Debreceni Egyetem

Informatikai Kar

Alkalmazásfejlesztés J2ME platformon

Konzulens:	Külső konzulens:	Készítette:
Dr. Adamkó Attila	Dr Laborczi Péter	Parisek Zsolt
Egyetemi tanársegéd	senior kutató	Programtervező informatikus MSC

Debrecen

2009

BEVEZETŐ 4

A JAVA 2 MICRO EDITION, JAVA ME 6

A Java Platform 6

Java ME 8
A Java ME architektúrája 8
Konfigurációk: 10
Profilok. 11

A CLDC 12
A CLDC-vel szemben támasztott követelmények: 12
A CLDC hatásköre 12
Kompatibilitás a Java nyelvvel: 13
Kompatibilitás a Java Virtuális Gép specifikációval: 13
Biztonság: 13
CLDC osztályok 15
A K virtuális gép (KVM) 17
A virtuális gép indítása és a JAM 17
Natív kód: 17
Java Code Compact (ROMizer) 18

A MIDP 19
A MIDP csomagok 20
A core csomagok: 20
A Java Me által bevezetett csomagok: 20

GLOBAL POSITIONING SYSTEM 27

A helymeghatározási módszer 28
Az eljárás lépései 28

Az NMEA 0183 szabvány 30
$GPRMC 30

ÖSSZEFOGLALÁS 30

MCHUNTER ALKALMAZÁS FEJLESZTÉSE 31

Az alkalmazás bemutatása 32
Az alkalmazás működése 32
Részletes bemutatás: 33
Az éttermek POI adatainak beszerzése 33
A letöltött POI adatok feldolgozása 34
A valós földrajzi koordináták megszerzése 37
A környezetet ábrázoló térképrészlet beszerzése 41
A „közeli” éttermek és a térképrészlet megjelenítése 43
A „főprogram” 48

ÖSSZEGZÉS... 51

Bevezető

Globális Helymeghatározó Rendszer (Global Positioning System, GPS) pozicionálás pontosságát szándékosan csökkentő, zavaró jel 2000. május 1-i kikapcsolása hatalmas lökést adott a GPS polgári célú felhasználásához. Egyre több olyan alkalmazás született, amely valamilyen formában kiaknázza a felhasználó térbeli koordinátáit, mint például a navigációs rendszerek. A GPS népszerűségének növekedése egy addig ismeretlen fogalmát vezetett be a helyfüggő szolgáltatás fogalmát, mely napjaink egyik legfelkapottabb témájának számít.

Az elmúlt félév során egy helyfüggő szolgáltatás megvalósításában vettem részt. Munkahelyemen, a Bay Zoltán Alkalmazott Kutatási Közalapítvány Ipari Kommunikációs Technológiák Innovációs Intézetében (Bay-IKTI), az „Ambiens intelligencia[1] alkalmazása a közúti közlekedésben" témakörével foglalkozom. A Bay IKTI egyik kiemelt projektének célkitűzése az, hogy a navigációs rendszerek számára értelmezhető utazási információkat állítson elő. Ez egy komplex feladat, magába foglalja az információ összegyűjtését, feldolgozását, illetve újbóli szétosztását. Az általunk létrehozott rendszer egy úgynevezett FCD (Floating Car Data) rendszer, melynek általános működése a következő:

Egy FCD rendszerben a hagyományos statikus adatgyűjtők (közutakon elhelyezett forgalomszámláló kamerák, hurokdetektorok) helyett az utakon mozgó egységek biztosítják a mindig friss közlekedési információkat. Nyomvonaluk folyamatos analizálásával az egyes útszakaszok aktuális áthaladási ideje kiszámítható. Egy ilyen FCD rendszer működéséhez viszonylag sok, lehetőleg egyenletesen eloszló mozgó szenzor szükséges.

Az FCD rendszerünknek adatot szolgáltató flotta mérete viszont jelenleg még nem éri el az optimális működéshez szükségest, ezért az általunk szolgáltatott forgalmi terheltség nem minden útra pontos, ha egyáltalán létezik. Több megoldás is született eme probléma megoldására.
Az első triviálisan adódó válasz az, hogy növeljük az adatszolgáltató flotta méretét. Ennek megvalósítása viszont már koránt sem triviális. A flottamenedzser cégek általában elzárkóznak az általuk menedzselt flották adatainak kiszolgáltatásától, ha pedig nem, akkor annak értékét túlontúl felértékelik és gazdasági okok miatt nem jön létre az együttműködés. A mi megoldásunk egy közösségi elven működő rendszer megvalósítása, melyben az adatszolgáltatók maguk a rendszert használó járművek.
Egy másik lehetséges út, ha a különböző útszakaszokról rendelkezésünkre álló, FCD és egyéb (pl. időjárási adatok, rendezvények) adatokat adatbányászati eljárások alá vetjük. Így lehetőség adódik feltárni a különböző utak forgalmi terheltségei közti mélyebb, nem triviális összefüggéseket. Az így nyert extra tudást a rendszerbe integrálva minőségi javulást érhetünk el.

---

[1] Az *ambiens* intelligencia (AmI) több tudományág, többek között a távközlés, a számítástechnika és a szenzorika új interdiszciplináris paradigmája. A koncepció lényege, hogy a felhasználókat olyan környezetbe ágyazott, feltűnésmentes számítási és infokommunikációs technológiákkal vegyük körül, melyekben a hangsúly a személyi számítógépekről egyre inkább a felhasználóbarát, hatékony és elosztott szolgáltatások hálózata felé tolódik el.

A dolgozat témáját az adatszolgáltató flotta méretét növelő megoldás megvalósítása ihlette. Az általunk készített rendszer egy szabványos kliens-szerver architektúrán alapszik, ahol a kliensek a mozgó járművekben elhelyezett PDA-k vagy okostelefonok. A kliensek rendelkeznek valamilyen GPS forrással, és az általuk gyűjtött adatokat egy szabványos http protokollon keresztül elküldik egy szervernek. A szerver ezt későbbi feldolgozás céljából eltárolja, és az adott kliens környezetére vonatkozó közlekedési információkat visszaküldi, amelyet a kliens megjeleníteni. A kliensalkalmazásunkat *MyTraffic*-nak neveztük el, mely szabadon letölthető a Bay-IKTI honlapjáról.
A MyTraffic tervezésekor elsődleges cél volt, hogy többféle platformon is futtatható legyen, illetve, hogy a felhasználónak ne kelljen semmilyen licenszdíjat fizetnie egyetlen rendszerkomponensért sem. Mindezen kritériumoknak eleget tevőnek tűnt a Java technológia, ezért a *Mytraffic* megvalósításához az ebben a környezetben alkalmazható Java Micro Edition-t választottuk.
Dolgozatomban a fejlesztés során szerzett tapasztalataimat osztom meg, bemutatni a Java ME előnyeit, illetve korlátait. A dolgozat első részében magáról a Java ME-ről írok, míg a második felébe a technológia gyakorlati alkalmazását szemléltetem egy példaprogramon keresztül.
Az alkalmazásom az *McHunter* elnevezést kapta, mivel a környezetünkben lévő McDonald's éttermeket jeleníti meg egy térképen. Az alkalmazás feltételezi, hogy a futtató hardver rendelkezik integrált GPS vevővel és képes http protokollon keresztül kommunikációt folytatni egy szerverrel.
Indításkor az alkalmazás letölti az éttermek POI[2] adatait, majd kiolvassa a GPS vevőből az aktuális koordinátákat. A koordináták birtokában már képes a saját környezetének megfelelő térképrészletet letölteni, majd azt megjeleníteni a környezetében lévő (ha van ilyen) McDonald's éttermekkel együtt. A térképet a Bay-IKTI által üzemeltetett térképszerverről töltöm le, de alkalmazhattam volna akár a Google Maps szolgáltatást is. Ez a különbség technikai szempontból nem érdekes, csupán csak egy URI-ben térnek el egymástól.
Az itt bemutatott alkalmazás, főként demonstrációs célzatú, az iparban is helytálló üzleti modellt nehezen lehet mögé elképzelni. Viszont jó alapot biztosít ahhoz, hogy az olvasóval megismertessem egy helyfüggő szolgáltatás megvalósításakor felmerülő problémákat.
Dolgozatom feltételezi, hogy az olvasó rendelkezik a megfelelő informatikai / hálózati alapismeretekkel. Megfelelő információ-háttér hiány esetében javaslom az irodalomjegyzékben lévő anyagok átolvasását.

[2] **POI (Points Of Interest) – hasznos helyek, érdekes pontok**: Különböző helyzetmeghatározó programok által használt kifejezés, mely a számunkra (vagy mások számára) fontos helyek, pontok jelölésére szolgál.

A Java 2 Micro Edition, Java ME

A Java Platform

A Java nyelvet a Sun Microsystems cég fejlesztette ki a 90-es évek elején. A cél egy olyan eszköz elkészítése volt, mely segítségével nem kell minden operációs rendszerhez külön-külön megírni a programot, hanem elég egyszer. Jelszavuk: „Write once, run everywhere” (írd meg egyszer, futtasd mindenhol). Céljuk eléréséhez egy új eszközt kellet kifejleszteniük, melynek neve: Java Platform.
A Java Platform egy olyan környezet, ami képes a Java programozási nyelven írt programok futtatására. Itt a platform szó nem egy konkrét hardvert vagy operációs rendszert takar, hanem egy virtuális gépnek nevezett futtató programot, és szabványos osztálykönyvtárak gyűjteményét.
Ezek után egy Java nyelven írt programot nem bináris kóddá fordítunk le, hanem úgynevezett bájtkódra. Ez a bájtkód önmagába nem futtatható, hanem egy értelmezőt igényel: a Java virtuális gépet (Java Virtual Machine, JVM), mely képes azt lefordítani az adott gép natív utasításkészletére. Ez a módszer lehetővé teszi, hogy különböző platformon ugyanazon kód futtatható legyen.
Kezdetben a bájtkód bináris kóddá fordítása futási időben történt , amely lényegesen lassabb program futási időt eredményezett a bináris kódú programokéhoz képest. Ennek javítása céljából a SUN kidolgozta a just-in-time (JIT) fordítót, melyet az 1.2-es verzió óta az összes SUN által kibocsájtott JVM verzió tartalmaz. A JIT alapötlete, hogy a JVM ne egyenként értelmezze a bájtkódú utasításokat futási időben, hanem egyszer a bájtkód betöltésekor fordítsa azt le az adott gép natív utasításaira. Így a bináris kódú programok futási idejét lehet elérni, de a program betöltése tovább tart. Napjainkban a Java Platform hivatalos neve **Java 2 Platform**, azonban a 2-es számot gyakran elhagyják.

Napjainkra a következő Java Platform kiadások fejlődtek ki: Standard Edition (Java SE), Enterprise Edition (Java EE) és Micro Edition (Java ME).

1. ábra, Java Platformok

Java Platform, Standard Edition vagy **Java SE:** a leggyakrabban használt Java platform, melyet általános célú (asztali vagy szerver) alkalmazáshoz fejlesztéséhez használhatunk . A Java SE tartalmazza a virtuális gépet („írd meg egyszer, futtasd mindenhol"), és tartalmaz az osztálygyűjtemények halmazát csomagok formájában. Ezen csomagokba van beágyazva a Java nyelv eszközrendszere. Ilyenek például a fájlrendszer, a hálózat és a grafikus interfészek kezeléshez szükséges osztályok, a matematikai számításokat megvalósító osztályok, a CORBA, az RMI, illetve a Java nyelv kezdeti sikereit megalapozó Applet technológiát megvalósító osztályok. A Java SE alapjául szolgál mind a Java EE, mind a Java ME kiadásokhoz.

Java Platform, Enterprise Edition vagy **Java EE:** egyszerűsíti a vállalati alkalmazások fejlesztését, szabványos moduláris komponenseket támogat szolgáltatások halmazán keresztül. Az alkalmazások működésének/viselkedésének számos részletét szabályozza automatikusan - összetett programozás nélkül.

A Java EE sok előnyét átveszi a Java SE-nek, úgy mint a „írd meg egyszer, futtasd mindenhol", JDBC API adatbázisok eléréséhez, CORBA technológia létezető vállalati erőforrásokkal való interakciókhoz, és biztonsági modell az adatok védelméhez internetes alkalmazások esetén.

Ezekre az alapokra építve a Java EE teljes támogatást ad Enterprise JavaBean komponensekhez. Új lehetőségként jelenik meg a Java Servlets Api, a JavaServer Pages és az XML technológia. (Bár az újabb J2SE változatok is tartalmaznak már alapvető XML kezeléséhez szükséges Java csomagokat.)

A Java EE szabvány tartalmaz komplett előírásokat, teszteket arra, hogy biztosítsa az alkalmazások hordozhatóságát azon vállalati rendszerek között, amelyek támogatják a Java EE platformot. Továbbá a Java EE specifikáció Web szolgáltatásokkal való (illetve azok közötti) kapcsolódást is biztosít a WS-I Basic Profile támogatásán keresztül.

Mivel a dolgozat témája a Java ME, ezért terjedelmi okok miatt a Java SE és Java EE részletesebb bemutatására nem térek ki. A téma után mélyebben érdeklődő számára ajánlom a http://java.sun.com/javase/ és a http://java.sun.com/javaee/ webhelyek felkeresését.

Java ME

Mit is takar a J2ME vagy más néven Java ME mozaik szó?
A válasz a SUN honlapján[3] a következőképen hangzik:

***A Java Platform, Micro Edition** vagy **Java ME** egy robosztus, flexibilis környezetet biztosít mobil és egyéb beágyazott eszközökön – mobil telefonok, PDA-k, beltéri egységek és nyomtatók – Java nyelven írt alkalmazások futtatására. A Java ME flexibilis felhasználói felületet, robosztus biztonsági eszközkészletet és beépített hálózati protokollokat tartalmaz, hálózati és egyedi alkalmazásokat fejlesztésére is egyaránt alkalmas. A Java ME a hordozható kódot támogató, a különféle eszközök natív képességét kiaknázó futtatási környezetet biztosít az alkalmazások számára.*

Ez a válasz a Java ME-t kevésbé ismerők számára túlontúl felületes, kissé marketing hatású, a hozzáértőbbek pedig kissé megmosolyogják. Próbáljuk ezt kissé jobban megfogelmezni .

A Java ME architektúrája

Mint az a SUN válaszából is kiderült, ez a platform a szűkös erőforrásokkal (memória, processzor, stb.) bíró eszközök számára lett kifejlesztve. Mivel viszont ez ilyen eszközök rendeltetése egymástól nagyban eltérő lehet (a nyomtatónkon nem próbálunk telefonálni), ezért e sokszínűség figyelembevételével kellet a platformot kialakítani. Ezen cél eléréséhez a Java ME architektúráját rétegekre bontották.

A Java ME architektúráját a következő 3 réteg alkotja:

- *Java Virtual Machine:* Ez a réteg implementálja a Java virtuális gépet, melynek funkciója a jól ismert *„írd meg egyszer, futtasd mindenhol”* elv teljesítése. A virtuális gép eltakarja az operációs rendszerek közötti különbségeket, és támogatást nyújt az adott konfiguráció számára.

