

# ***SZAKDOLGOZAT***

*Cédl Zoltán*

*Debrecen  
2011*

**Debreceni Egyetem  
Informatikai kar**

# **WEBES JÁTÉK FEJLESZTÉSE MESTERSÉGES INTELLIGENCIA ALKALMAZÁSÁVAL**

**Témavezetők:**

Dr. Kósa Márk  
Egyetemi tanársegéd és  
Csomor Benő

**Készítette:**

Cédl Zoltán  
Programtervező informatikus

Debrecen  
**2011**

# Tartalomjegyzék

Bevezetés.....	4
1. A megvalósítandó játékról.....	5
2. A kétszemélyes játékokról általánosságban.....	6
3. A játék formalizálása.....	8
3.1. Állapottér.....	9
3.2. Kezdőállapot.....	9
3.3. Operátorok.....	10
3.3.1. A hív operátor.....	10
3.3.2. A passz operátor.....	11
3.4. Célállapot.....	11
4. A játék megvalósítása.....	12
4.1. Az állapottér megvalósítása Java-ban.....	12
4.2. Keresőalgoritmusok.....	15
4.2.1. Minimax algoritmus.....	16
4.2.2. Heurisztikus minimax algoritmus.....	17
4.3. Heurisztika.....	19
5. A játék online verziója.....	20
5.1. Google Web Toolkit.....	20
5.2. Adatbázis.....	25
5.2.1. Az adatbázis kapcsolat felépítése Java-ban.....	26
5.3. Regisztráció.....	27
5.4. Bejelentkezés.....	28
5.5. Az aszinkron hívások a regisztrációnál és bejelentkezésnél.....	28
5.5.1. RPC megvalósítása a bejelentkezésnél.....	29
5.5.2. RPC megvalósítása a regisztrációnál.....	32
5.6. Főmenü.....	32
5.7. Játéklablak.....	33
5.8. Telepítés web szerverre.....	35
6. Összefoglalás.....	37
Függelék.....	38

## Bevezetés

A játék fogalma: A játék minden külső kényszerítő körülményt nélkülöző, szabad cselekvés. Játsszani nem kötelesség, azaz a játszás egy teljesen szabadon választott tevékenységi forma. Mindebből már látható, hogy a játék annyiban „szabad cselekvés”, amennyiben a társadalom, a kultúra a maga sajátos eszközeivel nem szorítja rá az individuumot a játéokra. Pszichológiai szempontból azonban a játék - valamely adott korcsoport számára - az önkifejezés, a megismerés, általában a társadalmi lét egyedül lehetséges formája. Másrészt a játék fogalma szinte a végtelenségig kitágul, s idetartoznak a gyermekjátékok mellett a felnőttek olyan időtöltései is, mint a különböző sportok, a kártyázás, a sakkozás. Megállapíthatjuk, a játékokat (általában is) az jellemzi, hogy a szabad akaraton alapulnak, nem produktívak, valóságszerűek, de nem valóságosak, térben és időben meghatározottak.

A játék figyelem, gondolkodás és emlékezet fejlesztésére szolgál. Nagymértékben fejleszti a különböző készségeket, képességeket, pl.: érzékelés, észlelés képességét, logikai készséget és a matematikai képességet. Ezek leginkább a kártyajátékokra jellemzőek. Az internet megjelenése előtt a kártyajátékokat csak papírból készült lapokkal lehetett játszani. Az internet elterjedése azonban széles körben lehetőséget biztosított a kártyajátékok online játszására is. Több száz fajta kártyajáték található az interneten. Az általam választott kártyajáték, a zsírozás, nagyon népszerű és elterjedt magyar kártyajáték. Az online változata kevés verzióban található meg.

Célunk egy olyan kétszemélyes webes verzió elkészítése, amely a gép és ember közötti játék lehetőségét biztosítja. Akiknek eddig még nem volt lehetőségük megismerni ezt a játékot, illetve társaság hiányában egyedül nem tudják játszani, azoknak lehetőséget kívánunk nyújtani az online játék játszására. Szeretnénk, ha minél több ember ismerhetné meg a zsírozást, hiszen egyaránt kellemes időtöltésre alkalmas és mellette gondolkodásra is készítő játék. Egy kis kikapcsolódást és a való világ gondjaitól kis időre elszakadást tudnánk hozni a játékosok számára.

## 1. A megvalósítandó játékról

A zsírozás egy egyszerű kártyajáték, amelyet magyar kártyával játszanak. A játék célja, hogy minél több ászt vagy tízest azaz „zsírt” gyűjtsünk össze az ütéseinkkel. A lehívott lapot csak azonos értékű lappal lehet ütni. Kiemelt szerepe van a hetesnek, mert ezzel bármit lehet ütni. A játékban nincs adu, a lapoknak nincs pontértéke és rangsora, így az nyer aki több zsírt gyűjt össze. A játékot ketten, hárman vagy négyen is lehet játszani. Hogyha többen játszunk, akkor szabály kiegészítést kell tennünk, például ha hárman játszunk, akkor ki kell venni két darab nyolcast a pakliból és a bent maradt nyolcas is hetes szintű ütő kártya lesz. Ha négy fő játszik, akkor ketten-ketten egy párt alkotnak, akik egymással szembe ülnek. Osztásnál minden játékos négy-négy lapot kap, a többi kártyát leteszik középre, ez lesz a pakli, amiből a kör végén húznak a játékosok. Az osztótól jobbra ülő játékos kezdi a kört. Egy lapot többször is meg lehet ütni. Aki a kör végén utoljára ütött az viszi a lapokat, ő húz elsőnek, és őt illeti meg a jog, hogy elsőként hívjon. Aki a játék során egyszer sem tudott vitt lapokat az „kopasz” marad. Mi a két személyes játék verziót fogjuk implementálni a továbbiakban.



1. ábra: egy példajáték

## 2. A kétszemélyes játékokról általánosságban

Sok tudományág történetében játszottak nagy szerepet különféle játékok, különösen igaz ez a mesterséges intelligencia fejlődésére. Az elsőként komolyan vizsgált játék a sakk volt (1950, Claude Shannon és Alan Turing sakkprogramja). Ezt a játékot azért tartották kiemelkedően fontosnak, mert a sakk olyasvalami, amiről általában úgy vélekednek, hogy intelligenciát igényel, továbbá a szabályai formálisan is egyszerűen megfogalmazhatók, továbbá állapotait is könnyű leírni. A játék mindezek következtében könnyen leírható egy állapottérben való kereséssel. Ez utóbbi megállapítás sok más játékra (elsősorban a kétszemélyes, táblás játékokra) igaz, ám az állapottér egyes játékokban nagyon nagy lehet. Ennek következtében a kétszemélyes játékok kutatásában döntő szerepet kaptak a keresést gyorsító, valamint az állapotteret csökkentő eljárások. Az eljárás maga sem egy egyszerű legrövidebb-út keresésének felel meg, ugyanis a játékok esetében számolnunk kell egy ellenfél jelenlétével, amely bizonytalanságot vezet be a problémába. Nem tudunk ugyanis előre egy olyan utat előállítani, amely mindenképpen célra vezet, hiszen nem tudhatjuk, hogy partnerünk milyen lépéseket fog tenni válaszként a magunk lépéseire. Így egy komplexebb feladattal nézünk szembe: egy olyan stratégiát kell kidolgoznunk, amelyet minden körben le kell futtatnunk, amikor ránk kerül a sor (azaz futási, végrehajtási időben), és ez alapján kell döntenünk, hogy az adott helyzetben mi a legmegfelelőbb lépés. Egy másikfajta bizonytalanságot vezet be a fentebb már említett probléma, mely szerint a keresési tér általában igen nagy. Így szinte lehetetlen olyan valós időben lefutó algoritmust létrehozni, amely garantáltan célravezető megoldást ad, azaz a lépéseink során nem fogjuk tudni megjósolni, hogy az adott lépés nyereshez vezet. Ezek a problémák különböztetik meg a játékok problémáját az egyéb keresések problémájától. Összegezve, egy játék leírásához meg kell adnunk:

- a játék lehetséges állásai (helyzeteit)
- a játékosok számát
- hogyan következnek lépni az egyes játékosok (pl.: egy időben vagy felváltva egymás után)
- egy-egy állásban a játékosoknak milyen lehetséges lépései (lehetőségei) vannak

- van-e szerepe a véletlennek a játékban és milyen feltételek mellett
- milyen állásban kezdődik és mikor ér véget a játék
- a játékosok mennyit nyernek, ill. veszítenek a játszmák során

A játékokat két nagy csoportba tudjuk sorolni:

1. szerencsejátékok: itt minden a véletlen műve, a játékosok nem tudják befolyásolni a játék kimenetelét (pl.: rulett)
2. stratégiai játékok: a játékosoknak ellenőrizhető módon van befolyásuk a játék kimenetelére (pl.: sakk, póker, amőba, malom)

Neumann János maga is foglalkozott ezzel. Harsányi János 1994-ben közgazdasági Nobel-díjat kapott, a játékelmélet az ő nevéhez fűződik.

