SZAKDOLGOZAT

Erdei József

Debrecen

2011

Debreceni Egyetem
Informatikai Kar

Állapottér-reprezentációk és implementációjuk mobiltelefonos alkalmazásokban

Témavezető:
Dr. Kósa Márk Szabolcs
egyetemi tanársegéd

Készítette:
Erdei József
programtervező informatikus

Debrecen
2011

Tartalomjegyzék

1. Bevezetés és célkitűzések

Napjainkra a mobiltelefonok igen nagymértékben elterjedtek. Alig van már olyan személy, akinek ne lenne legalább egy mobilja. A mobilok ugyanis most már nem csak telefonálásra használhatók, hanem rengeteg hasznos és szórakoztató funkcióval is fel vannak szerelve. És a mobilok legnagyobb előnyeként, hogy kicsik és hordozhatók, bármikor, például utazás közben is hozzáférhetünk ezekhez a funkciókhoz. Mivel a mobiltelefonok világán kívül a programozás is érdekel, ezért ebben a szakdolgozatban két, Java nyelven megírt játékról lesz szó. Az pedig, hogy mobilra történik a fejlesztés, tovább nehezíti a feladatot, mivel a Java Micro Edition-ben sokkal kevesebb lehetőség áll a rendelkezésünkre, mint például a Standard Edition-ben. Ahogyan a címben is olvasható, a mesterséges intelligencia technikáit alkalmazó játékokról fog szólni a szakdolgozat, azaz valamilyen tudással lesznek az alkalmazások ellátva és így olyan, mintha igazi ellenfelekkel játszanánk, amely tovább fokozza a játékélményt.

A szakdolgozat első felében a Java nyelvet, annak kialakulását, jellemzőit és technológiáit mutatom be. Ezt követően a Micro Edition technológiáról, annak konfigurációiról és profiljairól írok.

A dolgozat második felében két játék állapottér-reprezentációját, és J2ME-ben történő megvalósítását írom le. Az egyik játék a tilitoli, a másik a tic-tac-toe. Előbbi egy képkirakós, egyszemélyes, utóbbi pedig egy „jelgyűjtögetős”, kétszemélyes játék. A tic-tac-toe ugyanis az amőba kisebb méretű táblán játszódó változata. Részletesebben majd az 5.2., illetve az 5.3. fejezetben tárgyalom őket.

2. A Java mint programozási nyelv

A Java egy számítógép programozási nyelv, amely lehetővé teszi a programozók számára, hogy angol nyelv alapú utasításokat használjanak a numerikus kódok helyett. Ez úgynevezett magas szintű nyelv, mivel az utasítások nyelve ember közelibb. Ahogyan az angol nyelvnek, a Java nyelvnek is vannak szabályai, amely meghatározza, miképpen kell használnunk az utasításokat. Ezeket a szabályokat összefoglalóan szintaktikának nevezzük. A magas szintű utasításokkal megírt program numerikus kódra fordítódik le, amelyet a számítógép képes értelmezni és futtatni.

A Java nyelvet egy James Gosling által vezetett csapat fejlesztette ki a Sun Microsystems számára az 90-es évek elején. A nyelvet először Oak-nak nevezte el, az irodája előtt lévő tölgyfa után, de mivel ez a név már foglalt volt, ezért lett a nyelv neve Java. A Java eredetileg digitális mobil készülékeken való használatra lett tervezve, mint például mobiltelefonokra. Azonban amikor a Java 1.0-t 1996-ban nyilvánosságra hozták, az az interneten való használatot célozta meg. Több interaktivitást nyújtott a felhasználóknak azáltal, hogy a fejlesztők animált weboldalak gyártásába kezdtek. Az évek során egy sikeres nyelvvé fejlődött mind interneten, mind egyéb más területen használva. Egy évtizeddel később még mindig rendkívül népszerű nyelv maradt, amit világszerte több mint 6,5 millió fejlesztő használ.

A nyelv megalkotásakor az alábbi alapelveket vették figyelembe:

- *Egyszerű használat*

  A Java alapjai a C++ programozási nyelvből jöttek. Annak ellenére, hogy egy igen hatékony nyelv, túl bonyolultnak érezték a szintaxisát, és nem megfelelőnek minden Java követelmény számára. A Java a C++-ra épült és annak ötleteit fejlesztette tovább, aminek hatására egy nagy teljesítményű és egyszerűen használható programozási nyelvvé vált.

- *Megbízhatóság*

  A Java szükségesnek érezte, hogy a végzetes programozási hibák valószínűségét csökkentse. Mindezt szem előtt tartva bevezették az objektumorientált programozást. Miután az adatok és azok manipulációja egy helyre vannak csomagolva, megnövekedett a Java robusztussága.

- *Biztonság*

  Mivel eredetileg a Javát mobil eszközökre fejlesztették ki, amely a hálózatokon az adatcserét valósít meg, ezért magas szintű biztonság lett beleépítve. A Java nagy valószínűséggel napjaink egyik legbiztonságosabb programozási nyelve.

- *Platform függetlenség*

  A programoktól elvárás az, hogy működjenek bárhol, függetlenül attól, hogy milyen gépen futtatjuk őket. A Java ezért hordozható nyelvként lett megírva, hogy ne függjön egy-egy számítógép operációs rendszerétől, illetve hardverétől.

A Sun Microsystems csapata sikeresen kombinálta ezeket az alapelveket, így a Java népszerűsége a robusztusságra, biztonságra, könnyű használhatóságra és a hordozhatóságra vezethető vissza.

3. Java technológiák

3.1. Java EE

Java Platform, Enterprise Edition 6 egy ipari szabvány vállalati Java számításokhoz. Használhatjuk a Java EE 6 Web Profile-t arra, hogy újabb generációs webes alkalmazásokat készítsünk és teljesen átvegyük a hatalmat a vállalati alkalmazások készítése felett. A fejlesztők termelékenységi előnyökhöz juthatnak az egyre több annotáció, POJO, az egyszerűbb csomagolások és kevesebb XML konfigurációk használatával.

3.2. Java SE

Java Platform, Standard Edition. Java alkalmazások fejlesztését és telepítését teszi lehetővé asztali számítógépekre és szerverekre, valamint napjainkban egyre inkább használt beágyazott és valós idejű rendszerekbe.

3.3. Java ME

Java Platform, Micro Edition. Robusztus, rugalmas környezetet biztosít alkalmazások futtatására mobil és más beágyazott eszközökön, mint például mobil telefonokon, PDA-kon, TV set-top boxokon, nyomtatókon. A Java ME rugalmas felhasználói interfészt, robusztus biztonságot, beépített hálózati protokollokat tartalmaz, és támogatja a hálózatos és offline alkalmazásokat, amelyek dinamikusan letölthetők.

A három technológia összehasonlítása:

4. Java ME

4.1. Konfigurációk

A J2ME platform különféle eszközök széles skáláját fedi le. Egyes eszközök, mint például TV set-top boxok, Internet TV-k és felső kategóriás kommunikátorok széleskörű felhasználói interfész képességekkel és 2-től 16 MB-ig terjedő memóriával rendelkeznek.

Vannak azonban egyszerűbb készülékek is, pl. mobil telefonok, lapolvasók, amelyek egyszerűbb felhasználói felülettel és alacsonyabb memóriával bírnak. Emiatt, a Sun kettéosztotta a Java 2 Micro Edition-t konfigurációkra. Ezen konfigurációk virtuális gépek tulajdonságait, Java nyelvek jellemzőit és a Java API osztályait definiálják minden egyes konfigurációs környezet számára.