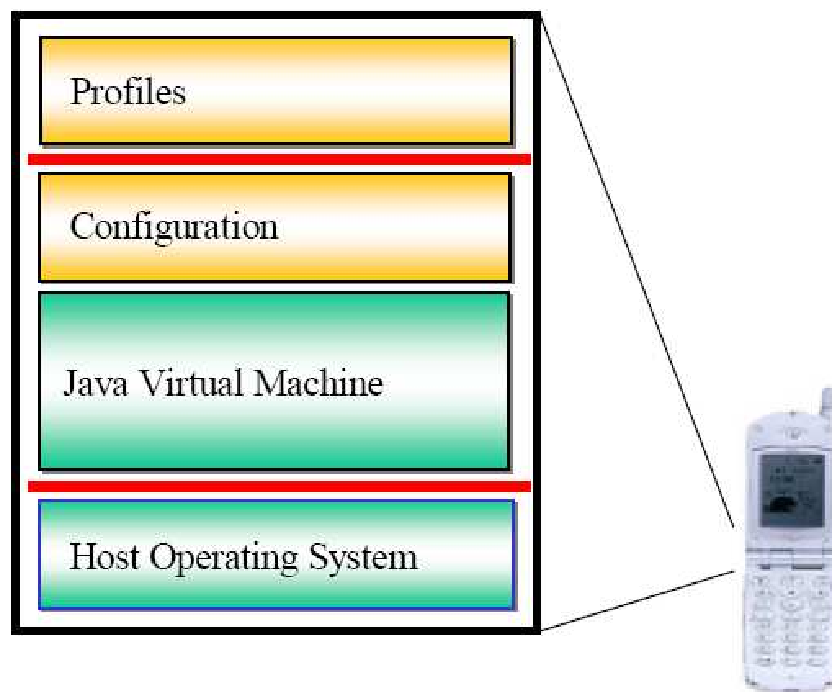
- *Configuration*: a konfiguráció kevésbé látható a felhasználók számára, de nagyon fontos a profilok implementálóinak. Lehetségessé teszi a hardverek kategorizálását, a piaci szegmensen egy „horizontális” csoportosítást képezve. Meghatározza a „legkisebb közös nevezőt” azaz a Java virtuális gép tulajdonságainak és a Java osztálykönyvtáraknak egy olyan minimumát definiálja, amelyek az egy kategóriába eső eszközökön mindenképpen rendelkezésre állnak.

- *Profile:* A Java ME architektúrájának legláthatóbb rétege a felhasználók és az alkalmazásfejlesztők számára. Családokba sorolja az eszközöket, a piaci szegmensen egyfajta „vertikális” csoportosítást hoz létre. A profilok az adott konfigurációra szabva vannak implementálva, feladatuk a programfejlesztés során rendelkezésre álló alkalmazás fejélesztési interfészek (Application Programming Interfaces, API)

[3] http://java.sun.com/javame/index.jsp

halmazának definiálása. Egy mobil alkalmazás fejlesztése során a legelső lépés a használni kívánt profil megadása, ezáltal biztosítja, hogy az újonnan létrejött alkalmazás futtathatóvá váljék mindazon hardvereken amelyek az adott profilt támogatják. Általában egy hardver többféle profilt is támogat.

Ez a három réteg ráépül a „gazda gép" operációs rendszerére, melyet a 2. ábra illusztrál.



2. ábra, **Java ME architektúráját**

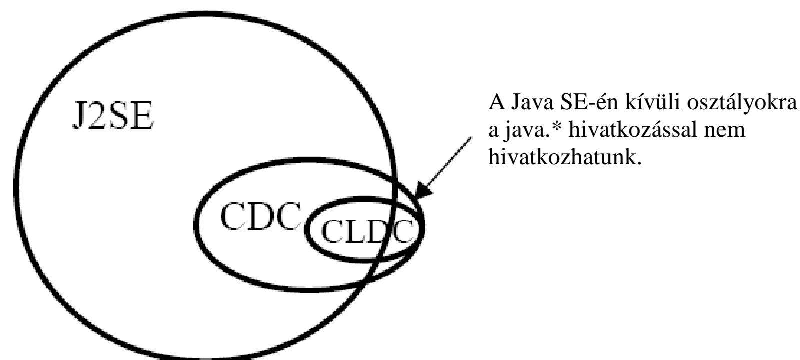
A virtuális gép implementációja és a konfigurációk specifikációja szorosan összekötődik. A hardverek alapvető képességei alapján ez a két réteg együt határozzák valósítják meg a kategorizálást. A különböző kategóriákon belül további megkülönböztetést (hardvercsaládokba sorolást) a profilok segítségével tehetünk. Ha egy alkalmazás igényli, akkor a profilok további Java osztályokkal bővíthetőek.
Ha egy eszközgyártó kifejleszt egy új eszközt, akkor azt vagy besorolja valamely már meglévő kategóriákba és azon belül valamelyik családba, vagy ha egyikbe sem illik bele, akkor bővíti a profilokat, konfigurációkat és virtuális gép technológiákat. Ezek az újítások a *Java Community Process (JCP)* formájában JSR-eken(Java Specification Requests) keresztül kerülnek bevezetésre. Egy JSR tartalmazza az aktuális leírását a tervezetnek és a végleges specifikációját a Java platformnak.

Konfigurációk:

A JCP jelenleg két konfigurációt definiál

Az egyik konfiguráció a *Connected Limited Device Configuration (CLDC JSR 30, JSR 139)*, melyet a kicsi, szűkös erőforrással rendelkező eszközök számára fejlesztettek ki. Ezek az eszközök tipikusan 192 kB - 512 kB szabad memóriával és 16-32 bites processzorral rendelkeznek. Input-Output eszközrendszerük szegényes, és hálózati kommunikációjuk általában nem támogatja a TCP/IP-t. CLDC egy sajátos virtuális gépet, úgynevezett *K Virtual Machine (KVM)* futtat. CLDC kategóriába sorolható szinte az összes mobiltelefon (bár napjainkban egy két csúcsmodell már CDC futtatására is alkalmas).

A másik konfiguráció a *Connected Device Configuration (CDC).* A CDC konfiguráció kategóriájába tartozó eszközök már jóval több erőforrással rendelkeznek mint a CLDC kategóriájú eszközök, ezért a Java SE platform sokkal több eszközét implementálja. A CDC a klasszikus virtuális gépet használja, amely tartalmaz minden olyan virtuális gép funkciót, amelyek egy asztali gépen futó Java Platformon fellelhetők. CDC kategóriájú eszközökre általában jellemző, hogy legalább néhány megabájt memória mindig rendelkezésre áll, általában 32 bites processzoruk van, fejlettebb az input-output eszközrendszerük, és a hálózati kommunikációjuk általában támogatja a TCP/IP -t. CDC kategóriájú eszköz például egy IP TV beltéri egység, vagy a gépjárművek beágyazott vezérlői.



3. ábra, A Java konfigurációk viszonya

Profilok.

A profilok a konfigurációkon belüli további csoportosítást teszik lehetővé. Erre azért van szükség, mert az egy kategóriába sorolt eszközök közötti különbség még mindig jelentős lehet és a családokba sorolás nélkül nem állna rendelkezésre egy alkalmazás futásához szükséges komplett környezet és sérülne az „írd meg és futtasd mindenhól" szabály. A magasabb szintű API-k (amelyek a profilokat adják) pontos meghatározásával definiálni tudjuk az alkalmazásunk életciklus-modelljét, felhasználói interfészét. Az alkalmazások fejlesztésénél kívánatos cél, ha legalább az egy családba tartozó eszközökön ugyanaz a kód futhasson (sajnos ez a mai napig nem sikerült teljesíteni teljesen). Napjainkra két CLDC kategóriájú és három CDC kategóriájú profil alakult ki melyek az alábbiak:

Information Module Profile (IMP) JSR 195

Ez egy CLDC kategóriájú profil, melyet azon hardverek számára fejlesztették ki, amelyek nem, vagy nagyon szegényes grafikus megjelenítési eszközkészletet tartalmaznak, viszont rendelkeznek hálózati kommunikációval. Ilyen eszközök például a routerek, hálózati kártyák.

Mobile Information Device Profile (MIDP); JSR 37, JSR 118:

A másik CLDC-beli profilt mobiltelefonok és PDA-k számára tervezték. Tartalmazza a hálózati kommunikációs, grafikus megjelenítési és tárolási eszközrendszert. Napjaink legnépszerűbb profilja.

Foundation Profile (FP); JSR 46, JSR 219:

A CDC legalacsonyabb szintű profilja. A J2SE API-kon alapszik, viszont nem rendelkezik grafikus interfésszel, ezért fő alkalmazási területe a hálózati nyomtatók, routerek és gateway-ek.

Personal Profile(PP); JSR 216:

Ezt a profilt is CDC kategóriájú eszközökre tervezték, a J2SE API-k halmazából áll, implementálja a teljes AWT-t (JAVA Abstarct Window Toolkit), és támogatja a webszolgáltatásokat. Fő alkalmazási területe a gazdagabb erőforrásokkal rendelkező PDA-k, illetve a beágyazott webböngészők.

PBP (Personal Basic Profile); JSR 217:

Környezetet biztosít az olyan hálózatra kapcsolható CDC-beli eszközöknek, amelyeknek csak az alap grafikus szintet igénylik. Az FP-t a következő 3 tulajdonsággal bővíti:

- Xlet alkalmazási modell, adaptálva a Java TV API-kat,
- J2SE AWT könnyűsúlyú komponensei,
- Inter-Xlet kommunikáció(IXT), a Remote Method Invocation (RMI) API egy részhalmazának felhasználásával

Alkalmazási területe jármű telematikai rendszerek, TV set-top box.

A CLDC

Mivel a dolgozat témájaként szolgáló alkalmazás főként mobiltelefonos környezetben használható, ezért az ezen eszközökre vonatkozó konfigurációt mutatom be részletesebben.

A mobiltelefonok konfigurációja a CLDC. Ezen konfiguráció célhardvereiről általánosságban elmondható, hogy:

- alacsony energiafogyasztású hardverek, melyek gyakran elemről működnek
- 16 vagy 32 bites processzorral rendelkeznek melynek órajele legalább 16MHz
- memória 192KB – 512kB nagyságú
- sokféle kommunikációs keppességgel rendelkeznek, de a kapcsolattal minősége változó és sávszélessége korlátozott.

A paraméterekből kiderül, hogy a CLDC célhardverei kicsi szűk erőforrással rendelkező eszközök, melyek speciális feltételeket szabnak a platform implementálói számára. A CLDC technológiát a KVM (K Virtual Machine) és az osztálykönyvtárak magja alkotja. A memória mérete erősen befolyásolta a platform kialakítását. A Java Platfor minimum 192KB memóriát igényel, melyből 160KB statikus memóriának kell lennie a CLDC osztályok és a KVM számára. A specifikáció feltételezi továbbá, hogy egy alkalmazás 32Kbyte Java Heap (halom) területen is képes futni. Ezen kritériumok erősen átformálta a virtuális gépet, erre utal az új elnevezése a KVM is.

A CLDC-vel szemben támasztott követelmények:

Egy CLDC alkalmazásnak sokféle kicsi eszközön kell futni a wireless kommunikációval rendelkező eszköztől kezdve a celluláris telefonokon át az eladó terminálok, és otthon alkalmazásokig. Garantálnia kell a hordozhatóságot és meg kell határoznia az alkalmazható Java technológiák egy olyan minimális halmazát, amelyek alkalmazhatóak ilyen sokszínűség mellet is.

A CLDC hatásköre

A CLDC konfiguráció a következő területeket szabályozza:

- kompatibilitás a Java nyelvvel és a Java Virtuális Gép specifikációval
- biztonság
- Java könyvtárak magja (java.lang.*, java.util.*)
- input / output
- hálózatba szervezés

Hatásköre nem terjed ki a következő területekre, ezeknek a kezelése a profilok feladata:

- alkalmazások életciklus modellje (telepítés, betöltés, törlés)
- kapcsolattartás a felhasználóval
- eseménykezelés

Kompatibilitás a Java nyelvvel:

Az általános cél, hogy egy CLDC a célhardverek által szabott feltételek megtartása mellet kompatibilis legyen az 1996.-ban James Gosling, Bill Joy, és Guy L.Steele. Addison-Wesley által specifikált Java nyelvel. Ezt a következők kivételével megvalósították:
A CLDC nem támogatja

- a lebegőpontos adattípusokat
- az objektumok finalizálását: az `Object.finalize()` metódus nem létezik a CLDC-ben
- limitálja a kivételkezelést is: a `java.lang.Error` legtöbb alosztályát nem támogatja

Kompatibilitás a Java Virtuális Gép specifikációval:

A cél itt is az, hogy a hardverek által szabott feltételek megtartása mellet egy CLDC virtuális gép a lehető legjobban kompatibilis legyen aTim Lindholm és Frank Yellin Addison-Wesley által 1996-ban kiadott Virtual Machine Specification specifikációval.
A következő kivételek mellet ez is sikerült:
Egy CLDC-beli virtuális gép

- nem támogatja a lebegőpontos adattípusokat.
- nem támogatja a JNI-t (Java Native Interface).
- nincs felhasználó-definiált JAVA osztálybetöltő.
- nincs reflection támogatás *(komponensek felderítése).*
- nem támogatja a szálcsoportokat és a démon szálakat.
- korlátozott a kivételkezelés.

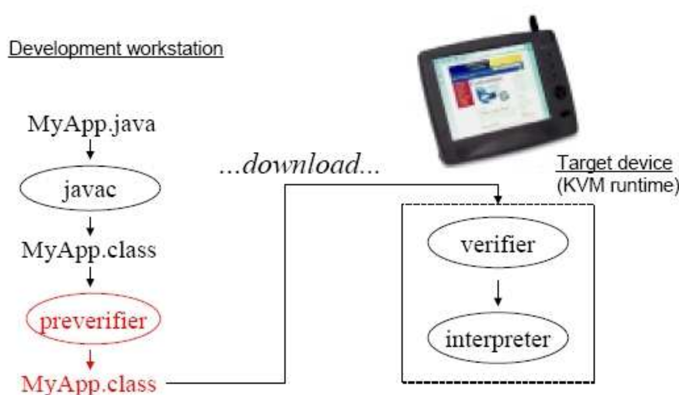
Biztonság:

A CLDC specifikáció a következő biztonságot növelő eszközkészlettel rendelkezik:

- Alacsony szintű virtuális gépbeli biztonság van megvalósítva azáltal, hogy az igényelt Java osztályok egy verifikációs lépésen kell keresztül esniük.
- Az alkalmazások egy úgynevezett „sandbox” -ban vannak futtatva, ezáltal minden más folyamattól védve vannak.
- Az osztályok rendszer csomagokba vannak burkolva, amely további védelmet biztosít egyéb alkalmazásokkal szemben.