A stratégiai játékokat különböző jellemzők alapján több csoportba tudjuk sorolni:

- játékosok száma szerint: 2,3,...,n személyes játékok, néha a véletlent is személynek veszik
- játszma állásból állásba vivő lépések sorozata, pl.: diszkrét játékok
- az állásokban véges sok lehetséges lépése van-e minden játékosnak és a játszmák véges sok lépés után véget érnek-e, pl.: véges / nem véges játékok
- a játékosok a játékkal kapcsolatos összes információval rendelkeznek-e a játék folyamán, pl.: teljes / részleges információjú játékok
- a véletlennek szerepe van-e a játékban, pl.: determinisztikus / sztochasztikus játékok
- a játékosok veszteségeinek és nyereségeinek összege megegyezik-e, pl.: zéró / nem zéróösszegű játékok

Ezek a legfontosabb jellemzők egy stratégiai játék leírásánál. A továbbiakban olyan játékot tekintünk, amely kétszemélyes, diszkrét, véges, teljes információjú, determinisztikus és zéróösszegű.

A meglévő jellemzők mellett valamilyen módon reprezentálnunk is kell a játékot. A játék problémavilágától és összetettségétől függően több módszert is használhatunk.

A legelterjedtebb játékreprezentációs módszer az állapottér-reprezentáció, amelynél meg kell adnunk a  $\langle A, kezdő, V, O \rangle$  négyes, ahol:

- $A$  halmaz elemei olyan  $(a, J)$  párok ahol  $a \in H$  egy játékállás,  $J \in \{A, B\}$  soron következő játékos
- $kezdő \in H$  a játék kezdőállapota,  $kezdő = (a_0, J_0)$
- $V \subset H$  olyan  $(a, J)$  párok halmaza ahol  $a \in H$  végállapot,  $J$  játékos nyer
- $O$  nem üres halmaz, az operátorok halmaza

Egy ilyen állapottér-reprezentációt szemléltetni tudunk egy játékfával, amelynek a gyökere a kezdőállapot, a faelemek a kezdőállapotból elérhető játékállapotok, a fa levélelemi pedig a célállapotok. Páros szinteken lévő állásokban a kezdő játékos, páratlan szinteken pedig az ellenfele léphet. Egy állást annyi különböző csúcs szemléltet, ahány különböző módon a játék során a kezdőállásból eljuthatunk hozzá. Az utak véges hosszúságúak, hiszen véges játékkal foglalkozunk. Ha a játék során a kezdő állapotból a játékosok valamelyik célállapotba érnek, akkor azt mondjuk, hogy lejátszottak egy játszmát. A játszmákat a játékfában a startcsúcsból a levélelemekbe vezető utak szemléltetik. Egy játék játékfája a játék összes játszmáját szemlélteti.

### 3. A játék formalizálása

A játékot állapottér-reprezentációval fogjuk formalizálni, melynek lépései:

1. A két játékos A és B. A szimbolizálja a gépet, B pedig minket.
2. Meghatározzuk a játék állások halmazát. Egy játékállást egy  $4 \times 8$  elemű mátrix szimbolizál, amelyet a két játékos kezében lévő lapok, a pakli és az asztalon lévő lehívott lapok együttese határozza meg. Mivel az operátoraink alkalmazási előfeltételeit úgy fogjuk meg meghatározni, hogy azok mindig játékállásból játékállásba vigyenek, így nem kell minden játékállásra külön kényszerfeltételeket szabnunk.
3. Meghatározzuk a kezdőállapotot, ami azt jelenti, hogy a játékosok kezébe nincsen még lap, a pakliba benne van mind a 32 kártyalap és az asztalon nincs lehívott lap.



4. Definiálunk két operátort. Az egyik a hív(játékos, kártyalap), a másik a passz(játékos) operátor lesz.
5. Meghatározzuk a célállapotot, azaz a két játékos kezében nincsen lap, az asztalon sincs lap és a pakliból elfogytak a lapok.

### 3.1. Állapottér

A játék világának állapotait rendezett elem 32-esekkel írhatjuk le. Az állapottér  $A \subseteq A_{1,2} \times A_{1,3} \times A_{1,4} \times A_{1,5} \times A_{1,7} \times A_{1,8} \times A_{1,9} \times A_{1,10} \times A_{2,2} \times A_{2,3} \times A_{2,4} \times A_{2,7} \times A_{2,8} \times A_{2,9} \times A_{2,10} \times A_{3,2} \times A_{3,3} \times A_{3,4} \times A_{3,5} \times A_{3,7} \times A_{3,8} \times A_{3,9} \times A_{3,10} \times A_{4,2} \times A_{4,3} \times A_{4,4} \times A_{4,5} \times A_{4,7} \times A_{4,8} \times A_{4,9} \times A_{4,10} \times J$  ahol a Descartes-szorzatban szereplő halmazok:  $A_{i,j} = \{1,2,3,4\}$ ,  $J = \{1,2\}$   $i = 1,2,3,4$   $j = 2,3,4,5,7,8,9,10$

- $i :=$  a kártyalap színét jelenti: piros, zöld, makk, tők
- $j :=$  a kártyalap értékét jelenti: alsó, felső, király, ász, hetes, nyolcas, kilences, tízes

$$\text{Az } \left( \begin{pmatrix} a_{1,2} & \cdots & a_{1,10} \\ \vdots & \ddots & \vdots \\ a_{4,2} & \cdots & a_{4,10} \end{pmatrix}, l \right) \in A \text{ állapotban } a_{i,j} = \begin{cases} 1, \text{ ha } A \text{ játékosnál van} \\ 2, \text{ ha } B \text{ játékosnál van} \\ 3, \text{ ha a pakliban van} \\ 4, \text{ ha a kijátszott lapok közt van} \end{cases}$$

ahol  $i = 1,2,3,4$ ;  $j = 2,3,4,5,7,8,9,10$  és  $l$  az a játékos, aki épp lépni készül.

A kényszerfeltétel azt fogalmazza meg, hogy egy játékos kezében legfeljebb négy lap lehet:

$$\text{Legyen } J_A \subseteq \{(a_{i,j}) \in A : \forall_i \forall_j a_{i,j} = 1 \wedge i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}\}$$

$$\text{Legyen } J_B \subseteq \{(a_{i,j}) \in A : \forall_i \forall_j a_{i,j} = 2 \wedge i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}\}$$

$$0 \leq \|J_A\| \leq 4 \wedge 0 \leq \|J_B\| \leq 4$$

### 3.2. Kezdőállapot

$$k = \left( \begin{pmatrix} 3 & \cdots & 3 \\ \vdots & \ddots & \vdots \\ 3 & \cdots & 3 \end{pmatrix}, 1 \right) \in A$$

### 3.3. Operátorok

Kettő különböző operátort fogunk definiálni.

