpedig a Caller osztály van megadva, mint receiver.

```
<receiver android:name=".Caller">
    <intent-filter>
        <action android:name="android.intent.action.NEW_OUTGOING_CALL" />
    </intent-filter>
</receiver>
```

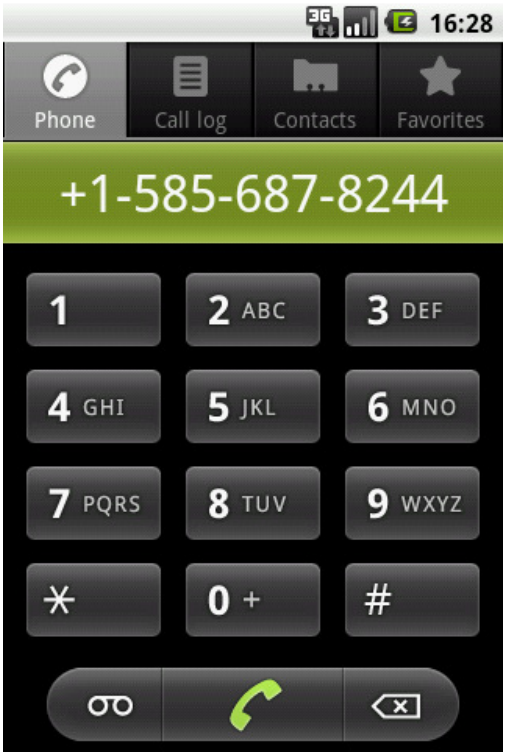
Tehát, a Caller osztály kiterjeszti a BroadcastReceiver abstract osztályt és az onReceive() függvény megvalósításával tudjuk elkapni az összes kimenőhívás kezdeményezést.

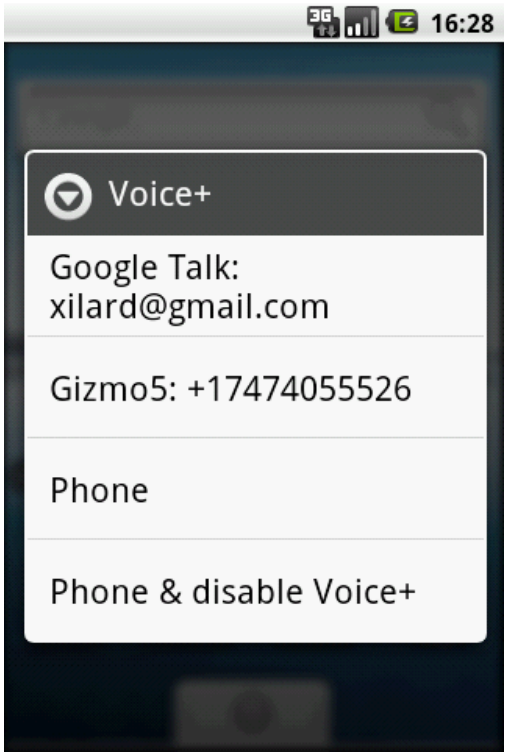
```
public class Caller extends BroadcastReceiver {
    static String lastNumber;
    static long lastTime;

    public static boolean isSkipNext = false;

    @Override
    public void onReceive(Context context, Intent intent) {
        String intentAction = intent.getAction();

        SharedPreferences settings =
context.getSharedPreferences(Constants.PREFERENCE_NAME,
android.content.Context.MODE_PRIVATE);

        if (settings.getBoolean(Constants.IS_OUTGOING_CALL_INTEGRATION, false)) {
            if (intentAction.equals(Intent.ACTION_NEW_OUTGOING_CALL)) {
                String number = intent.getStringExtra(Intent.EXTRA_PHONE_NUMBER);

                if (lastNumber != null && lastNumber.equals(number) && isSkipNext) {
                    // do nothing, normal call
                } else {
                    if (lastNumber != null && lastNumber.equals(number) &&
(SystemClock.elapsedRealtime() - lastTime) < 3000) {
                        // deny call
                        setResultData(null);
                    } else {
                        lastTime = SystemClock.elapsedRealtime();
                        lastNumber = number;

                        setResultData(null);

                        Intent ctsIntent = new Intent(context, CallTypeSelector.class);
                        ctsIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                        ctsIntent.putExtra(Intent.EXTRA_PHONE_NUMBER, number);
                        context.startActivity(ctsIntent);
```