Az osztályfájlok verifikációja:

A CLDC megköveteli, hogy a virtuális gép képes legyen kiszűrni a hibás osztályfájlokat. Habár már létezett a sztenderd osztályfájl verifikációs módszer a J2SE-ben, de ez a memória- és erőforrásszegény eszközökön nem alkalmazható, ezért egy alternatívát dolgoztak ki. Ebben az új módszerben minden betöltött Java osztályfájl tartalmaz egy úgynevezett „verem térkép” attribútumot. Ezt az attribútumot a CLDC vezette be, a The Java Virtual Machine Specification nem definiálja. Az attribútum egy „elővérifikációs” műveletet során kerül be az osztályokba, ily módon az osztályfájl minden metódusa ellenőrzötté válik. Az „elővérifikációs” művelet általában egy szerveren vagy egy asztali gépen van végrehajtva, mielőtt az osztályfájlok letöltődnek az eszközre. A „verem térkép” attribútum általánosságban 5 százalékkal növel meg az osztályfájlok méretét, de lehetővé teszi, hogy a CLDC-s virtuális gépek sokkal gyorsabban, jóval kisebb VM kóddal, és dinamikus RAM felhasználásával oldják meg az ellenőrzést, ugyanolyan hatékonysággal, mint az eredeti JVM-ben.



4. ábra, Osztályfájlok verifikációja a CLDC/KVM-ben

Osztályfájlok formája:

Ahhoz, hogy dinamikusan lehessen letölteni egy harmadik fél által készített alkalmazást vagy adatot, szükség van arra, hogy ezeket tömörített JAR (JAVA Archive) fájlokba rendezzük. A fájlok tömörítése az „elő verifikáció” folyamat után történik, így a JAR fájlban lévő osztályfájlok már tartalmazzák a verem térkép attribútumot.

CLDC osztályok

A Java SE-ből örökölt osztályok

A következő fájlok közvetlenül a Java 2 Standard Edition-ből öröklődnek. Ezen osztályok metódusai és attribútumai részei a komplett osztályoknak, melyeket a nagyobb Java kiadások is tartalmaznak, csak aktualizálva vannak a CLDC környezetéhez.

A Java core osztályai:

Rendszer osztályok

`Java.lang:`

`Object, Class, Runtime, System, Thread, Runable, String, Stringbuffer, Throwable`

Adattípus osztályok

`Java.lang:`

`Boolean, Byte, Short,Integer, Long, Character`

Collection osztályok

`Java.util:`

`Vector, Stack, Hashtable, Enumeration`

Dátum és Idő osztályok

`Java util:`

`Date, TimeZone, Calendar`

Kiegészítő osztályok

`Java util.Random,      java.lang.Math`

Kivételkezelő osztályok

`Java lang:`

`Exception, ClassNotFoundException, IllegalAccessException, InstantiationException, InterruptedException, RuntimeException, ArithmeticException, ArrayStoreException, ClassCastException,`

IllegalArgumentException, IllegalThreadStateException, NumberFormatException, IllegalMonitorStateException, IndexOutOfBoundsException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException, NegativeArraySizeExceptoin, NullPointerException, SecurityException

Java util:

EmptyStackException, NoSuchElementException

Java io:

EOFException, IOException, InterruptedException, UnsupportedEncodingException, UTFDataFormatException

Hibakezelő osztályok

Java.lang:

Error, VirtualMachineError, OutOfMemoryError

I/O osztályok

Java io:

Inputstream, Outputstream, ByteArrayInputstream, ByteArrayOutputstream, DataInput, DataOutput, DataInpuStream, DataOutputStream, Reader, Writer, InpuStreamReader, InpuStreamWriter, PrintStream

Eltérések a Java SE-től:

- A CLDC limitáltan támogatja az Unicode karaktereket.
- A CLDC nem támogatja a `java.util.Properties` osztályt mely része a Java SE-nek

CLDC-ben specifikált osztálok

A CLDC adatkapcsolata a Generic Connection-ön alapszik. Itt adatkapcsolatot Connection interfész felhasználásával tudunk létrehozni, melynek paraméterül egy URI-t kell megadni:

```
Connector.open („<protokol>:<address>;<parameters>”);
```

A Connection interfész használata különböző protokollok esetén :

- HTTP rekordok: `Connector.open("http://www.foo.com");`
- socketek : `Connector.open("socket://129.144.111.222:9000");`
- kommunikációs portok: `Connector.open("comm:0;baudrate=9600");`
- adatcsomagok: `Connector.open("datagram://129.144.111.333");`
- fájlok: `Connector.open("file:foo.dat");`
- hálózati fájl rendszerek: `Connector.open("nfs:/foo.com/foo.dat");`

Ez a példa csak a `Connector` használatát illusztrálja. A CLDC nem tartalmazza a protokoll implementálását. Hogy mely protokollok használhatóak, azt a profilok implementálóira bízza.

A K virtuális gép (KVM)

A KVM egy teljesértékű hordozható virtuális gép, amit szűkös erőforrás rendelkező eszközökre szántak, mint például a mobiltelefonok. A KVM neve arra utal, hogy a mérete néhány 10 **kilo**bájtra van korlátozva.
A KVM főbb jellemzői:

- a virtuális gép futásához 40-80 kilobájt statikus memória szükséges (ez függ a célhardvertől és a tartalmazott opcióktól),
- új hardverre könnyen és gyorsan telepíthető,
- modularitás és testreszabhatóság jellemzi

A KVM C nyelven lett implementálva, így könnyen telepíthető minden olyan platformon, amelyen elérhető egy C fordító. A forráskódjának jelentős része megegyezik a különböző platformokon, gép- és platformfüggő fájlt relatíve keveset tartalmaz. Ahhoz, hogy a KVM és az alatta futó operációs rendszer közötti kapcsolatot biztosítsák hogy néhány funkciót feltétlenül implementálni kell. Ilyen például:

- inicializálás
- heap allokálás / deallkokálás
- végzetes hiba esetén jelzés
- eseménykezelés
- időkezelés

A virtuális gép indítása és a JAM

KVM-et asztali gépen is futtathatunk. Itt az indítás parancssorból történik, ugyanúgy mint a Java SE esetében is. Olyan eszközökön, amelyek képesek betölteni az alkalmazásokat (ilyen a Palm OS), ez a módszer is rendelkezésre áll. Azon eszközökre viszont ahol ez nem lehetséges, a Java Application Manager (JAM) jelent megoldást. Ez kapcsolatot teremt a natív operációs rendszer és a virtuális gép között, így lehetővé válik a JAR formátumú alkalmazások letöltése és tárolása, a hálózati eszközkészlet (általában HTTP), vagy a GenericConnection használata. A JAM beolvassa a JAR fájlt és egy másodlagos leírófájlt, és betölti a KVM-et.

Natív kód:

A KVM nem támogatja a Java Native Interface (JNI). Ehelyett minden olyan natív kódot, amelyet a virtuális gép meghív, fordítási időben bele kell fordítani a virtuális gép kódjába. A natív metódusok hívásakor egy natív metódus leíró táblából hívjuk meg a metódust, mely tábla a programépítés folyamata során jön létre.

Java Code Compact (ROMizer)

A KVM támogatja a JAVA Code Compact (JCC) utility-t. Ez lehetővé teszi a Java osztályok belinkelését a virtuális gépbe, ezáltal csökkentve a VM indulási idejét. Implementációs szinten a Java fájlokból C fájlokat hoz létre, és befordítja a Java virtuális gépbe.

Ha a javac fordító használatával fordítjuk le az osztályainkat, akkor a forrásfájlból osztályfájlt kapunk, amit önmagába vagy JAR fáj részeként töltődik be a futtató rendszerbe.

A Java Code Compact egy kevésbé flexibilis megoldást jelent, de használatával csökkenthető a virtuális gép által igényelt memóriaigény.
A Java Code Compact képes:

- különböző bemeneti inputfájlokat kezelni,
- meghatározni egy objektum szerkezetét és méretét,
- csak a szükséges osztályrészeket betölteni a többi figyelmen kívül hagyásával.

A Java Code Compact Java-ban íródott, így könnyen alkalmazható bármely platformon.

A MIDP

A *Mobile Information Device Profile* (MIDP) egy kulcsfontosságú része a Java ME platformnak. A profil specifikációját a *Mobile Information Profile Expert Group* (MIDPEG) készítette, melyet több mint 50 cég alkot, köztük a vezető eszközgyártók, wireless szállítók és mobil szoftverkészítők . A MIDP és a CLDC együtt szabványos Java futási környezetet biztosít gazdag API gyűjteménnyel. Az első MIDP specifikáció 2000.09.01.-én jelent meg „*MIDP 1.0 Specification*” néven, melyet 2002.09.27.-án váltott a „*MIDP 2.0 Final Specification*”. Ez a verzió egészen mai napig használatos, habár a következő verziót már 2006.05.26.-án bejelentették (*MIDP 2.1 ChangeLog*).

A specifikáció definiálja azt a minimális hardver és szoftverkövetelményt amellyel egy MID eszköznek rendelkeznie kell.

Hardver követelmények:

- képernyő méret: 96x54 pixel, 1 bites színmélység, megközelítőleg 1:1 arányú kijelzővel.
- bemeneti eszköz: egykezes-, vagy kétkezes billentyűzet, vagy érintőképernyő.
- memória: 128KB ROM MIDP komponenseknek, 8KB ROM a MIDP alkalmazások számára szükséges hosszú távon tárolandó adatoknak (pl. beállítások), 32KB RAM a KVM heap területnek.
- hálózati elérés: kétirányú, nem feltétlen folyamatos, korlátozott sávszélességű kapcsolat.

Szoftver követelmények:

- Egy minimális *kernel* (mag) a rendelkezésre álló hardver kezelésére (megszakítás-, és kivételkezelés, minimális ütemezés). A kernel tudnia kell futtatni a Java virtuális gépet, viszont nem szükséges támogatnia a párhuzamos programfuttatást vagy a valós idejű ütemezést.
- Tudja írni és olvasni a nem illékony memóriát.
- Tudjon írni és olvasni a vezeték nélküli hálózati eszközökre/ről.
- Az időbélyeg használata a tárolóba bekerült adatokhoz.
- Képesség a grafikus kijelzőre való írásra.
- Képesnek kell lennie fogadni a felhasználói adatokat.
- Tudja kezelni az alkalmazások életciklusát.

A MIDP csomagok

Megpróbálták a csomagokat a lehető legszűkebbre szabni és csak azokat az API-kat belerakni, amelyek nélkülözhetetlenek az fenti követelményekhez. A MIDP csomagokat két nagy csoportba sorolhatjuk. Az egyik csoportba az úgynevezett Core (mag) csomagok, ezek a Java SE-ből lettek átemelve. A másik csoportba azok a csomagok kerültnek, amelyeket a Java ME vezetett be (mindegyik a `javax.microedition` csomagba tartozik)

A core csomagok:

- `java.lang` : A Java alaposztályait tartalmazza. A csomagot nem kell importálni, a csomagba tartozó osztályok bárhonnan elérhetőek. Ide tartozik pl: Object osztály, a primitív típusok burkoló osztályai (Boolean, Byte, Char, Integer, stb) a String és StringBuffer, a Math, a Thread és az Exception osztályok, illetve a Runnable interface.
- `java.util` : Olyan osztályokat és interfészeket tartalmaz, amelyek segédeszközként használhatunk a programban. Itt találhatóak a konténerek (Vector, Stack stb.), a naptárral, időkezeléssel kapcsolatos osztályok (Timezone, Date, Timer, TimerTask, stb.), valamint hasznos interfészek (pl: Enumeration).

A Java Me által bevezetett csomagok:

Felhasználói felület Csomag *`javax.microedition.lcdui`*

Osztályhierarchiája:

- class java.lang.**Object**
    - class javax.microedition.lcdui.**AlertType**
    - class javax.microedition.lcdui.**Command**
    - class javax.microedition.lcdui.**Display**
    - class javax.microedition.lcdui.**Displayable**
        - class javax.microedition.lcdui.**Canvas**
        - class javax.microedition.lcdui.**Screen**
            - class javax.microedition.lcdui.**Alert**
            - class javax.microedition.lcdui.**Form**
            - class javax.microedition.lcdui.**List** (implements javax.microedition.lcdui.Choice)
            - class javax.microedition.lcdui.**TextBox**
    - class javax.microedition.lcdui.**Font**
    - class javax.microedition.lcdui.**Graphics**
    - class javax.microedition.lcdui.**Image**
    - class javax.microedition.lcdui.**Item**
        - class javax.microedition.lcdui.**ChoiceGroup** (implements javax.microedition.lcdui.Choice)
        - class javax.microedition.lcdui.**CustomItem**
        - class javax.microedition.lcdui.**DateField**
        - class javax.microedition.lcdui.**Gauge**
        - class javax.microedition.lcdui.**ImageItem**
        - class javax.microedition.lcdui.**Spacer**
        - class javax.microedition.lcdui.**StringItem**
        - class javax.microedition.lcdui.**TextField**
    - class javax.microedition.lcdui.**Ticker**

Interfész hierarchiája:

- interface javax.microedition.lcdui.**Choice**
- interface javax.microedition.lcdui.**CommandListener**
- interface javax.microedition.lcdui.**ItemCommandListener**
- interface javax.microedition.lcdui.**ItemStateListener**

Fontosabb osztályai:

- **AlertType**: Az Alert osztály objektumainak típusát határozhatjuk meg vele. A figyelmeztetés történhet hanggal vagy vizuálisan. Előredefiniált típusok INFO, WARNING, ERROR, ALARM és CONFIRMATION.
- **Command**: Szemantikus információkat tartalmaz egy parancs által kiváltott eseményéről. A eseményt a CommandListener() interfész implementálja a CommandAction() metódusában.
- **Display**: Ez az osztály kezeli a képernyőt, a megjelenítést valósítja meg.
- **Displayable**: A megjelenített objektumok ennek az osztálynak a példányai tartalmazzák, így szoros kapcsolatban áll a Display osztállyal.
- **Canvas**: A Displayable osztály egyik alosztálya, mely alacsony szintű események kezelésére, illetve bizonyos elemek képernyőn való megjelenítésére szolgál.
- **Screen**: Displayable osztály másik alosztálya, mely az összes magas szintű felhasználói interfész-osztályt jelképezi.
- **Alert**: Valamilyen üzenetről értesíti a felhasználót.
- **Form**: Az űrlap képekből, szövegekből, grafikonokból stb. állhat. Ha az alkalmazás olyan elemet (Item) akar beilleszteni egy űrlapba, amely már benne van valamelyik Form objektumban, akkor IllegalStateException keletkezik, azaz egy elem egyszerre csak egy objektumban lehet benne. Az űrlap méretét a benne lévő elemek száma határozza meg.
- **List**: Listát tartalmaz, melynek elemei között lépegethetünk, választhatunk. Az alábbi listatípusok léteznek: IMPLICIT, EXCLUSIVE, MULTIPLE
- **TextBox**: Szövegdoboz-szerű képernyő-objektumokat lehet létrehozni.
- **Font**: Betűtípus kezelésére szolgál. Itt állítható be a betű stílusa (style), mérete (size), megjelenése (face).
- **Graphics**: Kétdimenziós grafikai műveleteket valósíthatunk meg vele a kijelzőn.
- **Image**: Mozgó- illetve állóképeket készíthetünk alkalmazásunkhoz.
- **Item**: Egy elemet reprezentál, melyet egy űrlapba helyezhetünk el.
- **ChoiceGroup**: A választóelemeket (jelölőnégyzetek, rádiógombok vagy választógombok) reprezentálja, melyeket szintén az űrlapokban használhatunk fel.
- **DateField**: A dátum és az idő megjelenítésére szolgál. Megjeleníthet csak dátumot, csak időt, illetve dátumot és időt egyszerre.
- **Gauge**: Grafikonok készítésére alkalmas.
- **Spacer**: A térközöket és az üres helyeket reprezentálja az elemek sorában, parancsok nem rendelhetőek hozzájuk.
- **TextField**: Az űrlapba illeszthető szövegelemeket reprezentálja.
- **Ticker**: A képernyőn folyamatosan futó szöveget hozhatunk létre. Nem lehet leállítani, csak ideiglenesen szüneteltetni.