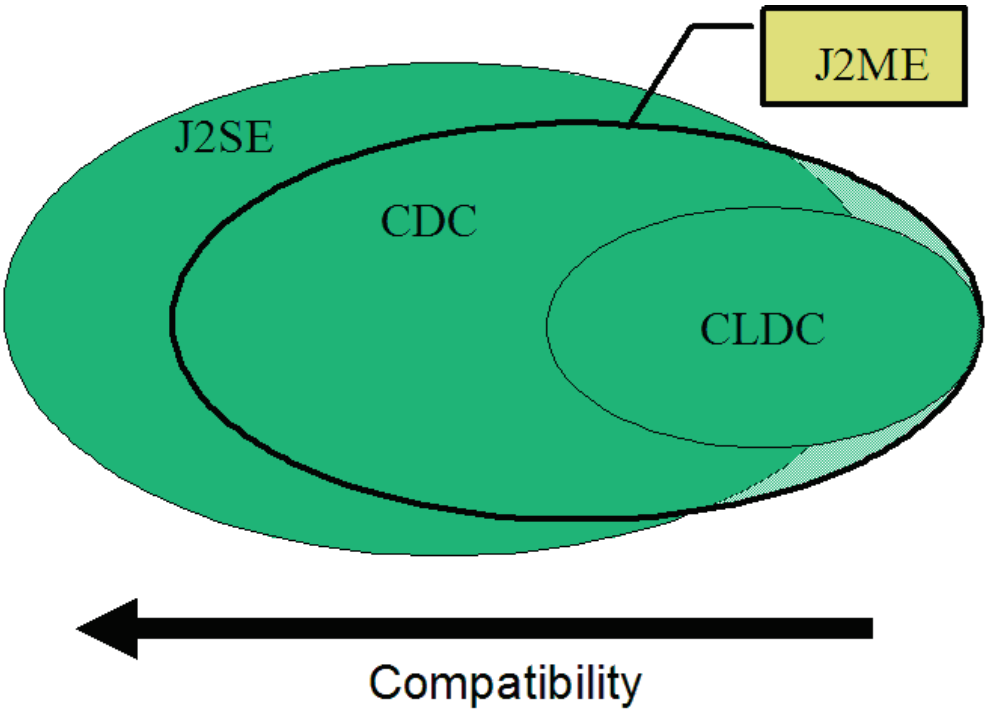
4.1.1. CDC – Connected Device Configuration

- Felhasználói interfész lehetőségeinek széles skálája (egészen addig, amikor már nincs is felhasználói felület)
- 2-től 16 MB-ig terjedő memória
- 16, illetve 32 bites processzor
- Csatlakoztathatóság más típusú hálózatokhoz
- Példák: TV set-top boxok, Internet TV-k, felső kategóriás kommunikátorok

4.1.2. CLDC – Connected Limited Device Configuration

- Nagyon egyszerű felhasználói interfész (beleértve a felhasználói felület hiányát is)
- Kis memória igény
- 16, illetve 32 bites processzor
- Vezeték nélküli kommunikáció, minimális sávszélesség
- Korlátozott energiafelhasználás, főként akkumulátorról működés
- Példák: mobil telefonok, lapolvasók, határidőnaplók

A kapcsolatot a CDC és a CLDC között az alábbi ábra mutatja:



4.2. Profilok

4.2.1. MIDP – Mobile Information Device Profile

A CLDC-re épülve, ez a profil egy szabványos platformot biztosít a kisméretű, erőforrás-korlátozott, vezeték nélküli mobil információs eszközök, mint pl. mobiltelefonok és kétirányú lapolvasók számára. Ráadásul amiatt, hogy ezek az eszközök vezeték nélkül is működnek, ezért kis kijelzővel, korlátozott adatbevitellel, tárhellyel, akkumulátor élettartammal és CPU-val rendelkeznek. Ezek alapján az eszközök két típusát különböztetjük meg. Egyik csoport a mobil információs készülékek, a másik pedig a hang alapú kommunikációs eszközök csoportja.

4.2.2. Foundation Profile

A Foundation Profile két célt szolgál. Egyrészt azoknak a Java 2 Platform képes eszközöknek biztosít egy profilt, amelyeknek szükségük van egy hálózatot is elérő Java környezetre, de nem igényelnek grafikus felhasználói felületet. Másrészt egy alap profilt nyújt, amelyhez más profilokat tudunk integrálni, hogyha más funkcionalitásokra van szükségünk, mint például grafikus felhasználói felületre vagy egyéb funkciókra.

4.2.3. Personal Profile

A Personal Profile a Java API-nak egy olyan halmaza, amely grafikus felhasználói interfész eszköztárral támogatja az erőforrás korlátozott eszközöket. A CDC-vel együtt, egy teljes körű J2ME alkalmazás környezetet biztosít egyedi termékek és beágyazott eszközök számára.

Főbb tulajdonságok:

- Teljes AWT kompatibilitás
- Applet alkalmazás programozási modell támogatás
- Migrációs útvonal régebbi technológiák számára, mint pl. PersonalJava

4.2.4. Egyéb profilok

- RMI Profile (Foundation Profilera és a CDC-re épülve)
  Ez a profil együttműködést biztosít a J2SE RMI-vel.
- PDA Profile (a CLDC-re épülve)

4.3. Virtuális gépek

4.3.1. KVM

A KVM egy virtuális gép, amelyet kisméretű, erőforrás korlátozott eszközökre fejlesztettek ki, mint például mobil telefonokra, lapolvasókra. A KVM implementálja a CLDC-t. Azáltal, hogy a KVM C nyelven lett megírva, ezért az könnyen hordozható más platformokra is. Mivel a KVM a Suntól ered, ezért ez egy standard/referencia implementáció J2ME-re, amely egyrészt elérhető Win32-re, Solarisra stb., bár a teljesítménye nem valami jó (pl. a Jbeddel összevetve).

4.3.2. Jbed

A Jbed egy kis és gyors Java virtuális gép, amelyet a svájci Esmertec cég fejlesztett ki beágyazott eszközökre, mint például PDA-kra vagy okostelefonokra. A legtöbb JVM-mel ellentétben, a JbedVM CLDC mindig az eszközön fordítja le a bájtkódot natív kódra. Így sokkal gyorsabb, mint az interpretált JVM, mivel nagyon kicsi marad. Ahogyan a neve is jelzi, a Jbed a CLDC-t is implementálja.

4.3.3. Other VM

Vannak még egyéb virtuális gépek is J2ME-re különböző jellemzőkkel és specializációkkal.

- Pl. Wabasoft: Waba-VM
- IBM: J9

5. Alkalmazásfejlesztés

5.1. J2ME alkalmazás fejlesztése során figyelembe veendő szempontok

5.1.1. Tervezés kisméretű eszközökre

- Legyen egyszerű. Távolítsuk el a felesleges funkciókat. Lehet, hogy azok egy másik alkalmazáshoz tartoznak.
- A kisebb jobb. A kisebb alkalmazások kevesebb memóriát használnak és rövidebb idő alatt telepíthetők. Célszerű a Java alkalmazásokat Java Archive (JAR) fájlokba tömöríteni.
- Minimalizáljuk a futásidejű memória használatát. Ahhoz, hogy csökkentsük a futásidejű memória használatát használjunk inkább skalár típusokat objektum típusok helyett. Ne függjünk a szemétgyűjtögetőtől. Ha valamire már nincs szükségünk, akkor ne a szemétgyűjtögetőre várjunk, hanem explicit módon szüntessük meg. Egy másik módja a futásidő csökkentésének az lehet még, hogy újrafelhasználunk objektumokat, illetve kerüljük a kivételeket.