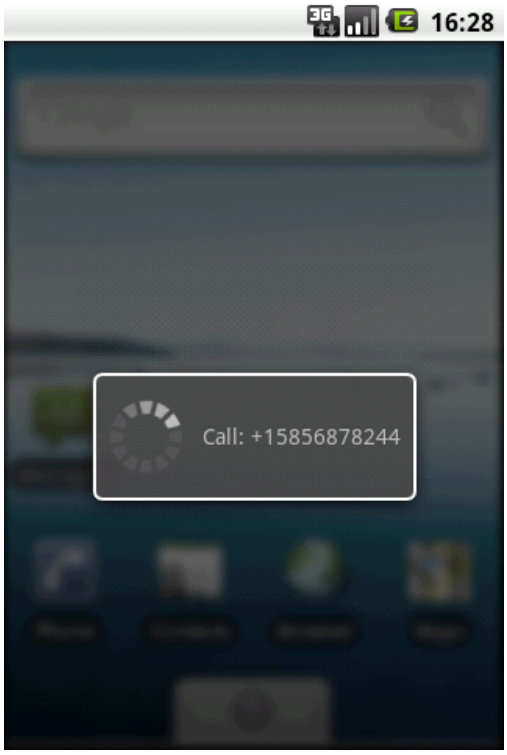
```
                }
            }

            isSkipNext = false;
        }
    }
  }
}
```

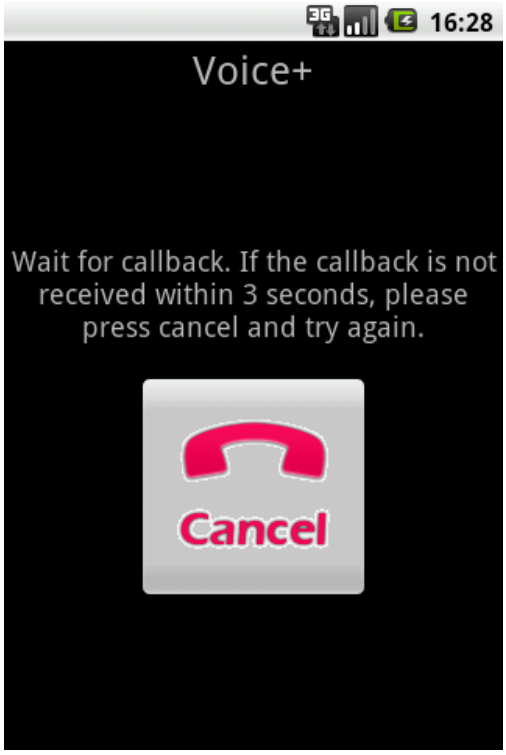
Az onReceive() függvényben dől el, hogy mit tegyünk, ha kimenő hívás történik. Ha be van kapcsolva a kimenő hívás integráció, akkor a hívási folyamat egy másik tevékenységen folytatódik, ahol a felhasználó eldöntheti milyen tipusú (Voice+ vagy GSM) hívást akar kezdeményezni. Az onReceive() függvényben van egy 3 másodperces újrahívás elleni védelem ugyanis hívás indításakor, ha gyorsan többször nyomjuk meg a hívás gombot, akkor az onReceive() függvény is többször fut le egymás után.

Egy hívási folyamat képei:









A hívás indításakor egy popup dialogus ablak ugrik fel, a második képen látható is. Az elmentett visszahívási számokat láthatjuk és még plusz 2 opciót a normál GSM hívásra:

- Home: telefon, mint eszköz fog felhívódni, USA SIM kártyával kell rendelkeznie.
- Google Talk: Google Talk kliensen történik a visszahívás.
- Gizmo5: virtuális USA-beli telefonszámon történik a visszahívás. Ehhez tartozik egy SIP account, ennek használatával kényelmesen internet telefonálhatunk.
- Phone: normál telefonhívást kezdeményez és változatlanul (bekapcsolva) marad a *Voice+* kimenő hívás integráció.
- Phone & disable Voice+: szintén normál telefonhívást kezdeményez, de kikapcsolja a *Voice+* kimenő hívás integrációt. Ezt akkor választja a felhasználó, ha huzamosabb ideig nem fog Google Voice szolgáltatáson keresztül telefonálni.