Játék Csomag *`javax.microedition.lcdui.game`*

A játék csomagban definiált osztályok egy gazdag játékkörnyezet kialakítását teszik lehetővé.

Osztályhierarchiája:

- class java.lang.**Object**
    - class javax.microedition.lcdui.**Displayable**
        - class javax.microedition.lcdui.**Canvas**
            - class javax.microedition.lcdui.game.**GameCanvas**
    - class javax.microedition.lcdui.game.**Layer**
        - class javax.microedition.lcdui.game.**Sprite**
        - class javax.microedition.lcdui.game.**TiledLayer**
    - class javax.microedition.lcdui.game.**LayerManager**

Fontosabb osztályai:

- **`GameCanvas`**: A `Canvas` osztály leszármazottja, feladata a játékok felhasználói felületének megvalósítása. A parancsok hozzárendelését és a bemeneti eseményeket átveszi a `Canvas`-től.
- **`Layer`**: Többrétegű képernyő valósítható meg vele.
- **`Sprite`**: Mozgóképek készítésére használhatjuk. Egy Sprite objektum képkockára bontja a megjelenítendő képet, és a képkockák frissítésével valósítható meg az animáció.
- **`TiledLayer`**: Segítségével kis képkockákból építhetjük fel a megjelenítendő képet. Használata olyan képeknél ajánlatos, amely sok ismétlődést tartalmaz.
- **`LayerManager`**: A különböző rétegeket kezelésére szolgál.

Alkalmazás életciklus csomag `javax.microedition.midlet`

Ez a csomag tartalmazza a MIDP környezetbe futó alkalmazásokat

Osztályhierarchiája:

- class java.lang.**Object**
    - class javax.microedition.midlet.**MIDlet**
    - class java.lang.**Throwable**
        - class java.lang.**Exception**
            - class javax.microedition.midlet.**MIDletStateChangeException**

A MIDP definiál egy alkalmazási modellt amely meghatározza, hogy mit jelent egy MIDlet, hogyan kell azt csomagolni, milyen futási környezetben alkalmazható, és hogy milyen egy MIDlet viselkedése. Az alkalmazási modell megkövetel egy úgynevezett MIDlet suite-ot. Ha egy MIDP alkalmazást szeretnénk futtatni, akkor ezt a MIDlet suite-ot kell letöltenünk a futtató eszközre.

Egy MIDlet suite elemei:

- futtató környezet,
- MIDlet suite-csomagolás,
- alkalmazás leíró,
- alkalmazás életciklus.

Futtató környezet

Minden eszköz megköveteli, hogy a felhasználó telepítse, kiválassza, futtassa és eltávolítsa a MIDlet-et. Ezen funkciók automatizálására egy úgynevezett *application managment* szoftvert készítettek. A manager biztosítja a MIDP specifikációnak megfelelő futtató környezetet a MIDlet(ek) számára.

MIDlet suite-csomagolás

Egy MIDlet suite alatt a JAR fájlba csomagolt MIDlet-ket értjük. Ez a JAR fájl a következőket tartalmazza:

- az alkalmazás osztályfájlit.
- egy *manifest* fájlt
- a MIDletek erőforrás fájljait

Egy MIDlet mindig tartalmaz egy olyan osztályt a `javax.microedition.midlet.MIDlet` osztályt származtatja, ez az osztály tartalmazza az alkalmazás belépési pontját (`public void startApp()` ellenben a Java SE-ben megszokott `public static void main()`), így ez az az osztály, amelyet az application managment példányosít.

A *manifest* egy leíró fájl a MIDlet suite-ról az application managment számára. Beállítja azokat az attribútumokat, amelyek a MIDlet telepítéséhez és futtatásához szükségesek (a MIDlet osztályfájlt (azaz a belépési pontot), a MIDlet nevét és ikonját.

Az alkalmazás leíró

Ezt a leíró fájlt az application managment szoftver használja. Ez szolgáltat konfiguráció-specifikus attribútumokat a MIDlet számára. A leíró fájlban megjelenő attribútumok elérhetők a MIDletek számára a `MIDlet.getAppProperty` metódus segítségével. A leíró fájl általában `jad` kiterjesztésű.

A leírónak a következő kötelező attribútumokat:

- `MIDlet-Name`
- `MIDlet-Version`
- `MIDlet-Vendor`
- `MIDlet-Jar-URL`
- `MIDlet-Jar-Size`

A következőket opcionális attribútumokat tartalmazhatja:

- `MIDlet-Description`
- `MIDlet-Icon`
- `MIDlet-Info-URL`
- `MIDlet-Data-Size`
- `MIDlet specifikus attribútumok, melyek nem kezdődhetnek „MIDlet"-el.`

A kötelező attribútumoknak szerepelniük kell a leíró és a manifest fájlban is. Ha ezek az azonos nevű attribútumok nem egyeznek meg pontosan, akkor a JAR nem installálható. Egyéb attribútumok esetén az értékek eltérhetnek. Ekkor a leíró fájl tartalma felülírja a manifest fájlban lévő értéket.

Alkalmazás életciklusa

Ha egy MIDlet suite-t installálunk, akkor az osztályfájlok, erőforrásfájlok és a leíró attribútumok felmásolódnak az adott eszközre. Ha futtatunk egy MIDletet, akkor a application managment a `javax.microedition.midlet.MIDlet`-et származtató osztályt példányosítja. Példányosítás után a MIDlet futó állapotba kerül. A MIDlet kérheti az application manager-t, hogy egy adott állapotba sorolja. Az ehhez szükséges metódusok a következők:

`public void pauseApp()` metódussal a *Paused* állapot,

`public void destroyApp(boolean)` metódussal a *Destroye*d állapot és a

`public void resumeMIDlet()`metódussal az *Active* állapot kérhető.

Amikor egy MIDlet befejeződik, vagy az application managment megszakítja, akkor a példány megsemmisül, és az erőforrásait újra fel lehet használni.

Ha a MIDlet `System.exit`-tel áll le, akkor egy `SecurityException` dobódik.

Hálozati csomag `javax.microedition.io`

A MID profil hálózati támogatása, főleg a *Generic Connection* keretrendszerre alapszik, amit a CLDC definiál.

Osztályhierarchiája:

- class java.lang.**Object**
    - class javax.microedition.io.**Connector**
    - class javax.microedition.io.**PushRegistry**
    - class java.lang.**Throwable**
        - class java.lang.**Exception**
            - class java.io.**IOException**
                - class.javax.microedition.io.**ConnectionNotFoundException**

Interfész hierarchiája:

- interface javax.microedition.io.**Connection**
    - interface javax.microedition.io.**DatagramConnection**
        - interface javax.microedition.io.**UDPDatagramConnection**
    - interface javax.microedition.io.**InputConnection**
        - interface javax.microedition.io.**StreamConnection** (also extends javax.microedition.io.OutputConnection)
            - interface javax.microedition.io.**CommConnection**
            - interface javax.microedition.io.**ContentConnection**
                - interface javax.microedition.io.**HttpConnection**
                - interface javax.microedition.io.**HttpsConnection**
            - interface javax.microedition.io.**SocketConnection**
                - interface javax.microedition.io.**SecureConnection**
    - interface javax.microedition.io.**OutputConnection**
        - interface javax.microedition.io.**StreamConnection** (also extends javax.microedition.io.InputConnection)
            - interface javax.microedition.io.**CommConnection**
            - interface javax.microedition.io.**ContentConnection**
                - interface javax.microedition.io.**HttpConnection**
                - interface javax.microedition.io.**HttpsConnection**
            - interface javax.microedition.io.**SocketConnection**
                - interface javax.microedition.io.**SecureConnection**
    - interface javax.microedition.io.**StreamConnectionNotifier**
        - interface javax.microedition.io.**ServerSocketConnection**
- interface java.io.**DataInput**
    - interface javax.microedition.io.**Datagram** (also extends java.io.DataOutput)
- interface java.io.**DataOutput**
    - interface javax.microedition.io.**Datagram** (also extends java.io.DataInput)
- interface javax.microedition.io.**SecurityInfo**

A csomag az input/óutput kezeléséhez szükséges osztályokat és interfészeket tartalmazza (`InputStream`, `OutputStream`, `stb`). Az osztályok közül kiemelkedő szereppel bír a `Connector` osztály, mely segítségével egy kapcsolatot tudunk létrehozni és azt kezelni. Egy kapcsolatot létrehozására és megnyitására a

```
public static Connection open(String name,int mode,boolean timeouts)
                                   throws IOException
```

metódus szolgál. Első paraméterként egy URI-t kell megadni amely az egyetlen kötelező paraméter. Az URI-val az elérni kívánt erőforrást címezzük, struktúráját már (a szükséges protokollnak megfelelően) a CLDC specifikációjánál bemutattam. A második és harmadik paraméter opcionális, velük a hozzáférés módját definiálhatjuk (írás, olvasás, mindkettő),

illetve időtúllépés esetén kivételt tudunk generáltatni. A metódus egy `Connection` típusú attribútummal tér vissza. Ezt a visszatérési értéket a kívánt típusra kell konvertálni a csomagba megadott interfészek segítségével (HttpConnection, SocketConnection, stb).

Például egy soros portról való olvasást a következő módon tehetünk meg:

```
CommConnection cc =(CommConnection)Connector.open("comm:0;baudrate=115200",Connector.READ);
```

Publikus kulcsot kezelő csomag `javax.microedition.pki`

A MIDP 2.0 egyik újítása, segítségével biztonságos kapcsolatot építhetünk ki. A csomag megteremtését azon elhatározás ösztökélte, hogy minden MIDP 2.0 eszköznek legalább az X.509-es titkosítást ismernie kell.

Osztályhierarchiája:

- class java.lang.**Object**
    - class java.lang.**Throwable**
        - class java.lang.**Exception**
            - class java.io.**IOException**
            - class javax.microedition.pki.**CertificateException**

Interfész hierarchiája:

- interface javax.microedition.pki.**Certificate**

Perzisztens adattárolás csomag `javax.microedition.rms`

A MIDP tartalmaz egy olyan mechanizmust, amely segítségévem lehetővé válik adatok prezisztens tárolása, és későbbi felhasználása. Ezt a mechanizmust **RMS-nek** (Record Management System) nevezték el. Az RMS-t megvalósító osztályok és interfészek kerültek ebbe a csomagba. Az RMS úgy működik, mint egy rekord szintű adattároló, gondoskodik az adatok sérthetetlenségéről, integritásáról (beleértve az akkumlátor cserét, vagy újraindítást is).

Osztályhierarchiája:

- class java.lang.**Object**
    - class javax.microedition.rms.**RecordStore**
    - class java.lang.**Throwable**
        - class java.lang.**Exception**
            - class javax.microedition.rms.**RecordStoreException**
                - class javax.microedition.rms.**InvalidRecordIDException**
                    - class javax.microedition.rms.**RecordStoreFullException**
                    - class javax.microedition.rms.**RecordStoreNotFoundException**
                    - class javax.microedition.rms.**RecordStoreNotOpenException**

Interfész hierarchiája:

- interface javax.microedition.rms.**RecordComparator**
- interface javax.microedition.rms.**RecordEnumeration**
- interface javax.microedition.rms.**RecordFilter**
- interface javax.microedition.rms.**RecordListener**

Global Positioning System

A hidegháború idején az Amerikai Egyesült Államok Védelmi Minisztériuma (Department of Defense) dolgozta ki a Globális Helymeghatározó Rendszert (Global Positioning System, NAVSTAR GPS) amely egy globális műholdas navigációs rendszer (GNSS – Global Navigation Satellite System). Pontossága jellemzően méteres nagyságrendű, de differenciális mérési módszerekkel akár mm-es pontosságot is el lehet érni, valós időben is.

A helymeghatározás 24 db műhold segítségével történik, melyek a Föld felszíne fölött 20 200 km-es magasságban keringenek, az egyenlítő síkjával 55°-os szöget bezáró pályán. Sík terepen egyszerre 7-12 műhold látható, melyből a helymeghatározáshoz 3, a tengerszint feletti magasság meghatározásához pedig további egy hold szükséges.

A GPS műholdak két frekvencián sugároznak, ezeket L1-nek (1575,42 MHz) és L2-nek (1227,6 MHz) nevezik. Minden műhold szórt spektrumú jelet sugároz, amit „pszeudo-véletlen zaj”-nak lehet nevezni (angol megnevezése: pseudo-random noise, röviden: '*PRN')*. Ez a PRN minden műholdnál különböző

Minden műholdon két darab rubídium- vagy cézium-atomóra van elhelyezve. Az oszcillátorok biztosítják az alapfrekvencia és a kód előállítását is. Az alapfrekvenciát az USDOD földi állomásai felügyelik, amit egyeztetnek az egyezményes koordinált világidővel (UTC) (amit az United States Naval Observatory (USNO) állít elő), azonban a két időfogalom és érték nem azonos egymással. Kölcsönös egyeztetéssel az USNO és a NIST által előállított UTC-idő 100 ns-on belül megegyezik egymással.

A civil felhasználásban az áttörést a Selective Availability (SA), a pozicionálás pontosságát szándékosan csökkentő, zavaró jel 2000. május 1-én történt kikapcsolása jelentette. Így ma a rendszer az addig alkalmazott költséges differenciális javítások nélkül is 10-30 méteres pontosságú helymeghatározást képes biztosítani. A tervek szerint a következő évtizedben valósul meg a modernizált, GPS III nevet viselő rendszer.

Több nagyhatalom is igyekszik saját GNSS rendszert fejleszteni. Az orosz GLONASS már működőképes, igaz kevés műholdja miatt gyenge lefedettséget biztosít. Az Európai Unió által finanszírozott Galileo projekt előreláthatóan a következő évtizedben lesz kész. A Galileo kompatibilis lesz a NAVSTAR rendszerrel, ezért megjelenésével javuló lefedettségre és megnövekedett pontosságra számíthatunk.

A helymeghatározási módszer

A műholdas helymeghatározó rendszer időmérésre visszavezetett távolságmérésen alapul. Mivel ismerjük a rádióhullámok terjedési sebességét, és ismerjük a rádióhullám kibocsátásának és beérkezésének idejét, ezek alapján meghatározhatjuk a forrás távolságát. A háromdimenziós térben három ismert helyzetű ponttól mért távolság pontos ismeretében már meg tudjuk határozni a pozíciót. A további műholdakra mért távolságokkal pontosítani tudjuk ezt az értéket.

Az eljárás lépései

1. A GPS-vevő folyamatosan rendelkezzen a műholdakon lévő atomórák pontos idejével.
2. Legalább 4 műhold láthatósága esetén „háromszögeléssel” meghatározható a földfelszíni pozíció.
3. Ehhez ismerni kell a vevő és a műholdak pontos távolságát, amelyhez a műholdak aktuális pályájának és a kisugárzott jel megérkezési idejének ismerete szükséges.
4. Hibák és korrekciók.