Vezessük be az alábbi predikátumot, amely az  $A_{i,j}$ -hez tartozó kártyalapot szimbolizálja:

$$\text{kártyalap} = \{(k, l) \in \mathbb{R}^2 : k = i \wedge l = j\}$$

#### 3.3.1. A hív operátor

A hív(játékos, kártyalap)  $\in O$  operátor lehívja a játékos kezéből a kártyalapot az asztalra. A következőképp adhatjuk meg:

$$\text{hív} : \text{dom}(\text{hív}) \rightarrow A,$$

ahol

$$\text{dom}(\text{hív}) \subseteq A.$$

Az operátor hatása:

$$\text{hív}\left(\left(\begin{pmatrix} a_{1,2} & \cdots & a_{1,10} \\ \vdots & \ddots & \vdots \\ a_{4,2} & \cdots & a_{4,10} \end{pmatrix}, l\right), \text{játékos, kártyalap}\right) = \left(\begin{pmatrix} a'_{1,2} & \cdots & a'_{1,10} \\ \vdots & \ddots & \vdots \\ a'_{4,2} & \cdots & a'_{4,10} \end{pmatrix}, l'\right)$$

ahol

$$a'_{i,j} = \begin{cases} 4, & \text{ha } (i,j) = \text{kártyalap} \\ a_{i,j}, & \text{egyébként} \end{cases}$$

$$l' = \begin{cases} 1, & \text{ha játékos} = 2 \\ 2, & \text{ha játékos} = 1 \end{cases}$$

$$\text{ahol } i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}$$

Az operátor alkalmazásának előfeltételei:

Csak akkor hívható le a kártyalap, ha az a játékos kezében van:

$$\left((i,j) = \text{kártyalap} \wedge a_{i,j} = 1\right) \supset \text{játékos} = 1$$

vagy

$$\left((i,j) = \text{kártyalap} \wedge a_{i,j} = 2\right) \supset \text{játékos} = 2$$

$$\text{ahol } i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}.$$

Ha a kártyalap a pakliban vagy az asztalon van nem hívható le:

$$a_{i,j} \neq 3 \wedge a_{i,j} \neq 4$$

$$\text{ahol } i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}.$$

### 3.3.2. A passz operátor

A passz(játékos)  $\in O$  operátor tulajdonképpen két műveletet végez: a játékos véget vet a körnek és húznak a játékosok. A következőképp adhatjuk meg:

$$\text{passz}: \text{dom}(\text{passz}) \rightarrow A,$$

ahol

$$\text{dom}(\text{passz}) \subseteq A.$$

Az operátor hatása:

$$\text{passz} \left( \left( \begin{pmatrix} a_{1,2} & \cdots & a_{1,10} \\ \vdots & \ddots & \vdots \\ a_{4,2} & \cdots & a_{4,10} \end{pmatrix}, l \right), \text{játékos} \right) = \left( \begin{pmatrix} a'_{1,2} & \cdots & a'_{1,10} \\ \vdots & \ddots & \vdots \\ a'_{4,2} & \cdots & a'_{4,10} \end{pmatrix}, l' \right)$$

ahol

$$a'_{i,j} = \begin{cases} 1, \text{ha } \|J_A\| < 4 \\ 2, \text{ha } \|J_B\| < 4 \\ a_{i,j}, \text{egyébként} \end{cases} \quad l' = \begin{cases} 1, \text{ha játékos} = 2 \\ 2, \text{ha játékos} = 1 \end{cases}$$

$$\text{ahol } i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}$$

Az operátor alkalmazásának előfeltételei:

Csak a pakliban lévő kártyákat lehet húzni:

$$a_{i,j} = 3$$

$$\text{ahol } i = \{1,2,3,4\} \wedge j = \{2,3,4,5,7,8,9,10\}$$

Csak akkor passzolhat a játékos, ha a két játékos kezében a lapok száma megegyezik és nem lehet négy kártya a kézben:

$$\|J_A\| = \|J_B\| \wedge \|J_A\| \neq 4 \wedge \|J_B\| \neq 4$$

### 3.4. Célállapot

$$V = \left\{ \left( \begin{pmatrix} 4 & \cdots & 4 \\ \vdots & \ddots & \vdots \\ 4 & \cdots & 4 \end{pmatrix}, 1 \right) \in A, \left( \begin{pmatrix} 4 & \cdots & 4 \\ \vdots & \ddots & \vdots \\ 4 & \cdots & 4 \end{pmatrix}, 2 \right) \in A \right\}$$

Ezzel a  $\langle A, k, V, O \rangle$  négyessel megadtuk az állapotterünk egy lehetséges reprezentációját.

## 4. A játék megvalósítása

A játéknak a Google Web Toolkit (GWT) segítségével egy Java és JavaScript nyelven, Apache webszerver és JDBC felhasználásával, SmartGWT felülettel rendelkező, Firefox webböngészőre optimalizált megvalósítását implementáljuk. A fejlesztéshez az Eclipse Galileo fejlesztői környezetet fogjuk használni.

### 4.1. Az állapottér megvalósítása Java-ban

A következő lépésben meg kell valósítanunk kód szinten a játékunk állapottér-reprezentációját. A játékunk állapotterét egy saját `Allapot` osztályból fogjuk származtatni, amely definiálja a legfontosabb attribútumokat és függvényeket:

- `public List<Operator> operatorok;`  
Az operátorok listáját tartalmazza.
- `public abstract boolean celAllapot();`  
A célállapot vizsgálatát ebben a metódusban kell implementálni.
- `public abstract boolean elofeltetel( Operator op );`  
Az operátorok alkalmazási előfeltételeinek vizsgálatát ebben a metódusban kell implementálni.
- `public abstract Allapot alkalmaz( Operator op );`  
Az operátorok hatásának meghatározását ebben a metódusban kell implementálni.
- `public abstract int heurisztika();`  
A játékállapothoz tartozó heurisztika értéke ennek a metódusnak lesz a visszatérési értéke.

Az előbb definiált osztályunkat a `ZsirAllapotTer` osztály fogja kiterjeszteni és definiálja a játékosokat, a paklit és a játékasztalt.

```
public class ZsirAllapotTer extends Allapot implements Serializable{

    /**
     * a 2 játékos
     */
    public Jatekos Ajatekos;
    public Jatekos Bjatekos;
    /**
     * a játszmában éppen következő játékos
     */
    public Jatekos aktualisJatekos;
```

```

/**
 * kártyapakli
 */
public List<KartyaLap> pakli = null;
/**
 * lehívott lapok az asztalon
 */
public List<KartyaLap> asztal = null;

...
}

```

Mielőtt rátérhetnénk az operátorokra, implementálnunk kell a `Jatekos` és `KartyaLap` segédosztályokat. A `Jatekos` osztály definiálni fogja a játékos nevét, azonosítóját (ID), kezében lévő lapokat és hány darab zsírt vitt el. A `KartyaLap` osztály definiálni fogja a kártyalap színét és értékét:

```

public class KartyaLap implements RandomAccess, Serializable{

    public static final int PIROS = 1, ZOLD = 2, MAKK = 3, TOK = 4;
    public static final int VII = 7, VIII = 8, IX = 9, X = 10;
    public static final int ALSO = 2, FELSO = 3, KIRALY = 4, ASZ = 5;

    public int szin;
    public int ertek;

    ...
}

public class Jatekos implements Serializable{

    public int ID;
    public String nev;
    public List<KartyaLap> lapok = null;
    public int zsirDB;

    ...
}

```

A fentebb definiált osztályokból jól látszik, hogy az eredeti állapotterünket felbontottuk kisebb darabokra. Külön kezeljük a paklit és a játékosoknál lévő lapokat. Erre azért volt szükség, mert szeretnénk az állapotterünket átláthatóbbá, jobban kezelhetőbbé tenni programozás szempontjából, és ami talán fontosabb, hogy ezáltal csökkenteni tudjuk az erőforrásigényt.

Ezután létre kell hoznunk az operátorainkat implementáló osztályokat. Mint azt már korábban említettük, az operátorainkat úgy fogjuk megalkotni, hogy mindig játéállásból játéállásba vigyenek, így nem kell külön kényszerfeltételeket szabnunk az egyes játéállapotokra. Mindkét operátorunk az `Operator` osztályt fogja kiterjeszteni.

Ez egy absztrakt osztály, csupán azért van rá szükségünk, ha általánosságban szeretnénk az operátorokra hivatkozni, pl. majd a keresőben fogjuk alkalmazni. Az operátoraink alkalmazási előfeltételeit és hatásukat az állapotterünket megvalósító `ZsirAllapotTer` osztályban fogjuk implementálni.