5.1.2. Tervezés mobil eszközökre

- Végeztessük a szerverrel a munka nagy részét. A számításigényes feladatokat mozgassuk a szerverre és hagyjuk, hogy az végezze el őket. A mobil eszközre pedig az interfészt és a minimális számítási igényű feladatokat bízzuk.
- Ügyeljünk a megfelelő nyelv kiválasztásra. A J2ME még mindig gyerekcipőben jár és lehet nem is tartalmazza a számunkra megfelelő opciókat. Más objektumorientált nyelvek, mint például a C++, jobb választás lehet az igényeink teljesítésére.

5.1.3. Teljesítményre vonatkozó javaslatok

- Használjunk lokális változókat. A lokális változók gyorsabban érhetőek el, mint az osztályszintű tagok.
- Kerüljük a sztring összefűzéseket. A sztring konkatenációk ugyanis csökkentik a teljesítményt és növelik a maximális memória használatot.
- Szinkronizáció helyett használjunk szálakat. Minden egyes művelet, amely meghaladja a másodperc 1/10-ed részét, külön szálat igényel. A szinkronizáció mellőzése a teljesítményt is növelheti.

- Használjuk a Model-View-Control architektúrát. Az MVC szétválasztja a modellt, a vezérlést és a megjelenítést, amely nemcsak teljesítményi, hanem későbbi bővítés céljából is megkönnyítheti a munkánkat.

5.1.4. Fordítási javaslatok

- Mint bármely más Java alkalmazást, ezt is lefordíthatjuk becsomagolás és telepítés előtt. A J2ME-ben azonban használnunk kell a J2SE fordítót a megfelelő beállításokkal.
- Legfőképpen a *–bootclasspath* használatára van szükségünk, amely azt jelzi a fordítónak, hogy nem J2SE, hanem J2ME osztályokat szeretnénk fordítani.
- Ne helyezzük a konfigurációs osztályokat a fordító *CLASSPATH*-jába. Ez ugyanis futásidejű hibát eredményezhet, mivel a fordító először automatikusan a J2SE alapvető osztályaiban keres, függetlenül attól, hogy mi van a *CLASSPATH*-ban. Vagyis a fordító nem fog tudni semmilyen más osztályra vagy metódusra hivatkozni, ami hiányzik az alapvető J2ME konfigurációból, és futás idejű hibát fog jelezni, amikor megpróbálja futtatni az alkalmazást.

5.1.5. Becsomagolási és telepítési javaslatok

- Mivel a J2ME korlátozott memóriával rendelkező kisméretű eszközökre lett tervezve, a szokásos Java preverifikáció nagy részét eltávolították a virtuális gépből, hogy kisebb helyigényt tegyenek lehetővé. Ennek eredményeként a telepítés előtt szükséges preverifikálni a J2ME alkalmazásunkat. Majd egy további ellenőrzést végeznek futási időben, hogy meggyőződjenek arról nem változott-e meg az osztály a preverifikáció óta.
- Pontosan hogyan kell elvégeznünk a preverifikációt vagy az osztályok helyességének ellenőrzését, az az eszközkészlettől függ. A CLDC rendelkezik egy preverify nevű parancssoros segédprogrammal, amely elvégzi az aktuális verifikációt és plusz információt szúr be erről az osztály fájlokba. A MIDP a vezeték nélküli eszközkészletet használja, ami tartalmaz egy grafikus eszközt, bár parancssorból is lehet futtatni.

5.2. Tilitoli

5.2.1. A játék leírása, szabályai

A tilitoli napjainkban is egy igen kedvelt logikai játék. Az eredeti játék úgy néz ki, hogy egy képet felosztanak egy (N×N)-es mátrixra, és ennek a mátrixnak az egyik elemét kiveszik. Ez a szabad hely teszi lehetővé, hogy a többi képrészeket ide-oda tologassuk – innen ered a játék neve is –, ameddig ki nem sikerül rakni az összekevert képrészletekből a kép eredeti állapotát. Ezt a klasszikus (N×N)-es vagy (M×N)-es felosztást azonban tovább is lehet nehezíteni azáltal, hogy például nem egyenlő méretű vagy formájú alakzatokra tördeljük a képet.

Napjainkban, mint sok egyéb játéknak, ennek is elkészítették már az elektronikus változatát. Szakdolgozatom első része is egy ilyen játékot fog bemutatni mobiltelefonra fejlesztve, bár az egyszerűbb (N×N)-es felosztással.

5.2.2. Állapottér reprezentáció

Adott N^2 db számozott, négyzet alakú lapocska, amelyek egy táblán (N×N)-es sémában helyezkednek el. Az N értékét a felhasználó adja meg. Az N nem lehet kisebb 3-nál és pozitív egésznek kell lennie. A táblán a nullás értékű mező szimbolizálja az üres lapocskát. Erre a helyre tologathatjuk a szomszédos lapocskákat. A feladat az, hogy adott kezdőállásból kiindulva célállásnak megfelelő elrendezést alakítsunk ki.

A probléma világa: a tábla a számozott lapocskákkal.

A világ leírása: a tábla egyes pozícióin épp mely számozott lapocskák vannak.

Egy-egy pozícióra hivatkozás: (sor, oszlop).

pozíció	(1,1)	(1,2)	…	(n,n)
lapocska	$l_{1,1}$	$l_{1,2}$	...	$l_{n,n}$

ahol $0 \leq l_{i,j} \leq n^2-1$ minden $1 \leq i, j \leq n$ esetén, ahol $l_{i,j}$ az (i,j) pozíció értéke és az n 3-nál nem kisebb természetes szám.

Tehát $H_{i,j} = H = \{0, 1, \ldots, n^2-1\}$ minden $1 \leq i, j \leq$ n-re.

A probléma világának egy-egy állapotát egy-egy olyan

$$\begin{pmatrix} l_{1,1}\, l_{1,2} \ldots l_{1,n} \\ l_{2,1}\, l_{2,2} \ldots l_{2,n} \\ \ldots\ldots \\ l_{n,1}\, l_{n,2} \ldots l_{n,n} \end{pmatrix}$$

érték (n×n)-es határozza meg, amelyben az értékek *H*-beli elemek, és az érték (n×n)-esben minden egyes érték *H*-ból pontosan egyszer fordul elő, vagyis:

$$\textit{kényszerfeltétel}(l) = \forall i \forall j \forall s \forall o(\neg(i = s) \vee \neg (j = o) \supset \neg (l_{i,j} = l_{s,o}))$$

Kezdőállapot:

$\textit{kezdő} \in \mathcal{H}$, beállítása a lapocskák összekeverésével történik.

Célállapot: $c \in \mathcal{C}$

Az az állapot, amikor a táblán lévő elemek a bal-felső saroktól indulva 0-tól kezdve, növekvően helyezkednek el, tehát:

$$c = \begin{pmatrix} 0 & 1 & 2 & \ldots & n\text{-}1 \\ n & \ldots & & & . \\ . & & & & . \\ . & & & & . \\ n^2\text{-}n & \ldots & & & n^2\text{-}1 \end{pmatrix}$$

Operátorok halmaza:

$\mathcal{O}$ = {*mozgat(irány)*}, ahol *irány* ∈ {*fel, le, jobbra, balra*}.