1. A GPS-vevőnek először a műholdakkal folyamatosan egyeztetett pontos időre van szüksége, ehhez a PRN-kódot használja fel. A PRN-kód jelzi a vevőnek, hogy melyik műhold jelét veszi, és az adott műholdtól milyen álvéletlen jelsorozatra számíthat. A ténylegesen megkapott és a vevőben várt jel egyedi mintázattal rendelkezik, ennek ismeretében a vevő megállapítja a jelek időbeli eltérését, és a saját óráját ennek megfelelően járatja.

2. Igazából nem „háromszögelés”-ről van szó, mivel három vagy több távolságból állapítjuk meg a vevő térbeli helyzetét, így „háromtávolságolás”-nak vagy latinosan *trilaterációnak* nevezhetjük ezt az eljárást. Elméletileg 3 műhold is elég lenne ehhez, ha mindegyik órája tökéletesen járna, a gyakorlatban azonban a rendszer ismert pontatlanságait figyelembe véve legalább 4 műholdat használnak a pozíció meghatározásához. A műholdaktól való távolság kiszámításához ugyanazt a módszert használja a vevő, mint a pontos idő szinkronizálásánál: a műholdról sugárzott és a vevőben meglévő idők eltérését állapítja meg. Az időbeli különbség szorozva a rádióhullámok terjedési sebességével kiadja a vevő és az adott műhold távolságát.

3. A vevő és a műholdak távolságához ismerni kell a műholdak aktuális pozícióit. Ehhez a műholdak kisugározzák az ún. *almanach* adatokat (ez a vevőkészülék bekapcsolásakor, illetve később periodikusan megtörténik), amelyek az egyes műholdak pályaadatait tartalmazzák. Ennek ismeretében a vevő kiszámítja a műhold Föld feletti helyzetét. Az Amerikai Védelmi Minisztérium (USDOD) folyamatosan radarokkal követi a műholdakat, és méri azok földfelszínhez viszonyított pozícióját, sebességét és magasságát. Ezekkel az adatokkal korrigálják a műholdakban lévő pályaelemeket (amelyeket a műholdak lesugároznak a vevő felé).

4. A műholdakon lévő atomórák nagyon pontosak, de nem tökéletesek. Az eltéréseket a földi állomások figyelik és szükség esetén korrigálják azokat.

A pályaelemek folyamatosan változnak a különféle zavaró hatások következményeként (ezeket összefoglaló néven „*efemerisz-hibának*” nevezik, mivel végső soron a műhold pályájára vannak hatással). Ilyen zavaró hatás a Föld anyageloszlásának, és így gravitációjának egyenetlenségei, a Nap és a Hold gravitációs hatása, illetve a napszél eltérítő ereje (amelyek mindig más irányból hat a műholdra). Bár ezek a hatások önmagukban kis pontatlanságot okoznak, mindet figyelembe veszik a pontos pályaszámításokhoz.

Jelentősen nagyobb torzítást okoz a rendszerben a légkör hatása a rádióhullámokra. A számítások leírásánál feltételeztük, hogy egyszerűen a távolság = sebesség x idő képlettel számolunk. Ez igaz is, csakhogy a rádióhullámok sebessége csak vákuumban állandó.

Ahogy a műhold jele a Föld felé terjed, áthalad az elektromosan töltött részecskéket tartalmazó Van Allen sugárzási övön, majd a vízpárát tartalmazó troposzférán, és mindkettőben valamennyire lelassul a vákuumbeli sebességhez képest.

Több módszer kínálkozik ennek a hibának a minimalizálására. Az egyik, hogy a hatás mértéke ismert, a korábbi mérésekből alkotott modellek alapján jól közelíthető egy adott napra. Azonban a légkör állapota soha nem állandó és soha nem pontosan ugyanaz. Ezért általában más módszert használnak a hibák kiküszöbölésére.

Felhasználható az L1 és L2 frekvenciák különbözősége, ugyanis a légkör hatása frekvenciafüggő. (Ezt a módszert csak a katonai vevők tudják kihasználni.)

A **differenciális GPS** (röviden: DGPS) elve kihasználja azt a tényt, hogy a földfelszín egy adott, ismert pontján lévő rögzített vevőkészülék milyen eltéréseket tapasztal a műholdakról sugárzott és az általa más forrásból megkapott jelek között. Az eltérések a többi hibaforrás számításba vétele után a légkör torzító hatásának tudható be. Ez a hibát csökkentő érték egyes rendszerekben (WAAS, EGNOS) letölthető az erre specializált műholdakról. Az így megnövelt pontosság csak a földi állomás környezetében használható ki igazán (ez tipikusan néhány száz km), ahol a légkör állapota még megegyezik a földi állomás fölötti légkör állapotával.

Az épületekről és nagyobb tárgyakról visszaverődő jel is eljut a vevőig és ezzel meghamisíthatja a pontos távolság kiszámítását.

Tipikus hibák (eredmény méterben)

A hiba oka	standard GPS	differenciális GPS
műhold órája	1,5	0
pályahiba	2,5	0
ionoszféra	5,0	0,4
troposzféra	0,5	0,2
vevő zaja	0,3	0,3
visszaverődés	0,6	0,6

Az NMEA 0183 szabvány

Az NMEA 0183 szabványt a National Marine Electronics Association dolgozta ki eredetileg különféle hajó navigációs célokra. Napjainkra a GPS-vevőkből kommunikációs porton kinyert információ leggyakrabban használt formátuma. A szabvány többféle fizikai rétege támogat, a legelterjedtebb a RS-232 szabványos soros port. A kommunikációra jellemző, hogy egyirányú, ezért a szabvány definiál egy beszélőt (talker) és egy hallgatót (listener).

Az alkalmazási réteg definiálja az úgynevezett NMEA mondat (sentences) szerkezetét, amely mindig „$" karakterrel kezdődik, majd ezt követi a mondattípus azonosító (pl: GPGGA, GPRMC). Az azonosító definiálja, hogy a következő, vesszővel elválasztott, adatok miként értelmezendőek (koordináta, sebesség, idő, stb) .
Többféle típusú NMEA mondat létezik, dolgozatomban ezek közül az alábbi típust használtam:

$GPRMC

Ajánlott minimális adat műholdas helymeghatározáshoz. Tartalmazza a pillanatnyi időt, pozíciót, sebességet, haladási irányt és a pozíció minőségére vonatkozó indikátorokat.
Pl.: $GPRMC,213143.11,A,4947.7108,N,00107.4796,E,4.96,0.00,021008,0.0,E,A*39

- Üzenettípus: *„GPRMC"*
- Aktuális UTC idő ÓÓPPMM.EEE formátumban. *(pl. 235723.57 = 23:57:23.57)*
- Pozíció rögzítve van-e? *(A: rögzítve, V: nincs rögzítve)*
- Szélesség fokban, tört percben (*XX°YY.YYYY'*) + félgömb egyenlítőtől (N: északi, S: déli) *(pl. 4755.17 = 47°55.17').*
- Hosszúság fokban, tört percben (*XX°YY.YYYY'*) + félgömb Greenwichtől (E: keleti, W: nyugati) *(pl. 2162.22. = 21°62.22')*
- Sebesség csomóban (1 csomó = 1,852 km/h)
- Aktuális dátum *NNHHÉÉ* formában *(pl. 010509 = 2009.05.01.)*
- Mágneses elhajlás (*Magnetic Variation*), opcionális: a lokális északi irány (amely az iránytű északi végének irányába mutat) adott pontban való eltérése a tényleges északi iránytól .
- Mód, opcionális, csak a 2.3 verziótól: a pozíció rögzítésének módját jelzi. Lehetséges értékei: *A=Autonomous, D=Differential, E=Estimated, N=Not valid, S=Simulator*. Csak az A és D értékek esetén beszélhetünk tényleges pozícióról.

Összefoglalás

Ebben a fejezetben bemutattam a J2ME platformot, a Globális Helymeghatározó rendszert és az általa használt NMEA 0183 szabványt. Megismerkedhettünk a legkisebb Java kiadás struktúrájával, a a mobiltelefonok és PDA-k számára kialakított konfigurációról és profilról. Bemutattam a Java ME-ben használt virtuális gépet, a KVM-et. Továbbá olvashattunk még a GPS működési elvéről, a gyakorlati alkalmazása során felmerülő problémákról, és annak megoldásairól.

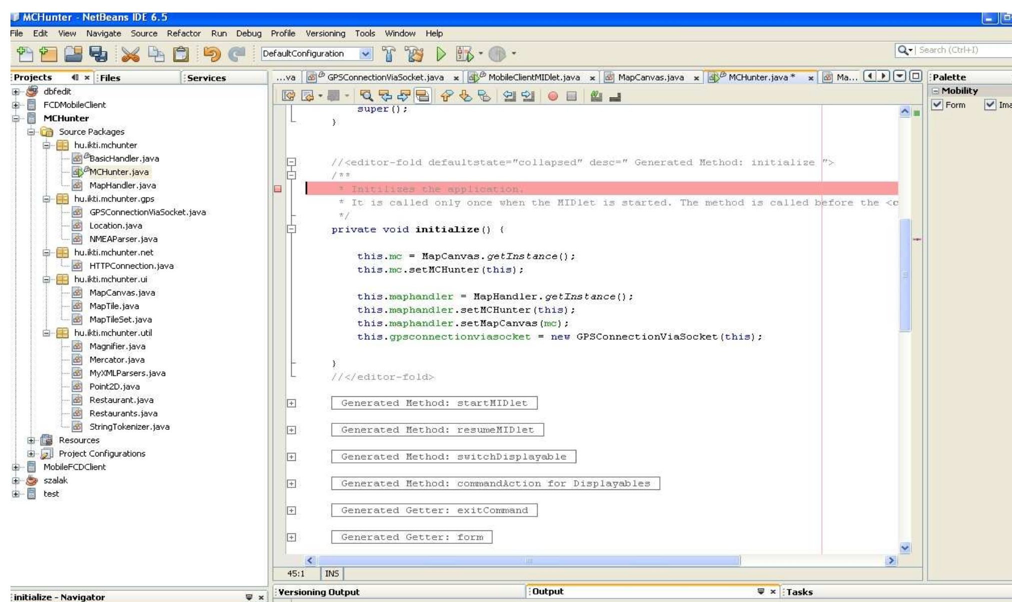
McHunter alkalmazás fejlesztése

A MCHunter alkalmazás főként demonstrációs célzatú. Elképzelésem az volt, hogy megismertessem az olvasóval, hogy hogyan lehet a gyakorlatban alkalmazni az előzőekben bemutatott Java ME környezetet. A bevezetőben már említettem, hogy a McHunter alkalmazás ötlete a Bay-IKTI-ben folyó MyTraffic projekt fejlesztés közben támadt.
A MyTraffic fejlesztésekor egyik legfontosabb kritérium a könnyen portolhatóság volt. A könnyen portolhatóság esetén a Java nyelv kézenfekvőnek tűnik, mivel ez a nyelv platformfüggetlenséget hirdet magáról. A Java kínál is egy ilyen környezetben használható kiadást (az előzőekben már ismert Java ME-t). Egy másik fontos fejlesztési kritérium, hogy a MyTraffic futatásához egyetlen fizetős komponenst se használjunk fel. Java támogatást szinte minden napjainkban forgalomba lévő készülék hirdet magáról (ez esetben nem kell külön JVM-et vásárolni a Java kódok futtatáshoz), ezért a Java ME e téren is megfelelő választásnak tűnt. Ezek után meghoztuk a döntést, a MyTraffic program egy Java ME kiadásba tartozó CLDC konfigurációjú MIDlet alkalmazás lesz. A program fejlesztése során számos gyakorlati tapasztalatra tettünk szert, amelyet a McHunter alkalmazás ismertetése során bemutatom az olvasónak. Az McHunter alkalmazásra is kiterjesztettem a fenti két kritériumot, azaz a könnyen portolhatóság és a teljesen ingyenes komponensekből való építkezést.
Az alkalmazásom fejlesztéséhez a Netbeans 6.5-ös fejlesztői környezetet használtam. A Netbeans számos segítséget biztosít az alkalmazás fejlesztéséhez, ilyen például:

- kód írása közbeni hibadetektálás
- a Mobiliti Pack használata esetén integrált CLDC emulátor
- lehetőség forráskódok generálására (konstruktorok, seter és geter metódusok)
- verziókövetés támogatása
- és ingyenes (ez manapság nem utolsó szempont)

A Netbeans 6.5 fejlesztő környezet a következő webhelyről letölthető:

```
http://www.netbeans.org/
```



5. ábra, NetBeans IDE 6.5

Az alkalmazás bemutatása

Az alkalmazásomat egy *HTC Touch HD*, és egy *Nokia N95 8GB* típusú eszközön fejlesztettem. Az első készülék egy *Windows Mobile 6.1* operációs rendszert futtat melyen a Java támogatást az *Esmertec* cég *Jbed* terméke biztosítja. A *Nokia N95 8GB* eszköz *S60* platformú, amely szintén támogatja a Java alapú programfejlesztést.
Mivel a HTC-re előre telepített Jbed, csak MIDletek futtatására alkalmas, ezért a McHunter egy CLDC kategóriájú, MIDP 2.0 alkalmazás. A feladat CDC alkalmazásként is megvalósítható lett volna, létezik Windows Mobile platformra fejlesztett CDC konfigurációjú JVM (ilyen például az IBM J9, vagy a NSIcom CrE-ME), de ezek használata mind licenszdíjas, így sérülnének a követelménylistába foglaltak.

Az alkalmazás működése

Az alkalmazás célja, hogy megjelenítse a közelünkben lévő McDonald's éttermeket. Ehhez az alkalmazásnak ismernie kell az éttermek pontos elhelyezkedését. Ezt az ismeretet az alkalmazás egy szerverről tölti le. Az éttermek koordinátái egy *kml* típusú fájl tartalmazza. A KML (*K*eyhole *M*arkup *L*anguage) XML-alapú jelölőnyelv térben ábrázolt alakzatok megjelenítésére.
A letöltött kml fájlt ezután fel kell dolgoznia. Ez többféleképpen történhet. Létezik XML kezelésére készített osztály, de történhet egyszerű string-feldolgozással is. A feladat magoldása során létrehoztam egy Restaurant nevű osztályt, melynek objektumai egy-egy éttermet reprezentálnak. A kml fájl feldolgozása során ilyen Restaurant példányokat hozok létre és helyezem el őket egy Vector típusú objektumba.
Ezután az alkalmazás már rendelkezik az éttermek pontos elhelyezkedésével, viszont a saját földrajzi koordinátáit még nem ismeri. Ezt az eszközbe integrált GPS vevőtől szerzi be. Az aktuális koordináták beszerzésére is több alternatívát kínál a Java. A legegyszerűbb megoldásnak a Java külön erre a célra kialakított *Location API* felhasználása tűnik. Másik lehetséges mód, ha direktbe kapcsolódunk a GPS vevőhöz, általában egy soros porton keresztül. A HTC eszköz esetében egy harmadik megoldást kellet alkalmazni, amit a későbbiekben ismertetek.
A saját koordinátái birtokában már megszerezheti az őt körülvevő környezetet ábrázoló térképrészletet. Ha az eszközön rendelkezésre áll egy térképadatbázis (mint a navigátorrendszerek esetén), akkor a térképrészletet beolvasása történhet innen is, ellenkezőleg egy térképszolgáltatótól szerezzük be a szükséges képet (például a *Google Maps* szolgáltatás felhasználásával).
Ha már rendelkezésünkre áll a megfelelő térképrészlet akkor ezt meg kell jeleníteni. A megjelenítésre egy *Canvas* osztály objektumát használom, egyszerűsége miatt. A megjelenítés során a térképrészletet illetve a „közeli" éttermeket a Canvas megfelelő pontjaira illesztem, majd kérem annak kifestését a *repaint* metódussal. Mivel egy mobil alkalmazásról van szó, időnként (én másodperces periodikással) frissítem az aktuális koordinátákat, így a GPS vevőt való adatbeszerzést és az azt követő lépések periodikusán ismétlődnek.