Az első operátorunk a hív operátor lesz, amely fentebb leírtak szerint a játékos kezéből a kiválasztott lapot leteszi az asztalra. Két paramétere lesz, az egyik a játékos, a másik pedig a kártyalap, amit le kell hívni.

```
public class Hiv extends Operator implements Serializable{  
    public Jatekos jatekos;  
    public KartyaLap mit;  
    ...  
}
```

A második operátorunk a passz operátor lesz, amelynek két fő feladata lesz: először megmondja, hogy melyik játékos vet véget a körnek, majd meghatározza, hogy ki kezdi a következő kört, és ez alapján húznak a játékosok. Egy paramétere lesz, a játékos, aki passzolt.

```
public class Passz extends Operator implements Serializable{  
    public Jatekos jatekos;  
    ...  
}
```

A célállapot meghatározását a következőképpen fogjuk implementálni:

```
public boolean celAllapot() {  
    if (pakli.isEmpty() &&  
        Ajatekos.getLapok().isEmpty() &&  
        Bjatekos.getLapok().isEmpty() &&  
        asztal.isEmpty()) return true;  
    else return false;  
}
```

Ezzel befejeztük a játék állapottér-reprezentációjának implementálását, a következő feladatunk a keresőalgorithmus és heurisztika implementálása.

## 4.2. Keresőalgoritmusok

A játék állapotter-reprezentációját egy gráf segítségével tudjuk szemléltetni, ez nevezzük játékgráfnak. Az állapotterünk elemei (az állapotok) a gráf csúcsai. A gráf csúcsai közül kitüntetett szerepet játszanak a kezdőállapotot szemléltető startcsúcs és a célállapotokat szemléltető terminális csúcsok. Ha a gráf egyik csúcsából közvetlenül elérhető a másik csúcs, akkor a közöttük lévő utat irányított él jelzi. Ha ezt a gráfot „kiegyenesítjük” egy fává, akkor a játék minden lehetséges játszóját szemléltető fát kapunk. Erre a fára azért lesz szükségünk, hogy a különböző kereséseket és műveleteket végrehajtva értékelni tudjuk az állásokat és a lépéseket, ugyanis az lesz a feladatunk, hogy egy játékosnak egy játékállásban „elég jó” lépést tudjunk ajánlani. Az emberi játékosoknál is hasonlóképpen működik, ugyanis annak a játékosnak van nagyobb esélye a játék megnyerésére, aki több lépésnyire tud előre gondolkodni és minél több lépéskombinációt tud mérlegelni. Ehhez nyújt segítséget a játék valamilyen mélységű és szélességű bejárása. A bejárás sebességét és eredményességét nagyban befolyásolja az állapotterünk összetettsége és a játékfánk mérete. A mi esetünkben, ahogy a későbbiekben látni fogjuk, a fánk mérete és összetettsége nagymértékben korlátozni fogja lépésajánlást. Ha a kezdőállapotból indulunk ki, akkor a paklit  $32! \sim 2.63 \cdot 10^{35}$  féleképpen tudjuk megkeverni és az első húzás után az első játékos a kezében lévő négy lapját 
$$\binom{32}{4} = \frac{32!}{4! \cdot (32-4)!} = 35960$$
 féleképpen kaphatja meg, a második játékos pedig a saját lapjait 
$$\binom{28}{4} = \frac{28!}{4! \cdot (28-4)!} = 20475$$
 féleképpen kaphatja meg. Ez összesen több mint ötvenötezer állapotot jelent, amely csupán csak a kezdőállapotból jött létre az első húzás után. Ha minden ilyen állapothoz hozzászámoljuk az ellenfél lehetséges lépéseinek számát, akkor jól látszik a számokban, hogy ennyi elemnek a bejárására és kiértékelésre reális időn belül az algoritmus képtelen.

A lépésajánláshoz három kereső algoritmust fogunk megnézni, egy általános minimax keresőt, egy heurisztikus minimax keresőt és egy speciális a játékra szabott heurisztikus minimax keresőt. Az utóbbit fogjuk implementálni és használni a játék során.

### 4.2.1. Minimax algoritmus

Klasszikus lépésajánló algoritmus, amelynek az a célja, hogy a támogatott játékosnak egy adott játékállásban „elég jó” lépést ajánljon. Az algoritmus a „minimalizáljuk a veszteséget és maximalizáljuk a nyereséget” elvre épül. A keresés végén visszakapott érték alapján úgy határozzuk meg a használandó operátort, hogy a gyökérelem gyermekeit sorba végigjárjuk és kiválasztjuk azt, amelyiknek a heurisztikája egyenlő ezzel az értékkel. Ha több ilyen elem is lenne, akkor azt választjuk ki amelyikhez rövidebb részfa tartozik, azaz a végállapotba rövidebb úton tudunk eljutni. Az algoritmus minden faelemhez rendelni fog egy heurisztikus értéket, viszont tényleges heurisztikát csak a levélelemeknél (terminális csúcsoknál) fog számolni, a többi faelem heurisztikus értéke pedig a gyermekelemek heurisztikájának minimuma / maximuma lesz attól függően, hogy páros vagy páratlan szinten állunk a játékfaban. Ha páros szinten vagyunk, akkor a gépi játékos készül lépni. Mivel az a célja, hogy az emberi játékost a lehető legrosszabb állásba vigye, ezért az ezen a szinten lévő faelemek heurisztikája egyenlő lesz a gyermekelemek heurisztikájának minimumával. Ha páratlan szinten vagyunk, akkor az emberi játékos készül lépni. Mivel a heurisztikát megadó függvényünket úgy definiáltuk, hogy mindig a gépi játékos számára adja meg a lépés jóságát, a lehetséges lépések közül mindig a legjobbat fogjuk választani, azaz a faelem heurisztikája a gyermekelemek heurisztikus értékeinek maximuma lesz. Az algoritmusnak szükséges bejárnia egy bizonyos mélységig a játékfát ahhoz, hogy lépést tudjon ajánlani, viszont ez a lépés a lehető legjobb lépés lesz. Az algoritmus pszeudokódja:

```
FUNCTION MiniMaxKereso (faelem, melyseg)
  IF melyseg = 0 OR faelem  $\in \mathcal{V}$  THEN RETURN faelem.heurisztika
  END IF
  max  $\leftarrow -\infty$ 
  FOR ALL operator  $\in \mathcal{O}$  DO
    gyermek  $\leftarrow$  alkalmaz(faelem, operator)
    max  $\leftarrow$  MAX(max, -(MiniMaxKereso(gyermekek, melyseg-1)))
  END FOR
  RETURN MAX
END FUNCTION
```



A kód felhasználja azt a megfigyelést, hogy  $\max(a, b) = -\min(-a, -b)$ , így egységesen tudja kezelni a két játékost.