3.3.2.1. GoogleVoice osztály

Magyarországon csak a Google Talk vagy virtuális USA telefonszámon tudjuk igénybe venni a visszahívásos Google Voice szolgáltatást. Google Talk szolgáltatás minden Google account-hoz tartozik. Egy általam használt virtuális telefonszám szolgáltató, a Gizmo5 is a Google-nek egy ingyenes szolgáltatása, de nem jár automatikusan. A Gizmo5 a telefonszámon kívül még SIP (Session Initiation Protocol) hozzáférést biztosít minden regisztrálónak. A Google Voice HTTP szolgáltatására alapozva készítettem el a saját GoogleVoice osztályomat, mellyel könnyedén menedzselhető az account és híváskezelés.

Az alábbi HTTP API-ra alapozva készítettem el a GoogleVoice osztályt:

- Bejeletkezés: https://www.google.com/accounts/ClientLogin
- Alap URL: https://www.google.com/voice/m
- Hívás: https://www.google.com/voice/m/sendcall
- Hívás törlése: https://www.google.com/voice/m/callsms
- Tel.szám lista: https://www.google.com/voice/m/selectphone

Ehhez az API-hoz szükségesek a *HTTP header*-ek és paraméterek az információcsere biztosításához. Az alábbi elengedhetelen *header*-ek szükségesek:

- Authorization: bejelentkezés után ezzel tudjuk megadni az *Authentication Token*-t a további kommunikációhoz.
- User-agent: itt a következő fix paramétert adom meg: *„Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13 (KHTML, like Gecko) Chrome/0.A.B.C Safari/525.13”*

Az, hogy milyen paramétereket adunk még meg az URL-ben, az API függő.

3.3.2.2. Bejelentkezés

A GoogleVoice osztály konstruktorával lehet bejelentkezni, a Google felhasználónév és jelszó szükséges a példányosításhoz. A sikeres bejelentkezés eredménye egy *Authentication Token* amit letárolunk. Létrehoztam egy másik konstruktort is, aminek csak ez az *Authentication Token* a paramétere, ugyanis egy ilyen token élettartama akár 2 hét is lehet, így nem kell felesleges időt

tölteni azzal, hogy mindig bejelentkezünk minden példányosításkor. Ez jelentősen felgyorsítja a hívási folyamatot, a hívó gomb elengedése után szinte azonnal kapjuk a visszahívást.

A bejelentkezéshez szükséges HTTP header:

- User-Agent

A bejelentkezéshez szükséges HTTP paraméterek:

- Email: <Google account>
- Passwd: <hozzá tartozó jelszó>
- Service: grandcentral
- Source: VoicePlus

A HTTP válaszban pedig megkapjuk az *Authentication Token*-t, amivel a további API szolgáltatásokat igénybe vehetjük.

3.3.2.3. Visszahívás telefonszámainak lekérése

Szükséges HTTP header-ek:

- Authorization
- User-Agent

HTTP paraméterek nincsenek ehhez a szolgáltatáshoz, itt egy teljes telefonszám listát kapunk.

A HTTP válasz egyszerűbb darabolása kedvéért készítettem egy RegExp kifejezést, ami kiválogatja a szöveg közül a *Forwarding Number* nevét, tipusát és számát. A teljes visszahívás lista lekérése az alábbi függvénnyel történik:

```
public ForwardingNumber[] phones() {
    ForwardingNumber[] fnRes = null;

    try {
        String phonesString = getRequest(phonesURLString);
        List<ForwardingNumber> fn = new ArrayList<ForwardingNumber>();
        Pattern p = Pattern.compile("(name=\"phone\"
```

```
value=\")(.*?)(\\l)(.*?)(\")(.*?)(\\s*)(.*?)(:)");
        Matcher m = p.matcher(phonesString);
        while (m.find()) {
            fn.add(new ForwardingNumber(Integer.parseInt(m.group(4)), m.group(8),
m.group(2)));
        }

        fnRes = fn.toArray(new ForwardingNumber[0]);
        forwardingNumbers = fnRes;
    } catch (IOException e) {
    }

    return fnRes;
}
```

3.3.2.4. Híváskezelés

A call és cancelCall API szolgáltatással kezdeményezhetünk hívást, vagy annak visszavonását.