Részletes bemutatás:

Az éttermek POI adatainak beszerzése

A POI adatok egy szerverről kerülnek letöltésre. A szerver és az alkalmazás közötti kommunikációt megvalósító osztály a *HTTPConnection*, mely a *hu.ikti.mchunter.net* csomagban található. Az adatok ezen osztály statikus *getViaHttpConnection* metódusának felhasználásával történik, mely egy *String* típusú objektumot vár hívási paraméterül, és egy ugyanilyen típusú objektummal tér vissza. A hívási paraméter egy URI, amely a letöltendő fájlt jelöli (jelen esetben ez: *http://91.83.41.226/2/McDonald.kml*). A metódus a *Generic Connection* keretrendszert használja, azaz a *Connector.open* metódussal építi ki az adatkapcsolatot. Az adatok beolvasása, a Java SE-ből jól ismert, *InputStream* csatornaosztályt segítségével történik. Az InputStream tartalmát annak read metódusával kiolvassuk és egy byte tömbön keresztül egy String objektumba pakoljuk. Ez az objektum képezi a metódus visszatérési értékét.

```
public static String getViaHttpConnection(String url) throws IOException {
    HttpConnection hc = (HttpConnection) Connector.open(url, Connector.READ, false);
    hc.setRequestMethod(HttpConnection.GET);
    InputStream in = hc.openInputStream();
    int contentLength =(int) hc.getLength();
    byte[] raw = new byte[contentLength];
    for(int i=0; i < contentLength; i++){
        raw [i] = (byte) in.read();
    }
    int length = in.read(raw);
    in.close();
    hc.close();
    String data = new String(raw, 0, length);
    data = data.trim();
    return data;
}
```

A letöltött POI adatok feldolgozása

A letöltött POI adatok *kml* fájl formájában vannak tárolva. A *kml* egy XML alapú fájl, tehát valójában ez egy XML fájl feldolgozását jelenti.

Példa egy KML fájlra:

```
<?xml version="1.0" encoding="UTF-8"?>
        <kml xmlns="http://earth.google.com/kml/2.0">
        <Placemark>
          <description>New York City</description>
          <name>New York City</name>
          <Point>
             <coordinates>-74.006393,40.714172,0</coordinates>
          </Point>

      </Placemark>
```

Erre a feladatra létezik beépített eszközök a Java-ban, melyeket a *javax.xml* és a org.xml.sax[4] csomagok tartalmaznak. Ekkor az adatok feldolgozása a következőképpen néz ki:

Valahol az alkalmazásban el kell helyezni e következő sorokat (itt egyben az adatok letöltése is megvalósul a *hu.ikti.mchunter.net* csomag *HTTPConnection.java* osztálya felhasználása nélkül):

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
String uri = " http://91.83.41.226/2/McDonald.kml";
HttpConnection hc = (HttpConnection) Connector.open(uri, Connector.READ, true);
InputStream in = hc.openInputStream();
saxParser.parse(in,new BasicHandler(this));
```

Emellett létre kell hozni egy *BasicHandler* nevű osztályt. A *BassicHandler* osztály a *DefaultHandler* osztályt származtatja, feladata a XML dokumentumok adatainak tárolása. Ez a megoldás implementálva lett a McHunter alkalmazásban, és tökéletes működést tanúsított mind az emulátoron mind az S60 platformon futtatva.

KVM megpróbálta betölteni az XML-t kezelő osztályokat a program egy *NoClassDef Found Error* kivétel dobásával terminált. Úgy tűnik, hogy az *Esmertec* cég a *Jbed* termékébe nem

[4] SAX = Simple API for XML

implementálta ezt a funkciót. Ezek után egy kevésbe elegáns (de működő) megoldást alkalmaztam. Egy saját XML feldolgozó osztályt írtam, amely egyszerű string-műveletekkel szétszedi az XML dokumentumot megfelelő részekre, és gondoskodik azok tárolásáról. Az XML feldolgozó osztályom neve *MyXMLParsers* és a *hu.ikti.mchunter.util* csomagba található. A *MyXMLParsers* az XML feldolgozásához igénybe veszi az ugyanebben a csomagban található *StringTokenizer* osztályt, amely egy szeparáló karakter alapján képes darabokra (*token*-ekre) szedni egy *String* objektumot. Az adatok tárolása Restaurant típusú objektumok formájában történik. Egy Restaurant objektum a következő attribútumokkal rendelkezik:

egy Image típusú (amely az étterem logóját tartalmazza),

egy szélességi,

és egy hosszúsági koordinátát reprezentáló attribútumokkal.

Mivel a megjelenítés két dimenzióban történik, ezért a magasság koordinátát nem tároljuk. Ezeket a *Restaurant* típusú objektumokat egy *Restaurants* konténerbe helyezzük. A *Restaurants* osztályt a *Vector* osztályból származtatjuk, és azt csak egy rendezést megvalósító sort metódussal egészíti ki. A sort metódusra azért van szükség, hogy a *restaurants* (ami a *Restaurants* egyetlen példánya) az éttermeket azok X koordinátájuk szerint növekvő sorrendbe tárolja. Ez azért hasznos, mert a megjelenítés során csökkenti a *Restaurant* objektumok vizsgálatának számát (ez főképpen az ország nyugati részén élő felhasználók esetén jelentős).

A MyXMLParsers osztály két attribútuma a (Restaurant) currentRestaurant és a (Restaurants) restaurants attribútumok. A következő értékadás után (amit a konstruktor tartalmaz) restaurants = Restaurants.getInstant() a restaurants referencia a Restaurants osztály egyetlen példányára fog mutatni. Erre az egy példányra való hivatkozást a Restaurants következő metódusa biztosítja:

```
private static Restaurants ref = null;

…

public static Restaurants getInstant(){

           if (ref == null){

                ref = new Restaurants();

           }

           return ref;

        }
```

A POI adatokat egy string formájában kapja meg a *MyXMLParser*. Ezt a stringet a ’<’ karakterek alapján a *StringTokenizer* osztály toknekre tördeli. A tokenek tartalma alapján a következő lépések valamelyike történik:

Ha a token a "*Placemark>*" stringgel kezdődik, akkor létrehozunk egy új Restaurant típusú objektumot:

```
if(token.startsWith("Placemark>")){
      this.currentRestaurant= new Restaurant();
}
```

Ha a kezdő karakterek a *name>* : az újonnan létrehozott objektum nevét beállítjuk:

```
if(token.startsWith("name>")){
   this.currentRestaurant.setName(token.substring(5));
}
```

Ha a kezdő karakterek a *coordinates>*: a setPoint2D metódussal a tokenben lévő koordinátákat kiszedjük és beállítjuk az objektum koordinátáit. Ezt követően bővítjük az éttermek listáját:

```
 if(token.startsWith("coordinates>")){
     this.setPoint2D(token.substring(12));
    restaurants.addElement(currentRestaurant);
}
```

Ezek a műveletek a fent megadott sorrenden (a sorrendet a *kml* fájl struktúrája tartalmazza) addig folytatjuk, amíg el nem éremjük a string végét, azaz a *kml* fájl végig nem olvassuk. Ekkor a *retaurants* tartalmát annak sort metódusa segítségével X koordináta szerint növekvő sorba rendezzük.

```
public void sort(){

          Restaurant a, b;

          for (int i = this.size()-1; i>1; i--){

               for(int j= 0; j < i; j++){

                  a = (Restaurant) this.elementAt(j);

                  b = (Restaurant) this.elementAt(j+1);

                  if(a.getCoordinates().getX() > b.getCoordinates().getX()){

                       this.removeElementAt(j+1);

                       this.removeElementAt(j);

                       this.insertElementAt(a, j);

                       this.insertElementAt(b, j);

                  }

}

    }
```

```
}
```

A sort egy klasszikus buborékrendezést valósít meg. Mivel Magyarországon jelenleg 94 darab étterem létezik, ezért ez az algoritmus lassúsága nem okoz problémát.

A valós földrajzi koordináták megszerzése

A koordináták beszerzéséhez az integrált GPS vevőhöz kell kapcsolódni valamilyen módon. Erre is több járható út kínálkozik. Használhatjuk a Java erre a célra definiált eszközét, a *Location API*-t.
Ekkor az API *Location* osztályát vesszük igénybe. Ezen osztály *getQualifiedCoordinates()* metódusa egy szintén ebben az API-ben definiált *QualifiedCoordinates* típusú objektumot ad vissza. A *QualifiedCoordinates* osztály a *Coordinates* leszármazottja, ezért a visszakapott objektumot egyszerűen *Coordinates* típusra tudjuk konvertálni. Ekkor már lehetőségünk van az X és Y koordináták megszerzésére ezen osztály `getLatitude()` és `getLongitude()` metódusaival.

```
Location location = mcHunter.getLocationProvider().getLocation(180);

// Get the coordinates of the current location.
Coordinates coordinates = location.getQualifiedCoordinates();

if (coordinates != null) {

   // Get the latitude and longitude of the coordinates.
   double latitude = coordinates.getLatitude();
   double longitude = coordinates.getLongitude();
   double ts = location.getTimestamp();
}
```

Ez a kódrész az emulátoron és a Nokián a specifikációnak megfelelően működött. Nem így viszont a HTC-n. A Jbed ismét egy *NoClassDef Found Error* kivétel dobásával örvendeztetett meg minket. Az interneten történő rövidebb keresgélés után meg is kaptam a választ, a Jbed nem tartalmazza a *Location API* (a JSR 179 -t). Mivel követelményként állítottuk fel, hogy egy több eszközön futtatható kódot alkossunk ezért más megoldás után kellet nézni.

Egy másik lehetséges út, ha közvetlen rákapcsolódunk a GPS vevőre, és az általa szolgáltatott NMEA mondatot mi magunk dolgozzuk fel és tároljuk. A HTC Touch Hd PDA-n, a GPS vevőt a COM4 soros porton 9600-as baudrate-en keresztül lehet elérni. A *Generic Connection* e céljára szolgáló eszköze a *CommConnection*, melynek használatát a következő kód szemlélteti:

```
CommConnection cc = (CommConnection)Connector.open("comm:com4;baudrate=9600");
```

```
        int baudrate = cc.getBaudRate();
        InputStream is  = cc.openInputStream();
```

A *CommConnection* használata teljesen azonos szemantikájú mint a *HttpConnection*-én. A *Connector.open* metódussal kiépítjük az adatkapcsolatot. Az adatok eléréséhez az *InputStream* csatornaosztálytájon keresztül lehetséges, amelyből már az ismert módon kinyerhetjük az információt. Implementáltam ezt a megoldást is, és láss csodát nem kaptam hibaüzenetet. A sikert kissé beárnyékolta viszont, hogy az NMEA mondatok sem érkeztek. Ezt a „csekély" problémát hosszabb debug-golás után sem sikerült megoldani. Mit is gondolhatnánk, 3:0 a Windows Mobile és az Esmertec cég által alkotott csapat javára.

A megoldás ismét a Google szállította. Rengetegen próbálkoztak eme probléma megoldásával, de sikerrel senki sem járt. Viszont a TrekBuddy cég egy igen nyakatekert, de használható megoldást talált, amit jobb híján én is adoptáltam. A TrekBuddy cég dolgozói feladták annak esélyét, hogy Java kódból közvetlen módon kommunikálnak a GPS vevővel. Ehelyett egy *#C* nyelven írt programmal felolvassák az NMEA mondatokat egy *socket* portra. Ekkor ugyan egy #C és egy Java programot kell egyszerre futtatni, de így Windows Mobile platformon Esmertec CLDC kategóriájú KVM alatt is sikerül GPS koordinátákhoz jutnunk (3 : 1).

A *com* port-ról *socket* port-ra olvasó programot ingyenesen letöltöttem, és a kódot ennek megfelelően implementáltam. Létrehoztam egy *GPSConnectionViaSocket* egy *Location* és egy *NMEAParser* osztályt melyek a *hu.ikti.mchunter.gps* csomagba kerültek. A GPS adatok kezelése a következő:

A *GPSConnectionViaSocket* származtatja a *Thread*-et, ezáltal a GPS adatok letöltése egy külön szálba történik. Ezt az osztályt a *McHunter* (a főprogram) példányosítja. Példányosítás után az objektum *connectToDevice* metódusát meghívva beállítjuk a megfelelő paramétereket (hostname = 127.0.0.1 és port = 20175), ebből összeállítjuk a *Generic Connectionhoz* szükséges URI-t, és kapcsolódunk a socketre.

A socketre kapcsolódás után megkeressük az első '$' karaktert (`while ((input = reader.read()) != 13) { ; }`), ez jelzi egy NMEA mondat kezdetét. Ezután belépünk egy végtelen ciklusba, és folyamatosan olvassuk a socketen lévő karaktereket. A beolvasott karaktereket egy StringBufferben gyűjtjük össze. Ha az aktuálisan olvasott karakter a '$', akkor ez egy új NMEA mondat kezdetét jelzi, tehát az előző mondat befejeződött. Ekkor az idáig összegyűjtött karaktereket átadjuk az NMEAParser osztálynak:

`NMEAParser.updateNMEASentence(outputStr, mcHunter);`.

A *mcHunter* átadása azért szükséges, mert csak a főprogramba van mód hibaüzenetek megjelenítésére, és kivételek az NMEA mondat feldolgozásakor is keletkezhet.