#### 4.2.2. Heurisztikus minimax algoritmus

Ha szeretnénk az előbbinél gyorsabban lépést ajánlani, akkor le kell csökkentenünk a bejárando faelemek számát. A heurisztikus minimax kereső egy ilyen módszert fog megvalósítani. Az algoritmus a minimax elvet fogja felhasználni, a módszer lényeg abban rejlik hogy, nem csak a levélelemekben fog heurisztikát számolni, hanem minden egyes faelem összes gyermekére kiszámoljuk a hozzátartozó játékállás heurisztikus értékét, viszont a faelem gyermekei közül csak azoknál fogjuk a fát tovább építeni, ahol a heurisztika a legmagasabb / legalacsonyabb volt attól függően, hogy a fa páros vagy páratlan szintjén állunk, a többi ággal felesleges foglalkoznunk. A heurisztika pontatlanságából adódóan az algoritmus nem tudja garantálni, hogy a legjobb operátort fogja ajánlani. Másfelől bizonytalanságot ad az is, hogy nem a részfához kötjük a heurisztikus érték meghatározását, hanem az adott játékálláshoz, így előfordulhat olyan eset is, ahol kezdetben a játékállapotot nagyon jónak értékeljük viszont hosszútávon lehet nem a legjobb megoldást adja és egy másik lépéssel jobban jártunk volna. Az algoritmus pseudokódja:

```
FUNCTION Minimax-lépés(<A, kezdő, V, O>, állapot, korlát, h)
  max ←  $-\infty$ 
  operátor ← Nil
  FOR ALL o ∈ O DO
    IF Előfeltétel(állapot, o) THEN
      új-állapot ← Alkalmaz(állapot, o)
      v ← Minimax-Érték(hA, kezdő, V, Oi, új-állapot, korlát - 1, h)
      IF v > max THEN
        max ← v
        operátor ← o
      END IF
    END IF
  END FOR
  RETURN operátor
END FUNCTION
```

```

FUNCTION Minimax-Érték(<A, kezdő, V, O>, állapot, mélység, h)
  IF állapot  $\in \mathcal{V}$  or mélység = 0 THEN
    RETURN h(állapot)
  ELSE IF Játékos[állapot] = J THEN
    max  $\leftarrow -\infty$ 
    FOR ALL o  $\in \mathcal{O}$  DO
      IF Előfeltétel(állapot, o) THEN
        új-állapot  $\leftarrow$  Alkalmaz(állapot, o)
        v  $\leftarrow$  Minimax-Érték(<A, kezdő, V, O>, új-állapot, mélység - 1, h)
        IF v > max THEN
          max  $\leftarrow$  v
        END IF
      END IF
    END FOR
    RETURN max
  ELSE
    min  $\leftarrow \infty$ 
    FOR ALL o  $\in \mathcal{O}$  DO
      IF Előfeltétel(állapot, o) THEN
        új-állapot  $\leftarrow$  Alkalmaz(állapot, o)
        v  $\leftarrow$  Minimax-Érték(<A, kezdő, V, O>, új-állapot, mélység - 1, h)
        IF v < min THEN
          min  $\leftarrow$  v
        END IF
      END IF
    END FOR
    RETURN min
  END IF
END FUNCTION

```

A játék menetének összetétele nem teszi lehetővé, hogy egy általános minimax keresőalgoritmust használjunk. A játék során a játékosok köröket játszanak le. Ezek a körök alkotják a játszmák alapját. Egy ilyen kör addig tart, amíg valaki nem tudja ütni a lapot vagy elfogyott a 4 – 4 lap a játékosok kezéből. A keresőalgoritmus ezekre a körökre futtat le egy heurisztikus minimax algoritmust, a fában minden faelem egy-egy ilyen kört szimbolizál. Mivel heurisztikus keresést használunk, ezért minden körnek számítunk egy heurisztikus értéket egy újabb heurisztikus minimax kereső segítségével. Ez már a körön belül fog futni.

Természetesen ehhez meg kell adnunk az állapotterünkben egy másik végállapot függvényt, amely az egyes körök végét ellenőrzi. Ebben az összetett keresőben is felhasználjuk azt a tudást, hogy a faelem gyermekei közül csak azoknál fogjuk a fát tovább építeni, ahol a heurisztika a legmagasabb / legalacsonyabb volt attól függően, hogy a fa páros vagy páratlan szintjén állunk, a többi ággal szintén felesleges foglalkoznunk. A függelékben megtalálható a teljes kódrészlete ennek a keresőalgoritmusnak [1].

### 4.3. Heurisztika

Informált keresés esetén rendelkezünk egy kiértékelő függvénnyel, mely az egyes csomópontokra meghatározza, hogy mennyire visz a közelebb a célállapothoz, azaz egy játékállás „jóságértékét” határozza meg. Ez a függvény gyakran csak becslés, így pontatlan lehet. A heurisztika egy olyan, nem bizonyítható ötlet, amely a gyakorlati esetek többségében drasztikusan csökkenti a problémamegoldás erőforrásigényét. Az elfogadható heurisztikus függvény sohasem becsüli felül a cél eléréséhez szükséges tényleges költséget. Egy játékállás heurisztikus értéke minél nagyobb (kisebb), annál kedvezőbb (rosszabb) lesz a játékos számára a lépés. Egy heurisztikus függvényt a lehető legjobban kell meghatároznunk, mert a különböző heurisztikák nagyban befolyásolhatják a lépésajánlásokat. Játéktapasztalatok alapján a következő heurisztikus függvény lett implementálva:

```
public int heurisztika() {  
    if (Ajatekos.zsirDB > Bjatekos.zsirDB)  
        return 5*(Ajatekos.zsirDB - Bjatekos.zsirDB) + 10 * Ajatekos.zsirDB;  
    else  
        return -1*(3*(Bjatekos.zsirDB - Ajatekos.zsirDB)  
            + 5 * Bjatekos.zsirDB);  
}
```

A heurisztika számítási módszere:

- ha az A játékos több zsírt vitt el, akkor számára ez nagyon jó állást jelent, ezért a két játékos zsírjainak különbségét felszorozzuk öttel és ehhez hozzáadjuk az A játékos zsírjainak tízszeresét.

- ha ugyanannyi zsírja van a két játékosnak vagy a B játékos vitt több zsírt, akkor az A játékos számára nem a legjobb lépés, ezért a két játékos zsírjainak különbségét felszorozzuk hárommal és ehhez hozzáadjuk az B játékos zsírjainak tízszeresét.

A játék egyszerűségéből adódóan a játékosoknak csupán arra kell figyelniük, hogy mennyi zsírt visznek el, ezért lett a heurisztika is ilyen egyszerű képlettel meghatározva.

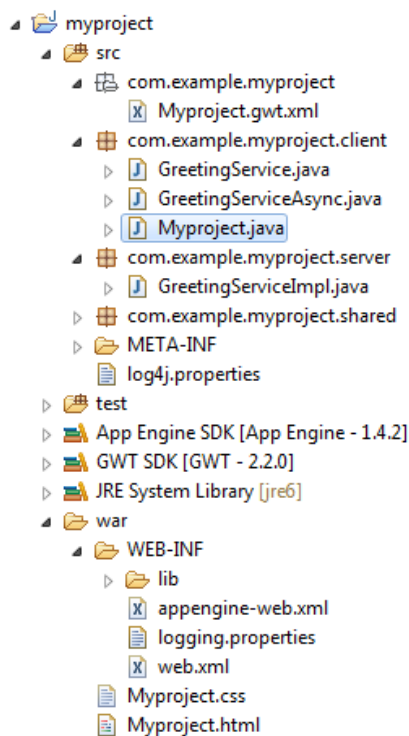
## **5. A játék online verziója**

Most már rendelkezésünkre áll a játék állapottere és hozzá keresőalgoritmus, ezután következhet a játéknak grafikus felülettel rendelkező verziójának elkészítése, amelyet web böngészőben a világom bárhol lehet játszani.

### **5.1. Google Web Toolkit**

A Google Web Toolkit (GWT) egy nyílt forráskódú fejlesztői eszköz, amely lehetővé teszi a fejlesztőknek, hogy komplex böngészőben futtatható web alkalmazásokat fejlesszenek és optimalizáljanak. A legnagyobb előnye a GWT-nek, hogy nem szükséges ismernünk a JavaScript nyelvet, ahhoz hogy minőségi AJAX web alkalmazást fejleszthessünk, ugyanis a kódolást és a hibakeresést Java nyelven végezhetjük el majd a GWT segítségével generálunk belőle egy optimalizált JavaScript kódot. Az így elkészült webes AJAX-os alkalmazást az összes modern böngésző támogatja.

A Google szolgáltató az Eclipse IDE-hez egy GWT bővítményt, amely jelentősen megkönnyíti a fejlesztést és a felmerülő hibák feltárását. A bővítményt kiegészíthetjük egy GWT Designer GUI szerkesztővel is, amely megkönnyíti a GUI elemek pozicionálást és attribútumainak beállítását. Ha minden szükséges eszközzel rendelkezünk, létrehozhatjuk az új GWT projektet.



2. ábra: új GWT projekt

Vegyük sorba miket kaptunk:

- `<modulnév>/src`: Ez a csomag tartalmazza a GWT modul definíciókat és a kezdeti applikációs fájlokat, ill. a `<modulnév>.gwt.xml` fájlt, amelyben a modulhoz kapcsolódó információk vannak tárolva.
  - `<modulnév>/src/client`: Ebben a csomagba lesznek a klienshez tartozó Java osztályok. Ezek lesznek JavaScript-re fordítva végső soron.
  - `<modulnév>/src/server`: Ebben a csomagban lesznek a szerverhez tartozó Java osztályok. A serveren ezek bájtkódba lesznek tárolva.
- `<modulnév>/test`: JUnit teszt könyvtár és a kezdő teszt osztályokat tartalmazza.
- `<modulnév>/war`: Tartalmazza a statikus erőforrásokat, amely akár szerver akár kliens oldalon is felhasználhatók. (képek, CSS fájlok, HTML fájlok stb.)