Operátor alkalmazási előfeltételek:

- A *mozgat(fel)* operátor alkalmazható az $l \in \mathcal{H}$ állapotra, ha teljesül a következő operátoralkalmazási előfeltétel:
  - Az üres mező (azaz a 0-ás értékű) nem az utolsó sorban van

$$\neg\exists i \neg\exists j((l_{i,j} = 0) \wedge (i = n))$$

- A *mozgat(le)* operátor alkalmazható az $l \in \mathcal{H}$ állapotra, ha teljesül a következő operátoralkalmazási előfeltétel:
  - Az üres mező (azaz a 0-ás értékű) nem az első sorban van

$$\neg\exists i \neg\exists j((l_{i,j} = 0) \wedge (i = 1))$$

- A *mozgat(jobbra)* operátor alkalmazható az $l \in \mathcal{H}$ állapotra, ha teljesül a következő operátoralkalmazási előfeltétel:
  - Az üres mező (azaz a 0-ás értékű) nem az első oszlopban van

$$\neg\exists i \neg\exists j((l_{i,j} = 0) \wedge (j = 1))$$

- A *mozgat(balra)* operátor alkalmazható az $l \in \mathcal{H}$ állapotra, ha teljesül a következő operátoralkalmazási előfeltétel:
  - Az üres mező (azaz a 0-ás értékű) nem az utolsó oszlopban van

$$\neg\exists i \neg\exists j((l_{i,j} = 0) \wedge (j = n))$$

Operátor alkalmazásának hatása:

- A *mozgat(fel)* operátort az $l \in \mathcal{H}$ állapotra alkalmazva a következőképpen definiált $l' \in \mathcal{H}$ állapotot kapjuk:

  ha az éppen $l_{s,o} = 0$

$$l'_{i,j} = \begin{cases} 0 & \text{ha } i=s+1 \wedge j=o \\ l_{s+1,o} & \text{ha } i=s \wedge j=o \\ l_{i,j} & \text{egyébként} \end{cases}$$

- A *mozgat(le)* operátort az $l \in \mathcal{H}$ állapotra alkalmazva a következőképpen definiált $l' \in \mathcal{H}$ állapotot kapjuk:

  ha az éppen $l_{s,o} = 0$

$$l'_{i,j} = \begin{cases} 0 & \text{ha } i=s-1 \wedge j=o \\ l_{s-1,o} & \text{ha } i=s \wedge j=o \\ l_{i,j} & \text{egyébként} \end{cases}$$

- A *mozgat(jobbra)* operátort az $l \in \mathscr{A}$ állapotra alkalmazva a következőképpen definiált $l' \in \mathscr{A}$ állapotot kapjuk:

  ha az éppen $l_{s,o} = 0$

$$l'_{i,j} = \begin{cases} 0 & \text{ha } i=s \wedge j=o-1 \\ l_{s,o-1} & \text{ha } i=s \wedge j=o \\ l_{i,j} & \text{egyébként} \end{cases}$$

- A *mozgat(balra)* operátort az $l \in \mathscr{A}$ állapotra alkalmazva a következőképpen definiált $l' \in \mathscr{A}$ állapotot kapjuk:

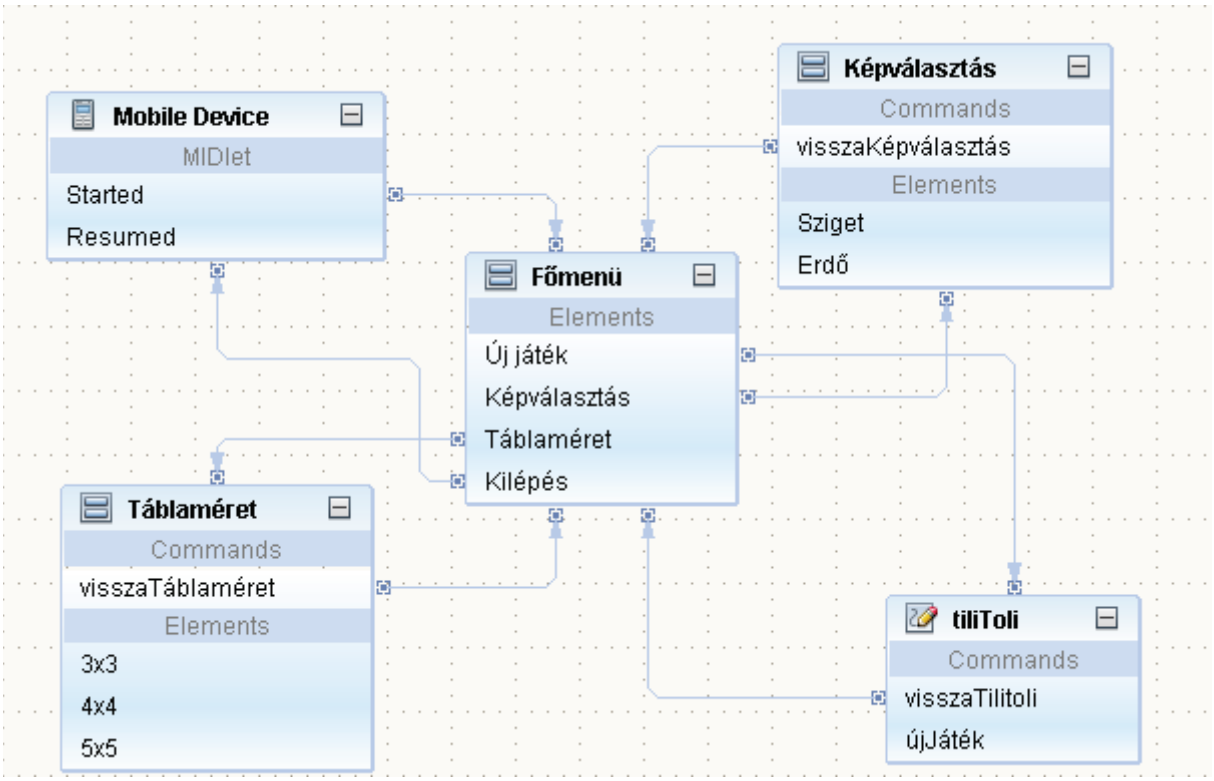
  ha az éppen $l_{s,o} = 0$

$$l'_{i,j} = \begin{cases} 0 & \text{ha } i=s \wedge j=o+1 \\ l_{s,o+1} & \text{ha } i=s \wedge j=o \\ l_{i,j} & \text{egyébként} \end{cases}$$

5.2.3. Megvalósítása Javában, a forráskód elemzése

A program megírásához a Netbeans 6.9.1-es verzióját használtam, azt a változatot, amely támogatja a Java ME platformra történő fejlesztést.

Mivel minden mobil alkalmazásnak tartalmaznia kell egy *MIDlet* osztályt, ami a Java alkalmazások *Main* osztályának felel meg, ennek az alkalmazásnak is van egy ilyen osztálya, amely a következőképpen néz ki:



Ahogy elindul az alkalmazás egy „Főmenü” nevű lista jelenik meg, amelyből újabb listákat tudunk elérni a játék paramétereinek beállításához. Ilyenek például, hogy mekkora részre szeretnék felosztani a táblánkat, vagy éppen milyen képet szeretnék kirakni.