```
        public void run() {
```

```
        int input;
        try {
            connection = (SocketConnection) Connector.open(uri);
            reader = new InputStreamReader(connection.openInputStream());
            StringBuffer output;
            String outputStr;
            while ((input = reader.read()) != 13) {; }
            while (true) {
               try {
                   output = new StringBuffer();
                   while ((input = reader.read()) != 13) {
                      output.append((char) input);
                   }
                   outputStr = output.toString().substring(1, output.length() - 1);
                   synchronized (NMEAParser.lock) {
                      NMEAParser.updateNMEASentence(outputStr, mcHunter);
                   }
                   Thread.sleep(100);
               }
               catch (IOException e) {
                   mcHunter.alert( e.getMessage());
               }
               catch (InterruptedException e) {
                   mcHunter.alert(e.getMessage());
               }
               catch (Exception e) {
                   mcHunter.alert(e.getMessage());
               }
            }
        }
        catch (IOException e) {
            mcHunter.alert(e.getMessage());
        }
        finally {
            try {
               if (reader != null) {
                   reader.close();
               }
               if (connection != null) {
                   connection.close();
               }
            }
            catch (Exception e) {
               mcHunter.alert(e.getMessage());
            }
        }
    }
```

Az NMEAParser működése a következő:

- Ha az NMEA mondattípus azonosító nem a *„GPRMC”,* akkor semmit sem csinál.
- Ellenkező esetben a *StringTokenizer* osztály segítségével az NMEA mondatot tokenekre tördeljük a ’,’ karaktereknél. A tokenek tartalmát kiolvassuk, és értékeit tároljuk.

Mivel a GPS vevő 1 másodpercenként szolgáltat friss értékeket, így a mondat feldolgozása után 0,1 másodpercig a szálat altatjuk ezzel is takarékoskodunk az erőforrásokkal és spórolunk az akkumlátorral. Joggal merül fel a kérdés, hogy miért nem 1 másodpercig altatjuk a szálat, ha friss adat csak ekkor érkezik. A válasz a következő: a GPS vevő nemcsak GPRMC típusú mondatot generál a socketre másodpercenként. Ezek tartalma is feldolgozásra kerül (nem csinálunk semmit), és aztán alszunk. Ha az alvás ideje 1 másodperc lenne, akkor a GPRMC típusú mondatok feldolgozásának ciklusideje több másodperc, ami hibás működést eredményezne.

A GPS koordináták kezeléséhez még egy osztály kötődik. Ez a *MapHandler* osztály, mely a *hu.ikti.mchunter* csomag tartalmaz. A *MapHandler* szintén a *Thread* leszármazottja, és feladatát a következő:

```
public void run() {
   while (true) {
       try {
          synchronized (NMEAParser.lock) {
              if ((currentlocation = NMEAParser.getCurrentLocation()) != null) {
                     this.mc.setGPS(new             Point2D(currentlocation.getLng(),
                     currentlocation.getLat()),
                     currentlocation.getNumberOfSatellites(),
                     currentlocation.getTimestamp());
              }
              NMEAParser.reset();
          }
          Thread.sleep(1000);
       }
       catch (Exception e) {
          mcHunter.alert(e.getMessage());
       }
   }
}
```

Megvizsgálja, hogy van-e érvényes GPS koordináta. Az `NMEAParser.getCurrentLocation()` visszatérési értéke *null* ha nem létezik, egyébként pedig egy az aktuális koordinátákat tartalmazó *Location* típusú objektum. A Location osztály attribútumai egy földrajzi koordinátát és a GPS kapcsolat tulajdonságait írják le. Az Location tartalmazza ezen felül az attribútumokat kezelő *setter* illetve *getter* metódusokat is. Ha létezik aktuális koordináta, akkor ezekkel felülírja egy *MapCanvas* (a megjelenítésért felelős osztály) típusú objektum megfelelő attribútumait. Ezután meghívja az *NMEAParser reset* metódusát, azaz törli a benne tárolt értékeket. Ezt követően 1 másodperces alvási periódus következik. Itt már van értelme

az 1 másodperces alvásnak, mivel a GPS vevőtől úgysem kapunk gyakrabban friss koordinátákat.

A környezetet ábrázoló térképrészlet beszerzése

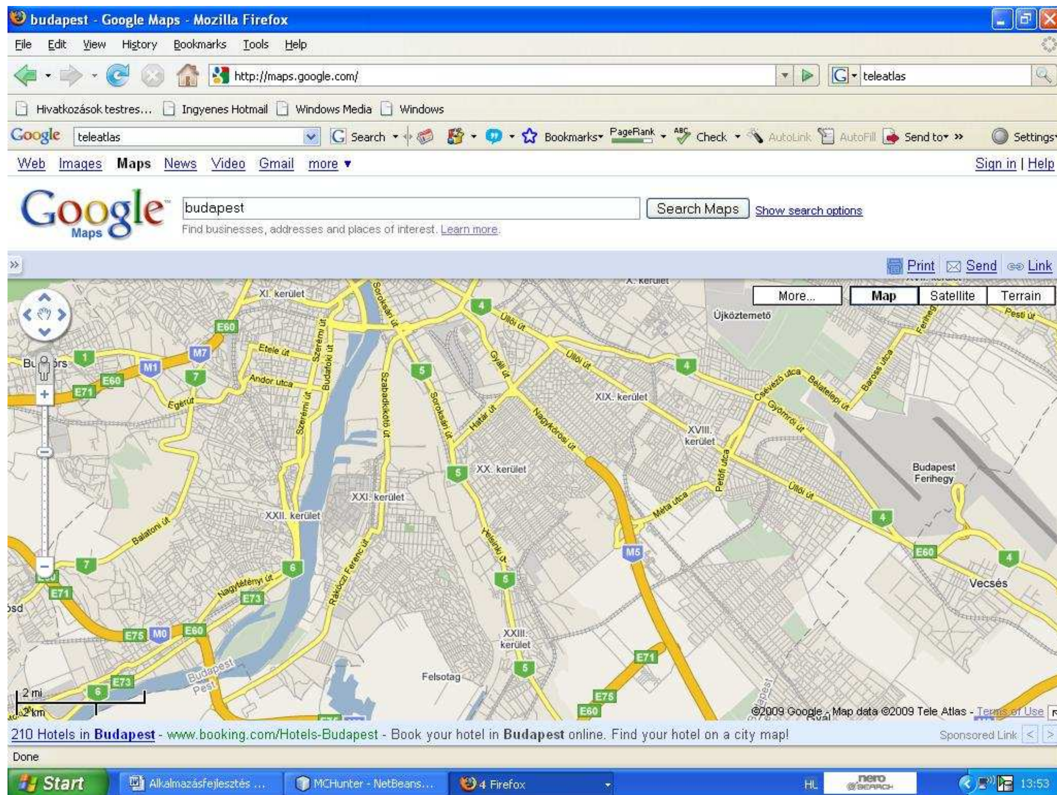
A GPS koordináták birtokában már tudjuk, hogy milyen térképrészlet van szükségünk. A megfelelő térkép beszerzésének is több módja van. Ha az eszközön rendelkezésre áll egy térképadatbázis (mint a navigátorrendszerek esetén), akkor ez is egy lehetséges alternatíva. Viszont más programok által használt térképadatbázis használata jogilag nem engedélyezett. Léteznek szabad forrású térképadatbázisok is, melyek jellemzően egy közösség által bejárt útszakaszok adataiból generálnak. Ilyen például az *OpenStreetMap* is. Jó, ha az ember ismeri ezeket, egy kisebb pontosságot igénylő projektben (igaz ez akár a McHunter-re is) ezek használhatóak. Viszont mivel nem áll mögöttük egy professzionális profitorientált csapat, ezért ezek adatainak pontossága nem garantált. Az alkalmazásunk nem használja ki egy létező térképadatbázisból származó előnyöket (például nem akarunk navigálni), viszont a szükséges térképrészletet ábrázoló kép generálásának problémája itt megoldandó feladat, ezért más megoldás használata célszerűbb.
Egy sokkal kényelmesebb megoldás, ha a *Google Maps* szolgáltatást vesszük igénybe. A *Google Maps* adatbázisát a *Tele Atlas* szállítja. *Tele Atlas*-ról a következőket érdemes tudni:

„For more than 20 years, Tele Atlas, the world's most trusted source of digital maps, …"[5]

„A Tele Atlas több mint 20 éve a világ legmegbízhatóbb térképadatbázis szállítója,…"

Ezt a tényt igazolni látszik az is, hogy a *Google* őket választotta ehhez a szolgáltatásukhoz. Ha a *Google Maps* szolgáltatására építjük a *McHunter* alkalmazást, akkor adoptáljuk ezzel a *„világ legmegbízhatóbb térképadatbázisát"* is, és nem kell elviselni egy közösség hobbiból összedobott térképadatbázis pontatlanságát. Abba a ténybe pedig, hogy a Csepel szigetnél a Kis-Dunaág hiányzik könnyű beletörődni, hisz minden nap lehet hallani a globális felmelegedés káros következményeiről.

[5] http://www.teleatlas.com/WhyTeleAtlas/TheTeleAtlasAdvantage/index.htm



6. ábra, Google Maps -2009

A *Google* ebben a szolgáltatásában bizonyos feltételek mellet ingyenesen rendelkezésre bocsájtja a megfelelő térképrészletet. A feltételek egyike, hogy fizetős szolgáltatás nem építhető rá, ami az esetünkben nem áll fent. Ha ezt az utat válasszuk, akkor elsőnek be kell szereznünk egy úgynevezett alkalmazás kulcsot (API key). Ezt a Google következő oldaláról egyszerűen beszerezhetjük: http://code.google.com/apis/maps/signup.html. Ha rendelkezünk már a megfelelő kulccsal akkor egy http protokollon keresztül megszerezhetjük a szükséges képet. Össze kell állítani egy megfelelő URI-t, melynek struktúrája a következő:

```
http://maps.google.com/staticmap?center=" + lat + "," + lng + "&format=" +
format + "&zoom=" + zoom + "&size=" + width + "x" + height + "&key=" +
apiKey;
```

Az attribútumok jelentései:

- *lat*: latitude érték, ezt az *NMEAParser*-től megkapjuk.
- *lng*: longitude érték, szintén az megkaphatjuk az NMEAParser-től.
- *format*: milyen formátumba szeretnénk megkapni a térképrészletet (pl: "png32")
- *zoom*: milyen zoomszinten legyen ábrázolva a térkép
- *width, height*: értelemszerűen a kép szélessége, magassága
- *apiKey*: a Google által szolgáltatott alkalmazás kulcs

A szükséges kép a következő módon rendelkezésünkre áll:

```
HttpConnection  hc = (HttpConnection) Connector.open(URI, Connector.READ, true);
hc.setRequestMethod(HttpConnection.GET);
InputStream  is = hc.openInputStream();
Image map = Image.createImage(is);
```

Én egy ezzel egyértékű megoldást alkalmazom, a különbség csak az, hogy a *Google* szolgáltatása helyett, a *Bay-IKTI* térképszerverétől szerzem be a képet. Ez a megoldás technikai szempontból semmibe sem tér el a fent bemutatott módszertől, a különbség csupán csak egy URI (az egyéni megoldás oka a saját térképszerver tesztelése).

A „közeli” éttermek és a térképrészlet megjelenítése

A megjelenítéssel kapcsolatos dolgokért a *MapCanvas* osztály felelős, melyet a *hu.ikti.mchunter.ui* csomag tartalmaz. Az osztály származtatja a *Canvas* osztályt és implementálja *CommandListener* interfészt. A *Canvas* osztálytól örökölt tulajdonságok a megjelenítést segítik. A Java SE-ben megszokott módon, itt is létezik a `paint` metódus, mely felülimplementálásával a megjelenítést szabályozhatjuk. Szintén a Canvas osztálytól örököljük a *keyPressed* metódust, mely segítségével egy adott gombhoz funkciót rendelhetünk. A metódus egy int típusú változót kap hívási paraméterül, mely a megnyomott gomb kódját tartalmazza. Én az 2-es, 4-es, 6-os és 8-as gombokhoz rendeltem funkciót, melynek feladata a térkép fel, balra, jobbra illetve lefele irányba való mozgatása.

```
protected void keyPressed(int keyCode) {
   switch (keyCode) {
        case KEY_NUM2:
           shiftMap("up");
           break;
        case KEY_NUM8:
           shiftMap("down");
           break;
        case KEY_NUM4:
           shiftMap("left");
           break;
        case KEY_NUM6:
           shiftMap("right");
           break;
   }
}
```

Hasonló funkciót lát el a *pointerReleased* metódus is, amelyet szintén a Canvas osztálytól örököltünk. Ez a metódus az érintőképernyő kezelésére szolgál, és szintén a futtató rendszer hívja meg.

7. ábra, A McHunter alkalmazás

Hívási paramétere két int típusú változó, melyek a képernyőn megérintett pont képernyő-koordinátarendszerbeli x és y koordinátáit adják. A képernyő-koordinátarendszer origója a képernyő bal felső sarka, az x koordináták jobb, az y koordináták pedig lefele irányba növekszenek. Azon eszközök számára (ilyen a HTC is), melyek rendelkeznek érintőképernyővel, a térkép fel, le, jobbra vagy balra irányba történő mozgatására, a megfelelő irányba mutató, ikonokat helyeztem el (7. ábra). A *pointerReleased* metódus megvizsgálja, hogy aktivizálásakor ezek a nyilakat próbáltuk-e lenyomni, és ha igen, akkor az ennek megfelelő kódot végrehajtja.

```
protected void pointerReleased(int releaseX, int releaseY) {
   int centerX = canvasWidth / 2;
   int centerY = canvasHeight / 2;

   if ((releaseX > (centerX - upImage.getWidth())) && (releaseX < ( centerX +
     upImage.getWidth()))  && (releaseY < ( upImage.getHeight() + 10 ))) {
       shiftMap("up");
   }

   if ( (releaseX > (centerX - downImage.getWidth())) && (releaseX < ( centerX +
              downImage.getWidth())) && ( releaseY >
              ( canvasHeight - downImage.getHeight()- 10))){
       shiftMap("down");
   }

   if ( ( releaseX < ( leftImage.getWidth() + 10 ) ) && ( releaseY > ( centerY -
   leftImage.getHeight())) && ( releaseY < ( centerY + leftImage.getHeight()))){
     shiftMap("left");
   }

   if ( ( releaseX > ( canvasWidth  - rightImage.getWidth() - 10) ) && ( releaseY >
              (centerY - rightImage.getHeight())) &&
              (releaseY < ( centerY + rightImage.getHeight()))){
       shiftMap("right");
   }
}
```

A *keyPressed* és a *pointerReleased* metódus egyaránt használja a *shiftMap* metódust. Ez a metódus feladata, hogy az aktuális GPS koordinátát reprezentáló *gpsLat* és *gpsLong* pontok értékét az szükséges elmozdulás irányának megfelelő lépésközzel növelje/csökkentse. Jelen esetben ez a lépésköz a zoomszint 0.0000015-szöröse. A változás megjelenítésére a *repaint* metódus szolgál, ezzel jelezhetjük a futtató rendszernek, hogy az ábrázolt kép megváltozott.

```
void shiftMap(String dir) {
   if (dir.equals("up")) {
       gpsLat = gpsLat + 0.0000015 * zoom;
   }
   else if (dir.equals("down")) {
       gpsLat = gpsLat - 0.0000015 * zoom;
   }
   else if (dir.equals("left")) {
       gpsLong = gpsLong - 0.0000015 * zoom;
   }
   else if (dir.equals("right")) {
       gpsLong = gpsLong + 0.0000015 * zoom;
   }
   mapTileSet.setCenterInMeters(Mercator.vetit(gpsLong, gpsLat));
   repaint();
}
```

Az aktuális GPS koordinátát reprezentáló *gpsLat* és *gpsLong* pontok értékét más McCanvas-beli metódus is módosíthatja. Ezel a metódussal már találkoztunk a *MapHandler* ismertetésekor. Ha az *NMEAParser*-ben létezik érvényes koordináta, akkor a kapott értékkel a MapHandler többek között a *gpsLat* és *gpsLong* tartalmát írja fölül:

```
if ((currentlocation = NMEAParser.getCurrentLocation()) != null) {
   this.mc.setGPS( …
```

Ezt a kódrészletet a MapHandler osztály tartalmazza, ahol az *mc* egy *MapCanvas* típusú objektum. Ezek után vizsgáljuk meg mit is csinál pontosan a *setGPS* metódus.