Vegyük szemügyre a modul xml fájlját (gwt.xml). Ebbe az állományba kell megnevezni az belépési pontot (EntryPoint) tartalmazó osztályt. A GWT modulnak legalább egy ilyen belépési ponttal rendelkeznie kell, például:

```
<entry-point class='szakdoga.zsir.client.Zsir' />
```

Elképzelhető hogy a modulnak több belépési pontja is van, ekkor minden egyes belépési pont sorra végre fog hajtódni. Legegyszerűbb megoldás, hogy csak egy modult hozunk létre. Most ezt a megoldást választjuk mi is. Ilyenkor <modulnév>.java osztályunk fogja implementálni az EntryPoint interfészt, amelynek lesz egy onModuleLoad() metódusa, például:

```
import com.google.gwt.core.client.EntryPoint;

public class Zsir implements EntryPoint{

    public void onModuleLoad() {}

}
```

Amikor a böngésző betölti a modulunkat, ez az onModuleLoad() metódus fog meghívódni.

Az alkalmazásunknak valamilyen módon kommunikálnia kell a web szerverrel, kéréseket küldi a szerver felé és üzeneteket / adatokat fogadni. A hagyományos web alkalmazásoknál minden egyes felhasználói interakciónál egy teljesen új HTML oldalt kell betöltenünk. Ezzel szembe az AJAX (Asynchronous JavaScript And XML) aszinkron hívások segítségével küldi és fogadja magát az adatokat ezáltal sokkal gyorsabb a válaszidő és az adatfolyam, így jóval kisebb sávszélességre van szüksége az alkalmazásoknak. A GWT is nyújt erre a problémára megoldást a GWT RPC (Remote Procedure Call) keretrendszer segítségével. A távoli eljáráshívások alapját a Java Servlet-ek képezik, és aszinkron módon kommunikálnak a szerverrel így egy AJAX alkalmazás minden előnyével rendelkeznek. Nézzük hogyan is valósítja meg a GWT az RPC hívásokat. Három dologra van szükségünk egy RPC létrehozásához:

1. Definiálni kell egy RemoteService interfészből kiterjesztett interfészt, amely egyben tartalmazza az összes RPC metódusunkat.
2. Létre kell hozni egy a RemoteServiceServlet osztályból kiterjesztett osztályt a szerver oldalon, amely implementálja az előbb létrehozott interfészt.
3. Definiálni kell egy aszinkron interfészt a szerverszolgáltatáshoz a kliens oldalra.

Szerencsére nekünk elég csak az 1. pontban leírt interfészt létrehozni a `<modulnév>/client` csomagba, a GWT a többit automatikusan legenerálja nekünk. Minden szolgáltatás implementáció tulajdonképpen egy servlet, de a `HttpServlet` helyett inkább a `RemoteServiceServlet`-et terjesztik ki. A `RemoteServiceServlet` automatikusan kezeli az adatok szerializációját, amelyek a kliens és a szerver között áramlanak. A szerializáció egy csomagolási eljárása, amely segíti objektum tartalmának mozgatását egyik alkalmazásból másik alkalmazásba vagy későbbi felhasználásra való eltárolását. Bármikor ha egy hálózaton objektumot szeretnénk küldeni a GWT RPC segítségével, akkor az objektumnak szerializálhatónak kell lennie, továbbá a metódus paramétereinek és a visszatérési értéknek is. Egy típus szerializálható és használható a szolgáltatás interfészben, ha a következők közül valamelyik állítás igaz rá:

- Minden primitív típus (`int`, `char`, `boolean` stb.) és a csomagoló osztályuk
- Szerializálható típusok egy tömbje
- Egy osztály szerializálható ha rendelkezik a következő három követelménnyel:
  - Implementálja vagy a `Java Serializable` vagy a `GWT IsSerializable` interfészt
  - Ha nem tartalmaz `final` és `transient` adattagot
  - Van egy alapértelmezett (argumentum nélküli) konstruktora bármilyen hozzáférési módosítóval (pl: `private Foo(){}` )

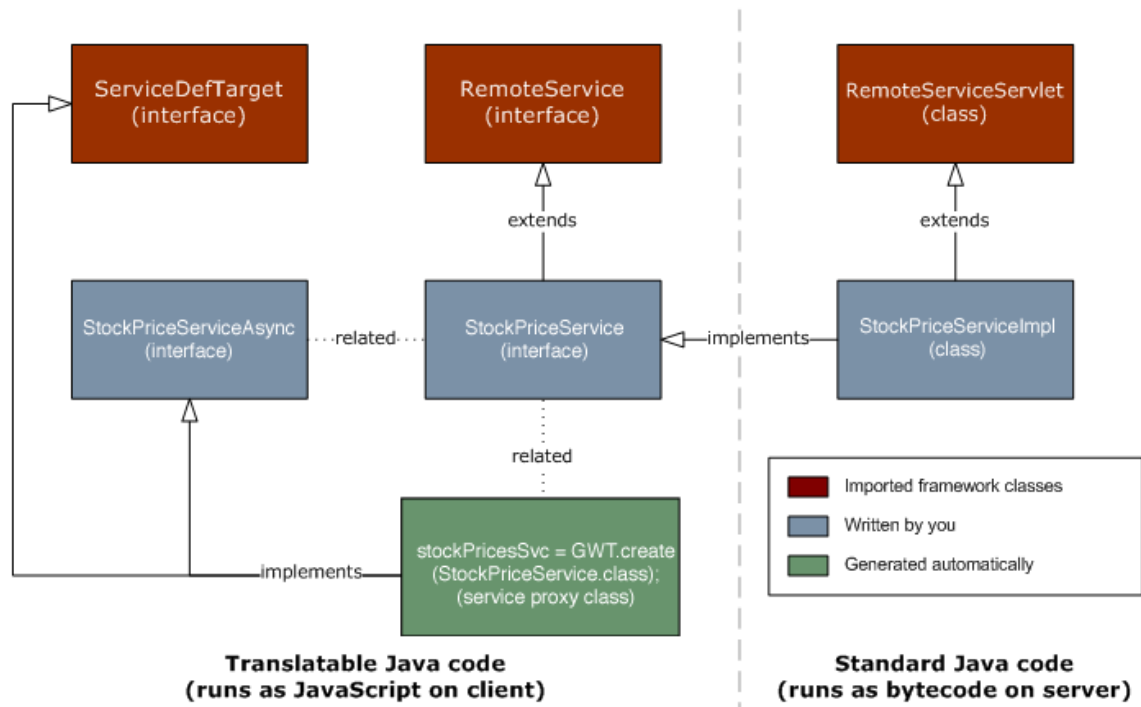
Ahhoz, hogy működjön a létrehozott szolgáltatás, a `web.xml` fájlban egy leírást kell adnunk arról, hogy a serveren hol lesz megtalálható a szolgáltatás, például:

```
<servlet>
  <servlet-name></servlet-name>
  <servlet-class></servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name></servlet-name>
  <url-pattern></url-pattern>
</servlet-mapping>
```

A `servlet-name` elembe megadjuk a szolgáltatás nevét, a `servlet-class` elembe megadjuk a server csomagban az interfészt implementáló osztályt, az `url-pattern` elembe pedig megadjuk a

modul nevét és az interfészben a `@RemoteServiceRelativePath(„relativPath”)` annotációval megadott elérési útvonalat.



3. ábra: A GWT RPC-hez szükséges Java komponensek hierarchiája és kapcsolata

A rendszerünkben a következő funkciókat szeretnénk megvalósítani:

- Egy adatbázis létrehozása a játékosok adatainak, pontjainak tárolásához.
- Egy regisztrációs rendszert, amelyen keresztül bárki beregisztrálhat a rendszerbe. Csak beregisztrált játékos játszhatnak a játékkal.
- Egy bejelentkező rendszert, ahol a regisztráció során megadott felhasználónév és jelszó segítségével be lehet lépni a rendszerbe. Csak belépés után lehet lehetséges játszani a játékkal.
- A játékosokhoz tartozó eredmények lekéréséhez valamilyen eszközt kell nyújtani.
- Bejelentkezés után a játék indítása vékony kliens technológiát használva.