A hívás indításához és visszavonásához szükséges HTTP header-ek:

- Authorization
- User-Agent

A hívás indításához szükséges HTTP paraméterek:

- Phone: <visszahívandó telefonszám>
- Number: <a hívott telefonszáma>

A hívás visszavonásához szükséges HTTP paraméterek:

- Cancel: cancel

3.4. Tesztelés

A fejlesztés végső fázisa a tesztelés, ekkor az elkészült programot futás közben alaposan teszteljük. A program mobiltelefonon és emulátoron is tesztelésre került. Emulátornak az előnyei, hogy az Android összes megjelent verzióin tesztelhető az alkalmazásunk. A már kifutott szériáktól kezdve a legújabb csúcskategóriás készülékeken futó Android verziókon is tudunk tesztelni. Sőt még a kiadásra váró Android verziókon is folyhat a tesztelés, amelyek még valódi eszközön meg sem jelentek a piacon. Az emulátor nem ad valósidejű futási időt, de a viselkedése kisebb megszorításokkal szinte megegyezik egy igazi eszközzel, fejlesztés közben igen nagy segítség. Az emulátor előnye továbbá, hogy amikor teljesen elölről kezdjük a tesztelést, nyugodtan nyomhatunk gyári beállítás visszaállítását a teljes felhasználói adat törléséhez, egy valódi eszközön nem szívesen hajtanánk végre ugyanezt mivel akkor a teljes személyes adataink is törlődnek a készülékről. Viszont ha emulátoron tökéletesen fut az alkalmazásunk, valódi eszközökön is teszteljük és lehetőleg a legtöbb fajtán. Egyre több mindenkinek van Android platformmal ellátott készüléke, családtagok, kollégák, barátok és ismerősök körében akad manapság néhány készülék, amin tudunk egy-egy végső tesztet futtatni.

3.5. A Voice+ publikálása az Android Marketbe

Mielőtt publikáljuk az alkalmazást a marketbe, előtte az alábbi teendőnket kell elvégezni:

- Készíteni kell egy fejlesztői profilt az Android Market-en, azaz meg kell adni a fejlesztő nevét, email címét, a fejlesztő honlapjának címét és telefonszámát. Ezeket a kontaktinformációkat fogja látni a vásárló/letöltő.
- 25 USD regisztrációs díjat kell fizetni a fejlesztői profil létrehozásához. Ezzel a regisztrációs díjjal jogosult a fejlesztő bármennyi program feltöltésére a marketbe.

Ha kész a fejlesztői profil, megkezdhetjük a feltöltést. Meg kell adnunk azokat az az állományokat, amiket fel szeretnénk tölteni:

- A futtatható állományt: VoicePlus.apk.
- 2 vagy több screenshot-ot a programról melyeknek a felbontásai a következők lehetnek: 320x480, 480x800, vagy 480x854 pixel méretű 24 bites PNG vagy JPEG. Az, hogy függőleges vagy vízszintes a tájolása a képnek az mindegy.
- Nagyfelbontású programikon, aminek a mérete 512x512 pixel méretű 32 bites PNG vagy JPEG.
- Egy promóciós ikont, aminek a mérete 180x120 pixel méretű 24 bites PNG vagy JPEG.
- Egy másik nagyfelbontású kép mely a program sajátosságait tartamazza, aminek a mérete 1024x500 pixel méretű 24 bites PNG vagy JPEG. Ilyen méretű képekből kerül ki néhány a Market főoldalára.