Ezeken kívül található egy „Kilépés” feliratú listaelem, amely arra szolgál, hogy bezárja az alkalmazásunkat. És legfontosabb elemként található még egy „Új játék” nevezetű parancs is, amelyre kattintva létrejön a *Tilitoli* osztálynak egy példánya a kiválasztott, vagy ha nem állítunk semmit, akkor az alapértelmezettként beállított táblamérettel és háttérképpel, és elindul a játék.

Ahogyan az előbb már szó volt róla, az alkalmazásnak van egy *Tilitoli* nevű osztálya, ami a *GameCanvas* leszármazottja. Ennek az osztálynak a feladata a tilitoli tábla megjelenítése, valamint a billentyűnyomások kezelése. A megjelenítést a *frissitKepernyo()* metódus valósítja meg, amely a *Runnable* interfész *run()* metódusában helyezkedik el. A *frissitKepernyo()* pedig újabb metódusokat hív meg, amelyek az alábbiak:

Elsőként meghív egy *letrehozHatter()* metódust, amely beállítja a telefon kijelzőjének háttérszínét feketére és elhelyez a képernyőn egy „Vissza” szövegű feliratot, ami a főmenübe való visszalépésre szolgál, és egy „Új játék” nevű címkét, amelyre ha rákattintunk, akkor a táblán lévő lapocskákat összekavarja.

Ezt követően van egy *megjelenitTabla()* metódus, amely a *TilitoliAllapot* osztály példányának aktuális állapotának tábláját a megfelelő lapocskákhoz párosítva és a telefon kijelzőjén a megfelelő pozíción elhelyezve megjeleníti. Ehhez azonban a kirakandó képet előzőleg azonos méretű *Sprite*-okra kell tördelnünk és egy *Sprite* objektumokat tartalmazó tömbben célszerű eltárolnunk. Így ugyanis könnyen tudunk hivatkozni bármelyik lapocskára.

```
//ax és ay az aktuális x és y koordináta a kijelzőn, db a Spriteok, vagyis a lapocskák száma
int db = 0, ax = 0, ay = 0;
//a teljes kép felosztása lapocskákra
for (int i = 0; i < N; i++) {
    ax = 0;
    for (int j = 0; j < N; j++) {
        kep[db] = Image.createImage(img, ax, ay, képMérete - 1, képMérete - 1, Sprite.TRANS_NONE);
        db++;
        ax += képMérete;    //képMérete egy lapocska méretét jelzi, amely értéke 240/N
    }
    ay += képMérete;
}
//a fekete kép méretének beállítása az aktuális lapocska méretre
kep[0] = Image.createImage(fekete,
        0, 0, képMérete - 1, képMérete - 1, Sprite.TRANS_NONE);

int x = 0, y = 62;  //a kijelző azon pozíciója, ahol a lapocskák megjelenítése kezdődik
int c = 0; // lapsorszám
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        kepSprite[c] = new Sprite(kep[c]);
        kepSprite[c].defineReferencePixel(0, 0);
        kepSprite[c].setRefPixelPosition(x, y);
        c++;
        x += képMérete;
    }
    x = 0;
    y += képMérete;
}
```

A lapocskák mozgatására a *mozgat()* metódus szolgál, amelynek törzse az alábbi:

```
private void mozgat() {
    int gombAllapot = getKeyStates();

    if ((gombAllapot & UP_PRESSED) != 0 && tta.elofeltetel(fel)) {
        tta = (TilitoliAllapot) tta.alkalmaz(fel);
    }
    if ((gombAllapot & DOWN_PRESSED) != 0 && tta.elofeltetel(le)) {
        tta = (TilitoliAllapot) tta.alkalmaz(le);
    }
    if ((gombAllapot & LEFT_PRESSED) != 0 && tta.elofeltetel(balra)) {
        tta = (TilitoliAllapot) tta.alkalmaz(balra);
    }
    if ((gombAllapot & RIGHT_PRESSED) != 0 && tta.elofeltetel(jobbra)) {
        tta = (TilitoliAllapot) tta.alkalmaz(jobbra);
    }
}
```

Ebben a metódusban lekérdezzük, hogy éppen mely gomb lett megnyomva a *getKeyStates()*-cel, majd ezt követően a megfelelő lépést végrehajtjuk.

Az egyes *if* utasításokban azt vizsgáljuk, mely gombot nyomtuk meg és hogy teljesül-e az aktuális állapotra a választott irányba történő lépés előfeltétele. Ha igen, akkor végrehajtódik a lépés, ha nem, akkor nem történik semmi.

Ezen állapotokat az *Allapot* osztály leszármazottja, a *TilitoliAllapot* példányai írják le. Az osztály konstruktorában beállításra kerül a paraméterként megadott (N×N) méretű tábla 0-tól (N-1)-ig értékekkel sorban feltöltve. Majd pedig összekeverjük a benne lévő értékeket a *keverTabla()* metódussal. Ezzel a megadtuk a kezdőállapotot.

```
public TilitoliAllapot(int N) {
    this.N = N;
    int k = 0;
    tabla = new int[N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            tabla[i][j] = k;
            k++;
        }
    }
    keverTabla();
}
```

Az osztályban található még – a *mozgat()* metódusban már látott – *elofeltetel()* illetve *alkalmaz()* metódusok. Előbbi azt vizsgálja, hogy a választott operátor (lásd később) alkalmazható-e az aktuális állapotra. Az *alkalmaz()* pedig módosítja az aktuális állapotot.

Ezeken kívül az osztály tartalmaz még egy *celAllapot()* nevű módszert is, amely azt figyeli, hogy a kép kirakott állapotban van-e, vagyis a tábla értékei a bal-felső saroktól indulva növekvő sorrendben vannak.

```
public boolean celAllapot() {
    int k = 0;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (tabla[i][j] != k) {
                return false;
            }
            k++;
        }
    }
    return true;
}
```

Ahogyan fentebb említettem, a csomagban található még egy *TilitoliMozgat* osztály is, amely az *Operator* leszármazottjaként a műveletek példányait biztosítja. Az osztály konstruktorában tudjuk megadni, hogy éppen mely irányban történjen a lapocska mozgatása.

```
private int irany;

public TilitoliMozgat(int irany) {
    this.irany = irany;
}
```

Az irányok megadására pedig az *Irany* interfész biztosít konstansokat.

```
public interface Irany {

    public static final int FEL = 0;
    public static final int JOBBRA = 1;
    public static final int LE = 2;
    public static final int BALRA = 3;

}
```

5.2.4. Képek a játékról működés közben

5.3. Tic-Tac-Toe

5.3.1. A játék leírása, szabályai

Leírás:

Szintén egy táblajáték a tilitolihoz hasonlóan, azonban van néhány különbség hozzá képest. Az egyik az, hogy itt nem táblán meglévő elemeket mozgatunk, hanem lépésenként egy-egy elemmel bővítve változtatjuk meg a tábla állapotát. A másik nagy különbség pedig az, hogy míg a tilitoli egyszemélyes, addig a tic-tac-toe kétszemélyes játék. Jelen szakdolgozatom gép és felhasználó közti játékot valósít meg. Viszont lehetne még olyannal bővíteni, amikor gép-gép ellen, illetve felhasználó-felhasználó ellen játszik.

Szabályok:

A tic-tac-toe egy (3×3)-as táblán játszódik, amelyre a két játékos egymás után felváltva helyezi el a saját jelét. Egyik játékost az X, másik játékost pedig az O jel szimbolizálja.