```
public void setGPS(Point2D GPS, int numberofsatelits, double timestemp) {
   hasPosition = false;
   if ((GPS.getX() == GPS.getX()) && (GPS.getY() == GPS.getY())) {
       gpsLong = floatGPS.getX();
       gpsLat = floatGPS.getY();
       hasPosition = true;
       repaint();
```

```
        }
        this.NumberOfSatellites=numberofsatelits;
        this.timestampFromGPS = timestemp;
    }
```

A metódus hívási paraméterei, egy kétdimenziós pontot reprezentáló objektum, egy *int* ami az aktuálisan látható GPS műholdak száma, illetve egy *double* ami a GPS vevőből származó idő. A kétdimenziós pont *getX* és *getY* metódusai visszaadják a pont koordinátáit, ezzel módosítjuk a *gpsLong* és *gpsLat* attribútumokat, illetve felülírjuk a műholdak számát és a GPS időt reprezentáló attribútumokat is.

A *gpsLong* és *gpsLat* attribútumok felülírásakor egy, értelmetlennek tűnő, feltételvizsgálat van.

```
if ((GPS.getX() == GPS.getX()) && (GPS.getY() == GPS.getY())){
```

Azt gondolhatnánk, hogy az itt megadott feltétel értéke mindig igaz, hiszen egy-egy objektum önmagához való hasonlóságát vizsgáljuk. A *Point2D X* és *Y* attribútumai *Double* típusú objektumok, amelyeknek értékei vagy egy adott valós szám, vagy *NaN* (a NaN-t a Double osztály definiál). Ezt a *NaN* való egyezést vizsgáljuk, ugyanis a *NaN*-ról azt kell tudnunk, hogy semmivel sem egyenlő, még önmagával sem.

A *MapCanvas* tartalmaz továbbá egy *Command* objektumot melynek feladat a programból való kilépés.

```
Command exit = new Command("Exit", Command.EXIT, 0);
this.addCommand(exit);
```

Az *exit* parancs figyelésért a *MapCanves* objektum felelős. Hogy ezt a feladatot ellássa, implementálja a *CommandListener* interfészt, így tartalmaznia kell a *commandAction* metódust is. Ha a *commandAction* az *exit* paranccsal van meghívva, akkor a vezérlés a *mcHunter* objektum (ami a főprogram) *exitMIDlet* metódusára kerül át a vezérlés és a program befejezi működését.

```
public void commandAction(Command c, Displayable d) {
    if(c == exit){
        this.mcHunter.exitMIDlet();
    }

}
```

A *MapCanvas* működése a következő:

Példányosításkor beállítódnak a megfelelő attribútum-értékek. Ezen folyamat során töltődnek be a térkép mozgatásához szükséges grafikus nyilak és a középpontot jelző célkereszt is.

```
leftImage = Image.createImage("/left48.png");
rightImage = Image.createImage("/right48.png");
upImage = Image.createImage("/up48.png");
downImage = Image.createImage("/down48.png");
crosshairsImage = Image.createImage("/redcrosshairs.png");
```

Már megvan a környezetet reprezentáló térkép, amit kirajzolunk. Ezután kigyűjtjük a „közeli" éttermek listáját, a *pickUpNearRestaurants* metódus segítségével.

```
private void pickUpNearRestaurants(){
   Restaurant restaurant;
   double maxDistanceX = canvasWidth / 2;
   double maxDistanceY = canvasHeight / 2;
   Point2D center = getCenter();

   this.nearRestaurants.removeAllElements();
   for(int i = 0; i < this.restaurants.size(); i ++ ){
        restaurant = (Restaurant) this.restaurants.elementAt(i);
        if( (restaurant.getX() - center.getX()) > maxDistanceX ){
           break;
        }
        if( ( ( center.getX() - restaurant.getX() ) < maxDistanceX) &&
                ( ( center.getY() - restaurant.getY() ) < maxDistanceY) &&
                ( ( restaurant.getY() - center.getY() ) < maxDistanceY)){
           this.nearRestaurants.addElement(restaurant);
        }
   }
}
```

A *pickUpNearRestaurants* metódus működése az alábbi:

Beállítjuk a középpontot és azon távolságértékeket, amelyeknél már nem látszik a képernyőn az adott étterem. Kiürítjük a közeli éttermek listáját majd egy ciklus keretében a *restaurants* (a „Letöltött POI adatok feldolgozása" fejezetéből már ismerjük) tartalmát bejárjuk. Az aktuális éttermek koordinátáit megvizsgáljuk, és ha az „közelinek" minősül, akkor felvesszük a közeli éttermek listájába. A közeliség vizsgálat két részben történik. Ezt azért választottam ketté, mert ha `(restaurant.getX() - center.getX()) > maxDistanceX )` feltétel teljesül, akkor felesleges bejárni a *restaurants* hátralévő részét, már közeli éttermet úgysem találunk (a *restaurants* ugyanis az éttermeket X koordináta szerint növekvő sorrendbe tartalmazza).

Ha megvannak a közeli éttermek, akkor már csak meg kell őket jeleníteni. Ez a *paintIcons* metódus feladata. A metódus itt is a középpont lekérésével kezdődik. Ezt követően a *Canvas* megfelelő helyére illesztjük a térkép mozgatásához használt nyilakat, és a középpontot jelző célkeresztet. Ehhez a *drawImage* metódust használjuk. A *drawImage*-nek meg kell adni az illesztendő képet, azt, hogy a *Canvas* melyik x/y koordinátájára szeretnénk az illesztést elvégezni, és erre az x/y koordinátákra melyik képbeli pixel illeszkedik. Ezután kiíratjuk a GPS koordinátákat, a GPS időt, és a látható műholdak számát. A feladatot a *drawString* metódus végzi el, melynek szemantikája a *drawImage*-ével azonos. Végül bejárjuk a közeli éttermek listáját egy ciklus keretében. Kiszámoljuk az étterem megjelenítési helyét a középponthoz viszonyítva, és kirajzoljuk. A megjelenítendő képet maga a Restaurant típusú objektum szolgáltatja.

```
public void paintIcons(Graphics g){
     Restaurant restaurant;
     Point2D center = getCenter();
     int distanceX, distanceY;

     g.drawImage(leftImage, 3, canvasHeight / 2, Graphics.VCENTER | Graphics.LEFT);
     g.drawImage(rightImage, canvasWidth - 3, canvasHeight / 2, Graphics.VCENTER |
             Graphics.RIGHT);
     g.drawImage(upImage, canvasWidth / 2, 3, Graphics.HCENTER | Graphics.TOP);
     g.drawImage(downImage, canvasWidth / 2, canvasHeight - 3, Graphics.HCENTER |
             Graphics.BOTTOM);
     g.drawImage(crosshairsImage, canvasWidth /2, canvasHeight / 2, Graphics.VCENTER |
             Graphics.HCENTER);

     g.setColor(0, 0, 0);
     g.drawString("GPS-koordinák (Lat,Long): " + this.gpsLat + "," + this.gpsLong, 0,
             canvasHeight -30, Graphics.BOTTOM | Graphics.LEFT);
     g.drawString("GPS-timestemp: " + this.timestampFromGPS, 0, canvasHeight -20,
             Graphics.BOTTOM | Graphics.LEFT);
     g.drawString("Number of satellites: " + this.NumberOfSatellites , 0, canvasHeight -10,
             Graphics.BOTTOM | Graphics.LEFT);

     for (int i = 0; i < this.nearRestaurants.size(); i++){
         restaurant =  (Restaurant) this. nearRestaurants.elementAt(i);
         distanceX = (int)(restaurant.getX()) - (int)(center.getX());
         distanceY       =       -(int)(restaurant.getY()       +       (int)(center.getY());
         g.drawImage(restaurant.getImage(),  centerX + distanceX,  (centerY + distanceY),
             Graphics.VCENTER | Graphics.HCENTER);
     }
 }
```

A „főprogram”

Mint azt a MIDlet ismertetésénél bemutattam, minden MIDlet alkalmazásnak tartalmaznia kell egy *javax.microedition.midlet.MIDlet* osztály leszármazottját. Jelen esetünkben ez a *McHunter* osztály, melyet *a hu.ikti.mchunter* csomag tartalmaz. *MIDlet*

leszármazottként, implementálnia kell a *startApp* metódust (az alkalmazás belépési pontját) és az alkalmazás életciklusát megvalósító metódusokat, azaz a *pauseApp, destroyApp, resumeMIDlet* metódusokat is. Ezen felül implementálja a futás közbeni hibaüzenetek megjelenítéséhez egy *switchDisplayable* metódust is:

```
public void switchDisplayable(Alert alert, Displayable nextDisplayable){
     Display display = getDisplay();
     if (alert == null) {
         display.setCurrent(nextDisplayable);
     }
     else {
         display.setCurrent(alert, nextDisplayable);
     }
 }
```

A *McHunter* a következő attribútumokkal rendelkezik:

- *(GPSConnectionViaSocket) gpsconnectionviasocket*: ez az objektum a friss GPS adatok beszerzéséért felelős
- *(MapHandler) maphandler*: felelősége, hogy ha van érvényes GPS koordináta azt átadja a MapCanvas részére
- *(MapCanvas) mc*: a grafikus felhasználó felületet szolgáltatja

A főprogram működése és ezzel együtt a teljes program működése az alábbi:

Indulásakor inicializálódnak az attribútumok:

```
this.mc = MapCanvas.getInstance();
this.mc.setMCHunter(this);

this.maphandler = MapHandler.getInstance();
this.maphandler.setMCHunter(this);
this.maphandler.setMapCanvas(mc);

this.gpsconnectionviasocket = new GPSConnectionViaSocket(this);
this.gpsconnectionviasocket.setMCHunter(this);
```

Szinte mindegyik osztálynak létezik *setMCHunter* metódusa, ez a hibaüzenetek megjelenítéséhez szükséges.

Inicializálás után letöltjük az éttermek POI adatait, és feldolgozzuk azt a MyXMLParsers segítségével:

```
String xml = HTTPConnection.getViaHttpConnection("http://91.83.41.226/2/McDonald.kml");
MyXMLParsers mp = new MyXMLParsers(xml, this);
```

A *MyXMLParsers* létrehozza a *Restaurans* egyetlen példányát és feltölti azt az éttermek adatival. A *MyXMLParsers* szintén megkapja a *McHunter* példányát, hogy szükség esetén a hibaüzenetek megjeleníthetőek legyenek. Ezek után a *Thread* leszármazottjait (*GPSConnectionViaSocket, MapCanvas és MapHandler*) elindítjuk, és a megjelenített objektumnak a *MapCanvas* példányát állítjuk be.

```
this.gpsconnectionviasocket.connectToDevice( "127.0.0.1", 20175);
this.mc.start();
this.maphandler.start();
this.switchDisplayable(null, mc);
```

Ezután a program úgymond önműködő, ezek a példányok megvalósítják a program teljes működését. A *gpsconnectionviasocket* folyamatosan próbálja beolvasni a friss GPS koordinátákat, és ha talál, akkor az *NMEAParsers* felhasználásával feldolgozza és tárolja. A *maphandler* folyamatosan ellenőrzi, hogy az *NMEAParsers* tartalmaz-e érvényes GPS koordinátát, és ha igen, akkor annak értékét átadja az *mc*-nek. Az *mc* letölti a szükséges térképrészletet melynek középpontja a kapott GPS koordináta. Emellett figyeli a felhasználói eseményeket is. Ilyen lehet a térkép léptetése, illetve programból való kilépés. Ha *exit* parancsot tapasztal, akkor a vezérlés visszakerül a főprogram (a McHunter példánya) *exitMIDlet* metódusára, ahol ezek a szálak leállításra kerülnek és a program terminál.

Összegzés

A kezdetben kikötött kritériumok túl szorosnak bizonyultak. A legjobb tudásom szerint sem sikerült egy többféle platformon működő kódot alkotni ingyenes komponensek felhasználásával. A Windows Mobile esetében lehetőség van olyan KVM alkalmazására, mely képes támogatni a *Location API*-t (ilyen a már említett *IBM J9*, vagy a *NSIcom CrE-ME*, de pénzért az *Esmertec* cég is képes ilyet szállítani), de mivel ezek licensz-díjasok ezért ezek használata ütközik az előre lefektetett kritériummal. A GPS vevő adatait, egy #C nyelven írt program segítségével szerezzük be, viszont ez a #C program az S60 platformon nem futtatható. Sőt, az S60 platformon az integrált GPS vevőhöz való kapcsolódáshoz szükséges beállítási paramétereket sem sikerült beszereznem (működik a *Location API,* ezért ilyen célú felhasználáshoz kizárólag ezt ajánlják).

Idáig nem említettem, napjaink egyik favoritjaként emlegetett mobil platformot, az *Android*ot. Ezt a platformot egy igen erős konzorcium fejleszti, melynek vezetője a *Google*. Előzetes hírek szerint, ez a platform számos napjainkban használatos platform hibáját kiküszöböli, ezért a szakemberek gyors elterjedésére számolnak. Az *Android* is biztosít Java támogatást, de ez inkább CDC kategóriájú eszközök platformja lesz, így a *McHunter* alkalmazás *Android* platformon történő futtatása további fejlesztést igényel.

Mindent összegezve, a Java nyelv mobil platformon történő futtatása esetén valóban teljesül a platformfüggetlenség, hisz nem adnak ki új eszközt, hogy valamilyen Java támogatást nem biztosítanak rajta. Ezt a sikert csorbítja viszont, hogy ezt egy újabb függőség bevezetése árán sikerült elérni, megjelent a JVM függőség. Remélhetőleg e téren pozitív változások lesznek a jövőben, és ekkor majd elmondható lesz, hogy a Java kódok futtatása minden függőségtől mentes lesz különféle mobil platformokon.

,Remélem, hogy a dolgozat témájaként kitűzött célt sikerrel teljesítettem, és sikerült megismertetni az olvasóval a legkisebb Java kiadás elméleti alapjait, és annak gyakorlati használatát a *McHunter* alkalmazás bemutatásán keresztül.

Felhasznált irodalom:

[1] Nyékiné G. Judit (2000) JAVA2 – Útikalauz programozóknak

[2] Dr Gordos Géza, Dr Laborczi Péter Ambiens intelligencia alkalmazások

- http://www.matud.iif.hu/07jul/11.html

[3] Connected Limited Device Configuration (CLDC)

- http://java.sun.com/products/cldc/overview.html

[4] Foundation Profile

- http://java.sun.com/products/foundation/overview.html

[5] Information Module Profile

- http://java.sun.com/products/imp/

[6] J2ME Building Blocks for Mobile Devices

- http://java.sun.com/products/cldc/wp/KVMwp.pdf

[7] Mobile Information Device Profile

-http://java.sun.com/products/midp/

[8] National Marine Electronics Association

- http://www.nmea.org/

[9] Personal Profile

- http://java.sun.com/products/personalprofile/index.jsp

[10] Java Community Process

- http://jcp.org/en/jsr/detail?id=217