A felhasználói felületünket két osztály fogja megvalósítani, a `LogInScreen` és a `GameTable`. Az első osztályunk fogja implementálni a regisztráció és bejelentkezés funkcióját, a második minden egyéb funkciót, amire szükségünk lesz.



## 5.2. Adatbázis

A regisztráció és a bejelentkezés kezeléséhez szükségünk lesz egy adatbázisra. A MySQL adatbázis rendszert fogjuk használni. A MySQL adatbázis a világ legnépszerűbb nyílt forrású adatbázisa, a magas teljesítményének, megbízhatóságának és könnyű használhatóságának köszönhetően. A világ számos nagy és gyorsan növekvő cégei, mint a Facebook, Google, Adobe is a MySQL adatbázisát használják. Segítségével könnyen tudunk Java-ban adatbázist kezelni, mivel a MySQL támogatja a JDBC szabványt. Az adatbázisunkat a szerver oldalon fogjuk létrehozni, egy táblánk lesz, amelyben tárolni fogjuk a játékosok bejelentkezéshez szükséges felhasználónevét és jelszavát, és a játékban elért eredményét.

A táblánkat a következő sql script futtatásával tudjuk létrehozni:

```
CREATE TABLE IF NOT EXISTS `dt_user` (  
  `USER_ID` int(11) NOT NULL auto_increment,  
  `USER_NAME` varchar(40) NOT NULL default '',  
  `USER_PASSWORD` varchar(32) NOT NULL,  
  `USER_POINT` int(11) NOT NULL default '0',  
  PRIMARY KEY (`USER_ID`),  
  KEY `USERNAME` (`USER_NAME`)  
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=0 ;
```

A *dt\_user* tábla oszlopai:

- *user\_id* : ez lesz a tábla elsődleges kulcsa, értéke minden sikeres regisztráció után egyel fog nőni, ezzel fogjuk a játékosokat azonosítani.
- *user\_name* : a játékos felhasználóneve, egyedinek kell lennie
- *user\_password* : a játékos jelszava, MD5 kódolás segítségével lesz eltárolva
- *user\_point* : a játékos megszerzett pontjai

### 5.2.1. Az adatbázis kapcsolat felépítése Java-ban

Szükségünk lesz egy MySQL connector-ra, amelyet könnyen beszerezhetünk a <http://dev.mysql.com/downloads/connector/> honlapról. Ezután a projekt tulajdonságainál a Java Build Path –et kiválasztva a Libraries fülön hozzáadjuk mysql-connector-java-5.1.15-bin.jar elérési útját.

Az adatbázis kapcsolat felépítéséhez egy saját `DataBaseConnection` osztályt fogunk használni. Öt attribútumot fog definiálni, amelyeknek az értékét a konstruktorban állítjuk be:

- Beállítjuk az adatbázis elérési útját `jdbc (jdbc:subprotocol:subname)` formátumban:

```
this.DataBase = "jdbc:mysql://localhost:3306/zsir";
```

- Az adatbázishoz csatlakozó felhasználó nevét és jelszavát:

```
this.username = "root";
```

```
this.password = "";
```

- Meg kell mondani a JVM-nek, hogy melyik drivert használja:

```
Class.forName("com.mysql.jdbc.Driver");
```

- Létrehozuk a kapcsolatot az adatbázissal

```
Connection conn = (com.mysql.jdbc.Connection)
```

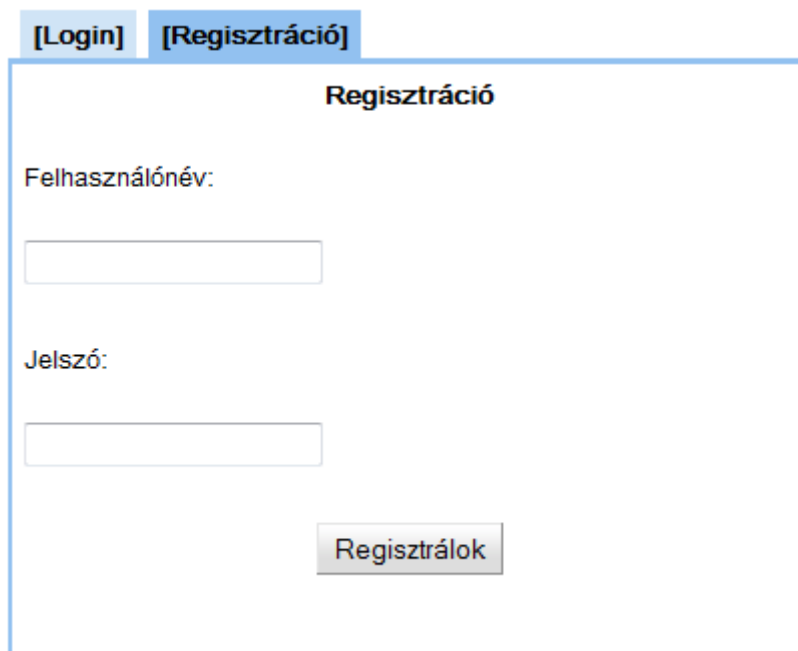
```
DriverManager.getConnection( DataBase, username, password );
```

- Létrehozunk egy úgynevezett `Statement`-et amely majd segít az sql lekérdezések futtatásában:

```
Statement stm = (Statement) conn.createStatement();
```

Egy metódusa lesz még az osztálynak, amely segítségével le tudjuk kérdezni a `Statement`-et.

### 5.3. Regisztráció

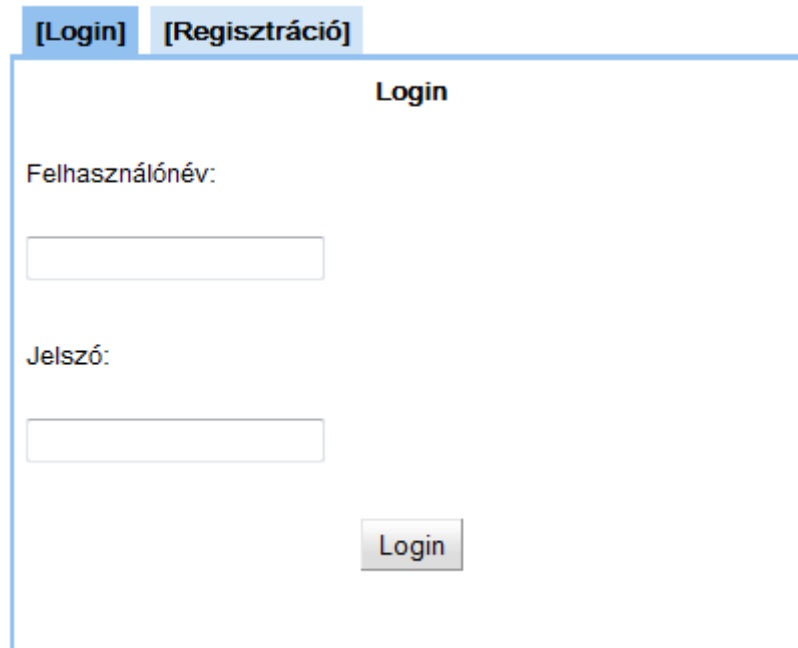


The image shows a software window titled "Regisztráció" (Registration). At the top, there are two tabs: "[Login]" and "[Regisztráció]", with the latter being the active tab. Inside the window, there are two text labels: "Felhasználónév:" (Username) and "Jelszó:" (Password). Below each label is a rectangular text input field. At the bottom right of the window is a button labeled "Regisztrálok" (Register).