A játék célja az, hogy három azonos jel gyűljön ki egy vonalban (sorban, oszlopban vagy valamelyik átlóban). Az a játékos nyer, amelyik ezt a feltételt hamarabb tudja teljesíteni. Ennek függvényében a játéknak három kimenetele lehet. Vagy az X jelű játékos nyer, vagy az O jelű, illetve ha mindkét játékos egyformán jól játszik, akkor az is előfordulhat, hogy egyiknek sem sikerül három jelet összegyűjteni és a tábla megtelik. Ekkor döntetlen lesz a végeredmény.

Mivel a tábla igen kisméretű, ezért ha mindkét játékos jól játszik, akkor a kezdő játékosnak jelentősen több esélye van a győzelemre, másképp fogalmazva a másodikként lépő játékos már legjobb esetben is csak döntetlen eredményt hozhat ki a játékból. Emiatt az kezdő játékos első lépésként nem rakhat a középső mezőre.

5.3.2. Állapottér reprezentációja

Adott egy (3×3)-as tábla, amely a játék indulásakor teljesen üres, és a játék folyamán lépésenként bővül egy-egy elemmel. A játékban két játékos vesz részt, akik egymás után felváltva helyezik el saját jelüket. A feladatuk az, hogy minél előbb kigyűjtsenek egy vonalban hármat a saját jelükből.

A probléma világa: egy tábla és két játékos.

A világ leírása: a tábla egyes pozícióin melyik játékos szimbóluma van.

Egy-egy pozícióra hivatkozás: (sor, oszlop).

pozíció	(1,1)	(1,2)	…	(3,3)
mező	$h_{1,1}$	$h_{1,2}$	···	$h_{3,3}$

ahol $0 \leq h_{i,j} \leq 2$ minden $1 \leq i, j \leq 3$ esetén.

Tehát $H_{i,j} = H = \{0, 1, 2\}$ minden $1 \leq i, j \leq 3$-ra és $p \in \{A, B\}$, ahol $h_{i,j}$ az (i,j) mezőn szereplő érték, p pedig a játékos.

A probléma világának egy-egy állapotát egy-egy olyan (h, p) páros határozza meg, ahol

$$\left(\begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix}, p \right)$$

a (3×3)-as mátrixban az értékek H-beli elemek, p pedig az éppen lépni következő játékos.

Kezdőállapot:

$kezdő \in \mathcal{H}$, az az állapot, amikor a tábla teljesen üres, tehát minden mező értéke 0 és az „A” játékos következik lépni.

$$kezdő = \left(\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, A \right)$$

Végállás: $v \in \mathcal{V}$

$\mathcal{V} = \{(h, p) \mid végállás(h)\}$

Végállás akkor alakul ki, ha a tábla megtelik, vagyis a táblának nincs olyan mezője, amelynek az értéke 0:

$$\forall i \forall j \, (h_{i,j} \neq 0)$$

Vagy ha három azonos jel gyűlik ki egy vonalban (sorban, oszlopban vagy valamelyik átlóban) :

valamelyik sorban: $\exists i \, ((h_{i,1} \neq 0) \supset ((h_{i,1} = h_{i,2} \wedge h_{i,1} = h_{i,3}))$

$\vee$

valamelyik oszlopban: $\exists j \, ((h_{1,j} \neq 0) \supset ((h_{1,j} = h_{2,j} \wedge h_{1,j} = h_{3,j}))$

$\vee$

főátlóban: $(h_{1,1} \neq 0) \supset (h_{1,1} = h_{2,2} \wedge h_{1,1} = h_{3,3})$

$\vee$

mellékátlóban: $(h_{1,3} \neq 0) \supset (h_{1,3} = h_{2,2} \wedge h_{1,3} = h_{3,1})$

$v \in \mathscr{D}$ végállásban az a játékos veszít, aki éppen lépni következik, amikor az előbb felsorolt feltételek közül az utolsó négyből valamelyik teljesül. Ha a feltételek közül egyedül a megtelt tábla feltétel teljesül, akkor a játék kimenetele döntetlen lesz.

Operátorok halmaza:

$\mathscr{O} = \{lerakX(x, y); lerakO(x, y)\}$, ahol $1 \leq x, y \leq 3$

Operátor alkalmazási előfeltételek:

- A *lerakX(x, y)* operátor alkalmazható az $(h, p) \in \mathscr{A}$ állapotra, ha teljesül a következő operátoralkalmazási előfeltétel:
    - Az (x, y) pozíciójú mező üres és „*A*” következik lépni

$$h_{x,y} = 0 \wedge p = A$$

- A *lerakO(x, y)* operátor alkalmazható az $(h, p) \in \mathcal{H}$ állapotra, ha teljesül a következő operátoralkalmazási előfeltétel:
  - Az (x, y) pozíciójú mező üres és „*B*” következik lépni

$$h_{x,y} = 0 \wedge p = B$$

Operátor alkalmazásának hatása:

- A *lerakX(x, y)* operátort a $(h, p) \in \mathcal{H}$ állapotra alkalmazva a következőképpen definiált $(h', p') \in \mathcal{H}$ állapotot kapjuk:

$$h'_{i,j} = \begin{cases} 1 & \text{ha } i = x \wedge j = y \\ h_{i,j} & \text{egyébként} \end{cases}$$

$$p' = B$$

- A *lerakO(x, y)* operátort a $(h, p) \in \mathcal{H}$ állapotra alkalmazva a következőképpen definiált $(h', p') \in \mathcal{H}$ állapotot kapjuk:

$$h'_{i,j} = \begin{cases} 2 & \text{ha } i = x \wedge j = y \\ h_{i,j} & \text{egyébként} \end{cases}$$

$$p' = A$$

5.3.3. Megvalósítása Javában, a forráskód elemzése

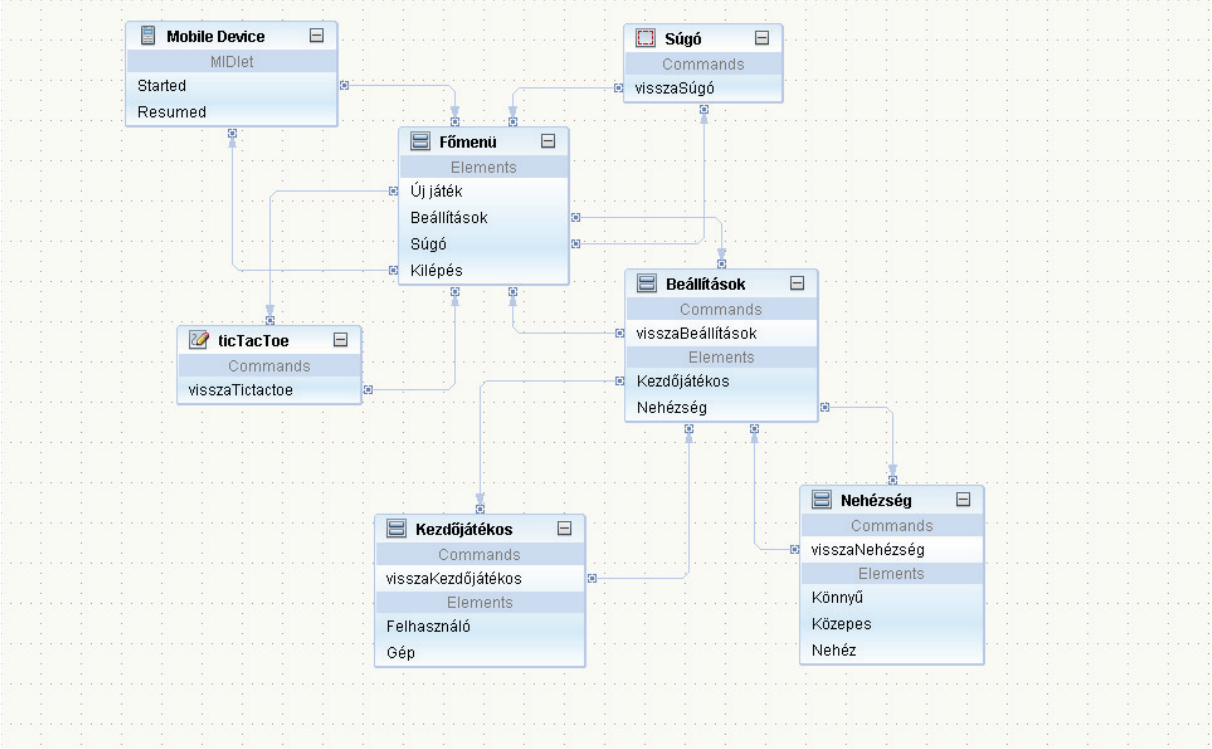
Ahogyan a tilitoliban, ebben az alkalmazásban is szerepelnie kell egy *MIDlet* osztálynak, amelynek jelen esetben a *TictactoeMIDlet* nevet adtam.