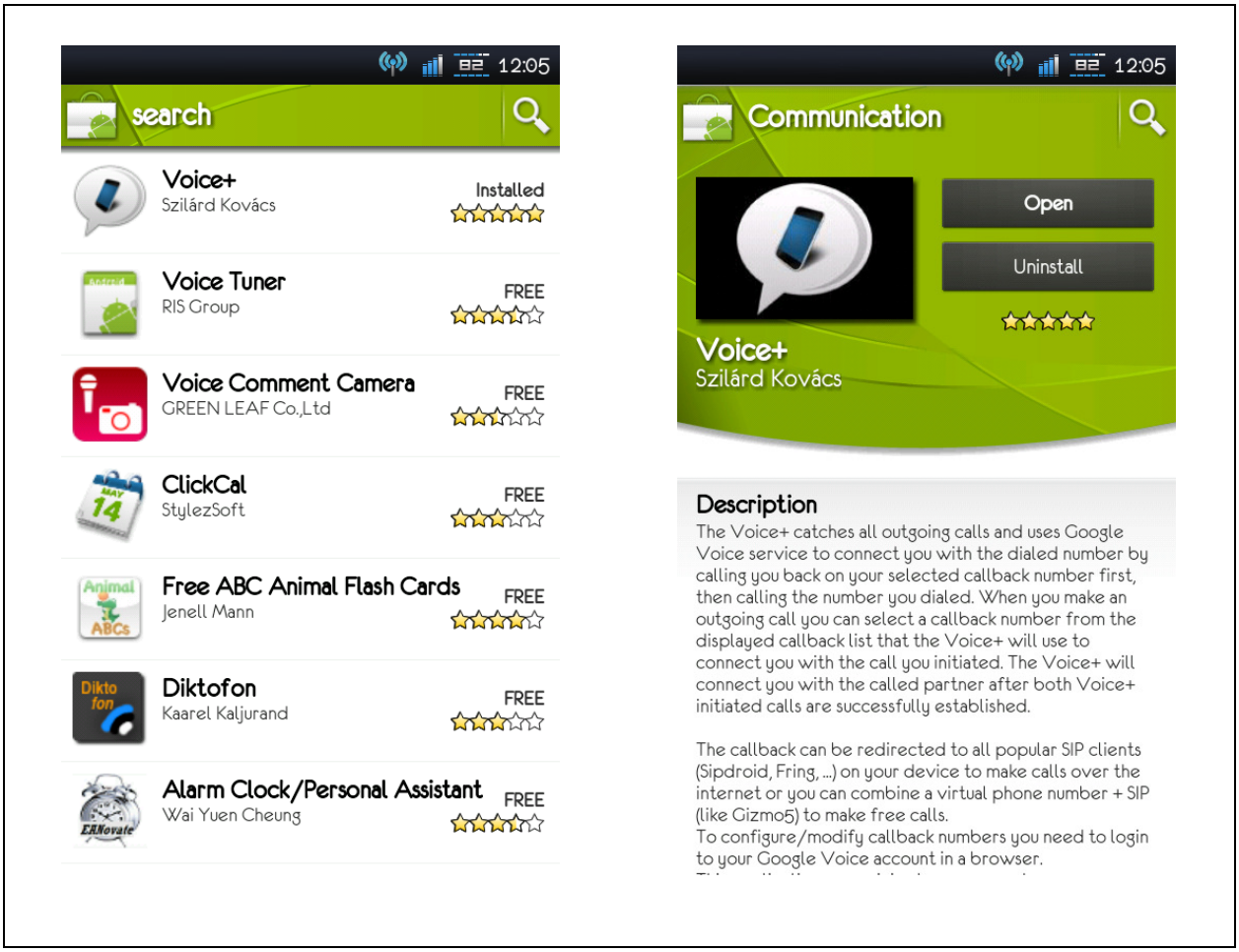
Némi leírást is meg kell adni a programhoz, alapból az angol nyelvű program megnevezést és leírást ajánlja fel. A program nevéhez beírtam a *Voice+* megnevezést és a következő leírást adtam meg a programhoz:

> The Voice+ catches all outgoing calls and uses Google Voice service to connect you with the dialed number by calling you back on your selected callback number first, then calling the number you dialed. When you make an outgoing call you can select a callback number from the displayed callback list that the Voice+ will use to connect you with the call you initiated. The Voice+ will connect you with the called partner after both Voice+ initiated calls are successfully established.

The callback can be redirected to all popular SIP clients (Sipdroid, Fring, ...) on your device to make calls over the internet or you can combine a virtual phone number + SIP (like Gizmo5) to make free calls.
To configure/modify callback numbers you need to login to your Google Voice account in a browser.
This application uses minimal memory and resources, no voicemail/SMS/alert/notification support.