A *MIDlet* indulásakor itt is egy „Főmenü” jelenik meg, amiből érhetőek el további lehetőségek a játék tulajdonságainak beállítására, illetve a súgó.

A „Beállítások” menüben lehetőségünk van annak meghatározására, hogy a gép vagy a felhasználó kezdje a játékot, valamint a gépi ellenfél tudásszintjét is ki tudjuk választani

három lehetőség közül. Alapértelmezettként kezdőjátékosnak a felhasználó, nehézségi szintnek pedig a könnyű szint van beállítva.

A „Súgó” menüpont alatt a játék használatáról és a szabályokról találhatunk információt. A „Kilépés” parancs segítségével pedig az alkalmazásból tudunk kilépni.



Az „Új játék” parancsra kattintva létrejön a *Tictactoe* osztálynak egy példánya, és elindul a játék. A *Tictactoe* osztály ebben az esetben is a *GameCanvas* leszármazottja, valamint implementálja a *Runnable* interfészt. Ahogyan a játékvászon név is jelzi, az osztály feladata tábla és a rajta elhelyezkedő jelek megjelenítése, valamint a játék lebonyolítása. Utóbbi alatt a felhasználó által lenyomott billentyűk figyelését, és a gép által ajánlott lépések kezelését értem.

Az osztályban található egy *run()* metódus, amely azt figyeli, hogy éppen melyik játékos következik lépni.

```
gombAllapot = getKeyStates();
if (ttta.getKovJatekos() == A && (gombAllapot & FIRE_PRESSED) != 0) {
    valasztottMezo = kurzorMezo;
    x = valasztottMezo.getX();
    y = valasztottMezo.getY();
    operator = new LerakX(x, y);
    if (ttta.elofeltetel(operator)) {
        ttta = (TictactoeAllapot) ttta.alkalmaz(operator);
    }
} else if (ttta.getKovJatekos() == B) {
    lepesajanlo = new Lepesajanlo(midlet.getNehézség().getSelectedIndex());
    ajanlottMezo = lepesajanlo.ajanl(ttta.getTabla());
    x = ajanlottMezo.getX();
    y = ajanlottMezo.getY();
    operator = new LerakO(x, y);
    if (ttta.elofeltetel(operator)) {
        ttta = (TictactoeAllapot) ttta.alkalmaz(operator);
        kurzorMezo = ajanlottMezo;
    }
}
```

Ha az „A” játékos, vagyis a felhasználó következik lépni, akkor az OK gomb megnyomását követően az alábbi műveletek hajtódnak végre:

Lekérjük annak a mezőnek az x és y koordinátáját, amelyen a kurzor található. Ugyanis a felhasználó a navigációs gombokkal és az OK megnyomásával kiválasztotta, hogy erre a mezőre szeretné elhelyezni a jelét. Ezután megvizsgáljuk, hogy a választott mezőre elhelyezhető-e a felhasználó szimbóluma, jelen esetben az X jel. És ha a feltétel teljesül, akkor az aktuális állapotra alkalmazzuk a választott operátort és átadódik a lépés joga a másik játékosnak, amely az *alkalmaz()* metódusban van implementálva. Ha nem, akkor új mezőt kell választania a felhasználónak egészen addig, amíg az operátoralkalmazási előfeltétel nem tejesül.

Ha pedig a „B” játékos, azaz a gép következik, akkor létrehozunk egy *lepesajanlo* objektumot az aktuális nehézségi szintnek megfelelően, amely az *ajanl()* metódusának segítségével javasol egy lépést a gép számára. Ezután pedig hasonlóan járunk el, mint a felhasználó esetében annyi különbséggel, hogy a lépés után a kurzor arra a mezőre mutat, ahová a gép rakta a jelét. Erre azért van szükség, hogy a felhasználó nyomon tudja követni, mi volt a gép utolsó lépése.

A lépésajánlás az alábbi algoritmus szerint működik. A felírás sorrendjében az elsőnek teljesülő feltételnél megadott utasítást hajtja végre:

1. Ha van két saját jel egy vonalban, akkor rak egy harmadikat.

2. Ha az ellenfélnek, vagyis jelen esetben a felhasználónak van két jele egy vonalban akkor megakadályozza, hogy harmadikat tegyen.
3. Ha üres a középső mező, akkor középre rak.
4. Ha van üres sarki mező, akkor azok közül véletlenszerűen választ egyet.
5. Ha van üres oldalsó mező, akkor közülük választ véletlenszerűen.

Ezt az algoritmust azonban csak a nehéz nehézségi szint kiválasztásakor alkalmazza a játék. Ha közepes nehézségre van állítva, akkor ezek közül néhányat kihagy, főként a támadó lépéseket. Ezalatt az 1. és a 3. lépést értem. Ha pedig a könnyű szint van kiválasztva, akkor teljesen véletlenszerűen választ egy lépést a még üres mezők közül, és nem használja ezt az algoritmust.

A játék addig folytatódik, amíg valamelyik játékosnak ki nem gyűlik a három jele egy vonalban vagy a tábla be nem telik. Ennek a figyelésére a *TictactoeAllapot celAllapot()* metódusa szolgál:

```
public boolean celAllapot() {
    if (isKigyult3()) {
        return true;
    }
    if (isTablaMegtelt()) {
        return true;
    }
    return false;
}
```

Az, hogy kigyűlt-e három, a következő metódus ellenőrzi:

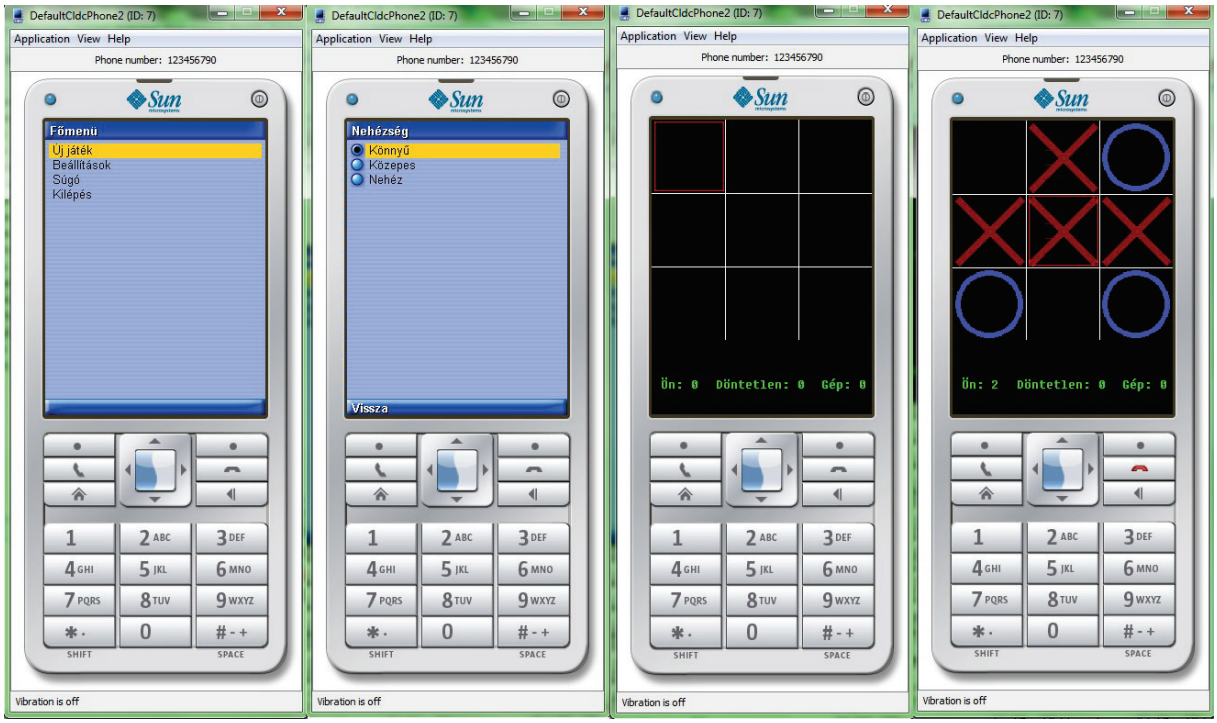
```
private boolean isKigyult3() {
    //sorok ellenőrzése
    for (int i = 0; i < 3; i++) {
        if (tabla[i][0] == tabla[i][1] && tabla[i][0] == tabla[i][2] && tabla[i][0] != 0) {
            nyertes = tabla[i][0] == 1 ? 1 : 2;
            return true;
        }
    }
    //oszlopok ellenőrzése
    for (int i = 0; i < 3; i++) {
        if (tabla[0][i] == tabla[1][i] && tabla[0][i] == tabla[2][i] && tabla[0][i] != 0) {
            nyertes = tabla[0][i] == 1 ? 1 : 2;
            return true;
        }
    }
    //átlók ellenőrzése
    if (tabla[0][0] == tabla[1][1] && tabla[0][0] == tabla[2][2] && tabla[0][0] != 0) {
        nyertes = tabla[0][0] == 1 ? 1 : 2;
        return true;
    }
    if (tabla[0][2] == tabla[1][1] && tabla[0][2] == tabla[2][0] && tabla[0][2] != 0) {
        nyertes = tabla[0][2] == 1 ? 1 : 2;
        return true;
    }

    return false;
}
```

Ezen belül már azt is vizsgáltam, hogy melyik játékosnak gyűlt ki a három jele és ez alapján módosítom a játék pontállását.

```
if (ttta.celAllapot()) {
    switch (ttta.getNyertes()) {
        case 0: {
            dontetlenekSzama++;
            break;
        }
        case 1: {
            jatekosPontszam++;
            break;
        }
        case 2: {
            gepPontszam++;
            break;
        }
    }
```

Ezt követően pedig a táblát üresre állítom és az eredmény kiírását is módosítom a telefon kijelzőjén, és a játék kezdődik elölről.

5.3.4. Képek a játékról működés közben



6. Összefoglalás

A szakdolgozat első felében tehát szó volt a Java nyelvről, jellemzőiről, majd pedig a három Java technológiáról, azok összehasonlításáról. Ezután részletesebb leírást készítettem a Java Micro Edition-ről, annak konfigurációiról és profiljairól.

Az 5. fejezettől kezdve pedig konkrét mobil alkalmazások fejlesztését mutattam be. Először egy egyszemélyes játékot, a tilitolit, majd pedig egy kétszemélyest, a tic-tac-toe-t. Mindkét játék esetében a mesterséges intelligenciából ismert állapotteret implementáltam Java Micro Edition környezetben, a struktúrához illő osztályszerkezettel megvalósítva.

Mind a tilitoli, mind pedig a tic-tac-toe igen jó kikapcsolódást biztosít, viszont még mindegyiket lehetne továbbfejleszteni. A tilitoli esetében például, ahogyan a leírásában is tárgyaltam, lehetett volna a kirakandó képet a (3×3)-as séma helyett más sémában felosztani, ezzel is nehezítve a kirakást. A tic-tac-toe-nál pedig a tábla területét lehetne nagyobbra állítani. Bár akkor már nem is tic-tac-toe, hanem amőba lenne a játék neve. Ekkor viszont már a mesterséges intelligenciából ismert minimax vagy negamax algoritmus használata lenne célszerűbb lépésajánlásként.

De mindezen továbbfejlesztési lehetőségek hiányában is remélem, hogy kellemes időtöltést okoztam a játék használóinak.

7. Köszönetnyilvánítás

Köszönetet mondok Dr. Kósa Márk tanár úrnak, hogy lehetőséget adott ezen szakdolgozat megírásához. Kiváló szakmai tudásával, tapasztalatával és ötleteivel segítette munkámat és legfőképpen azt köszönöm, hogy bármikor tudott fogadni, amikor csak szükségem volt a tanácsaira.

8. Irodalomjegyzék

Dr. Várterész Magda

A mesterséges intelligencia alapjai

A Java programozási nyelv

http://java.about.com/od/gettingstarted/a/whatisjava.htm

Java technológiák

http://www.oracle.com/technetwork/java/javaee/overview/index.html

http://www.oracle.com/technetwork/java/javase/overview/index.html

http://www.oracle.com/technetwork/java/javame/overview/index.html

J2ME: Introduction, Configurations and Profiles

http://www.ifi.uzh.ch/~riedl/lectures/Java2001-j2me.pdf

J2ME: Step by step

http://www.digilife.be/quickreferences/pt/j2me%20step%20by%20step.pdf

Personal Profile

http://java.sun.com/products/personalprofile/overview.html

A szakdolgozatban felhasznált képek az adott témákról szóló weboldalakról lettek letöltve.

Plágium - Nyilatkozat

Szakdolgozat készítésére vonatkozó szabályok betartásáról nyilatkozat

Alulírott (Neptunkód: HTZ66N) jelen nyilatkozat aláírásával kijelentem, hogy az

Állapottér-reprezentációk és implementációjuk mobiltelefonos alkalmazásokban

című szakdolgozat/diplomamunka

(a továbbiakban: dolgozat) önálló munkám, a dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. tv. szabályait, valamint az egyetem által előírt, a dolgozat készítésére vonatkozó szabályokat, különösen a hivatkozások és idézések tekintetében.

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

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

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

hallgató

Debrecen, 2011. április 30.