Még néhány adatot kell megadnunk arra vonatkozólag, hogy milyen korosztálynak kerül kipublikálásra a programunk, mely területen tölthetik le a programunkat, és kontaktinformációt módosíthatjuk, ha nem ugyanazt szeretnénk megjeleníteni, mint amit a fejlesztői profilban megadtunk.

Ezután nincs más dolgunk, mint megnyomni a *Publish* gombot és azonnal kikerül az alkalmazás az Android Market-be, ettől kezdve bárki számára elérhető a http://market.android.com webcímen vagy egy Androidos készüléken közvetlenül:


12:05
search
Voice+
Szilárd Kovács
Installed
Voice Tuner
RIS Group
FREE
Voice Comment Camera
GREEN LEAF Co.,Ltd
FREE
ClickCal
StylezSoft
FREE
Free ABC Animal Flash Cards
Jenell Mann
FREE
Diktofon
Kaarel Kaljurand
FREE
Alarm Clock/Personal Assistant
Wai Yuen Cheung
FREE
12:05
Communication
Open
Uninstall
Voice+
Szilárd Kovács
Description
The Voice+ catches all outgoing calls and uses Google Voice service to connect you with the dialed number by calling you back on your selected callback number first, then calling the number you dialed. When you make an outgoing call you can select a callback number from the displayed callback list that the Voice+ will use to connect you with the call you initiated. The Voice+ will connect you with the called partner after both Voice+ initiated calls are successfully established.
The callback can be redirected to all popular SIP clients (Sipdroid, Fring, ...) on your device to make calls over the internet or you can combine a virtual phone number + SIP (like Gizmo5) to make free calls.
To configure/modify callback numbers you need to login to your Google Voice account in a browser.

4. Összefoglalás

A szakdolgozatom elkészítésével sikerült betekintést nyerni az Android platformra való fejlesztés lépéseibe. Első lépésben az Android platform került ismertetésre, mely a bemutatott alkalmazás implementálása mellett számos új lehetőségeket rejt. Mára már kiforrott mind a platform és mind a fejlesztői környezet, mellyel nagyon könnyen fejleszthetünk alkalmazásokat.

Továbbá ismertettem a Voice+ alkalmazás megvalósításának lépéseit. Olyan alkalmazást valósítottam meg, mellyel a Google Voice felhasználók ezután mobillal is bárhonnan igénybe tudják venni ezt a szolgáltatást, interneten keresztül telefonálhatnak, ahol van 3G lefedettség vagy WiFi elérhetőség.

Mivel az Android nyílt forráskódú, ezáltal szinte azonnal szabadon bővíthető a legújabb technológiák felhasználásával. Az Android platform fejlődése tovább folytatódik azáltal, hogy a fejlesztői közösségek együtt dolgoznak az innovatív mobil alkalmazások létrehozásán.

5. Köszönetnyilvánítás

Szeretnék köszönetet mondani mindenkinek, aki közvetlen vagy közvetett módon segítette a dolgozat elkészülését.

Köszönettel tartozom Bátfai Norbert témavezetőmnek, hogy lehetőséget biztosított munkám sikeres elvégzéséhez és dolgozatom megírásához. Köszönöm segítőkész támogatását és dolgozatom alapos átnézését.

Köszönöm a külső konzulensemnek Szabó Ákosnak a hasznos tanácsait és a Micont Kft eszközeit a dolgozatom elkészítéséhez.

Végül szeretném megköszönni a családom türelmét és segítségét.

6. Irodalomjegyzék

- Open Handset Alliance
  http://www.openhandsetalliance.com
- Android
  http://www.android.com
- Android Open Source Project
  http://source.android.com
- Publising on Android Market
  http://market.android.com/publish
- Google Voice
  https://www.google.com/voice
- Google GrandCentral
  http://www.google.com/grandcentral
- Gizmo5
  http://www.google.com/gizmo5
- Gartner IT research and advisory company
  http://www.gartner.com
- Unwiredview mobile technology blog
  http://www.unwiredview.com
- Canalys: expert analysis for the high-tech industry
  http://www.canalys.com/