4. ábra: Regisztrációs ablak

A regisztrációs ablakot egy `TabLayoutPanel` egyik fülére helyezzük el, két mezővel: felhasználónév és jelszó. A regisztráció során ellenőrizni fogjuk, hogy a megadott felhasználó név létezik-e már az adatbázisban. Ha igen, akkor nem engedjük regisztrálni ilyen névvel és az RPC híváson keresztül egy `UserAlreadyExistsException` kivétellel tudatjuk ezt a rendszerrel. Ha még nem létezik, akkor hozzáadjuk a `dt_user` táblához a felhasználót a hozzá kapcsolódó adatokkal. A felhasználó biztonsága érdekében nem a valódi jelszavát, hanem annak az `MD5Crypt` osztályunk `crypt()` metodusa által visszaadott MD5-ben lekódolt képét fogjuk letárolni, így ha valakinek sikerülne az adatbázishoz hozzáférnie, nem tudná kinyerni a jelszavát. Az MD5 (Message-Digest algorithm 5) egy 128 bites, kódolási algoritmus, amely bármilyen adatból – függetlenül a méretétől, vagy a típusától – egy 32 karakter hosszú hexadecimális hasht eredményez. Az adatbázisunkban a jelszó oszlop ezért 32 karakter hosszú. A kódolás egyirányú, így nem lehet visszafejteni. Éppen emiatt biztonságos, és kiválóan alkalmas eredetiség ellenőrzésre.

## 5.4. Bejelentkezés



The image shows a software window titled 'Login'. At the top, there are two tabs: '[Login]' and '[Regisztráció]'. The '[Login]' tab is active. Inside the window, the title 'Login' is centered at the top. Below it, there are two labels: 'Felhasználónév:' and 'Jelszó:'. Each label is followed by a text input field. At the bottom right of the window, there is a button labeled 'Login'.

5. ábra: Bejelentkező ablak

A bejelentkezés ablakot egy `TabLayoutPanel` másik fülére helyezzük el, két mezővel: felhasználónév és jelszó. A bejelentkezés során megvizsgáljuk a felhasználó által megadott felhasználónév és jelszó helyes-e. A jelszót átkódoljuk az MD5-ös képébe és ezzel az értékkel fogjuk összehasonlítani az eltárolt jelszót. Ha az összehasonlítás során a felhasználónév vagy a jelszó nem egyezik meg, akkor nem engedjük a játékost belépni a rendszerbe és erről értesítjük a rendszert az RPC híváson keresztül egy `WrongUnOrPwException` kivétellel. Ha a felhasználónév és jelszó helyes, akkor engedélyezzük a játékosnak a belépést a rendszerbe és elindítjuk a játékot.

## 5.5. Az aszinkron hívások a regisztrációnál és bejelentkezésnél

Ebben a részben megnézzük, hogyan lehet megvalósítani az aszinkron hívásokat az 5.1-es pontban leírtak alapján. Mindkét funkciónál a hívást hozzákapcsoljuk egy gomb eseménykezelőjéhez. Nézzük meg részletesebben hogyan is néz ki ez implementációs szinten.

### 5.5.1. RPC megvalósítása a bejelentkezésnél

Hozzuk létre a bejelentkezést kezelő interfészt. Egy metódusa lesz, amely két paraméterrel rendelkezik: felhasználónév és jelszó, és egy saját User típussal fog visszatérni.

```
@RemoteServiceRelativePath("loginCheck")
public interface LoginService extends RemoteService {
    User login(String username, String password) throws WrongUnOrPwException;
}
```

Hozzuk létre az előző interfésznek az aszinkron változatát, amely ugyanezzel a metódussal rendelkezik, viszont kibővül a paraméter listája és visszatérési értéke void lesz.

```
public interface LoginServiceAsync {
    void login(String username, String password, AsyncCallback<User> callback);
}
```

Hozzuk létre az interfészünket implementáló osztályt szerver oldalon és implementáljuk a login metódust.

```
public class LoginServiceImpl extends RemoteServiceServlet implements
LoginService {

    @Override
    public User login(String username, String password) throws
WrongUnOrPwException {
        ...
    }
}
```

A login metódus implementálása során a következőket végezzük el:

- Megvizsgáljuk, hogy a játékos a bejelentkezéshez szükséges felhasználónév és jelszó mezőt kitöltötte-e

```
if (username.isEmpty() || password.isEmpty()) throw new
WrongUnOrPwException();
```

- Felépítjük a kapcsolatot az adatbázissal

```
DataBaseConnection dbc = new DataBaseConnection();
Statement stm = dbc.getStatement();
```

- Egy SQL lekérdezés segítségével megvizsgáljuk, hogy helyes felhasználónevet és jelszót adott-e meg a játékos

```
ResultSet qrRec = stm.executeQuery("SELECT USER_ID FROM dt_user WHERE
USER_NAME= '" + username + "' AND USER_PASSWORD = '" +
MD5Crypt.crypt(password) + "'");
```

```
if ( qrRec.next() ) ID = qrRec.getInt("USER_ID");
else throw new WrongUnOrPwException();
```

- Ha minden rendben ment visszaadjuk a felhasználót

```
return new User(username, ID);
```

Meg kell adnunk a szolgáltatást leíró bejegyzéseket a szerver számára, ehhez módosítanunk kell a web.xml fájlt a következő bejegyzéssel:

```
<servlet>
  <servlet-name>loginServiceImpl</servlet-name>
  <servlet-class>szakdoga.zsir.server.LoginServiceImpl</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>loginServiceImpl</servlet-name>
  <url-pattern>/zsir/loginCheck</url-pattern>
</servlet-mapping>
```

Definiáljuk a szolgáltatásunk proxy osztályát a LogInScreen osztályban.

```
public LoginServiceAsync loginSvc = GWT.create(LoginService.class);
```

A bejelentkező szolgáltatás készen van, már csak annyi dolgunk maradt, hogy igénybe is vegyük a szolgáltatást. Ehhez létrehozunk egy gombot az ablakon, amelyre rákattintva elindul egy eseménykezelő, amelynek az lesz a feladata, hogy elindítsa a bejelentkező szolgáltatást.

```
Button loginButton = new Button("Login");
loginButton.addClickHandler(new ClickHandler() {

    @Override
    public void onClick(ClickEvent event) {

        if (loginSvc == null){
            loginSvc = GWT.create(LoginService.class);
        }

        AsyncCallback<User> callback = new AsyncCallback<User>() {

            @Override
            public void onSuccess(User result) {
```

```

        RootPanel.get("loginScreen").clear();
        new GameTable(result.getID());
    }

    @Override
    public void onFailure(Throwable caught) {
        System.out.println("kivétel a loginba");
        if (caught instanceof WrongUnOrPwException) {
            pwTextBox.setText("");
            Window.alert("Hibás felhasználónév vagy jelszó");
        }
    }
};

loginSvc.login(userNameTextBox.getText(), pwTextBox.getText(),
    callback);
});

```

Vegyük sorba mit látunk a fenti kódrészletben:

- Létrehozunk egy `Button` típusú gombot, amelyhez egy `ClickHandler` eseménykezelőt definiálunk. Ez azt az eseményt fogja kezelni, amikor a felhasználó ráklikkel a gombra, és ekkor az `onClick` metódus fog meghívódni.
- Ellenőrizzük, hogy a szolgáltatás definiálva van-e, ha nincs akkor megtesszük.
- Kezeljük a szerver aszinkron válaszát az `AsyncCallback<T>` interfész segítségével. A `<T>` generikus paraméter mindig a szervertől kapott válasz típusát jelöli, ebben az esetben `User`. Az aszinkron hívásunk sikerességtől függően két metódus hívódhat meg:
  - `void onSuccess(T result)` : ez a metódus akkor fog meghívódni, ha a szerveren nem következett be semmilyen váratlan hiba és a kérésünk sikeresen lefutott. A `T` típus pedig a kéréshez megadott metódus visszatérési típusa lesz. A metódusban letöröljük az ablakból a bejelentkező felületet és a `new GameTable(result.getID())` hívással elindítjuk a játékot.
  - `void onFailure(Throwable caught)` : ez a metódus akkor fog meghívódni, ha a szerveren történt valamilyen váratlan hiba és nem lehetett a kérésünket teljesen feldolgozni, paraméterül pedig a szerveren bekövetkeztetett kivételt kapja meg.
- Hívjuk meg a távoli eljárást.

## 5.5.2. RPC megvalósítása a regisztrációnál

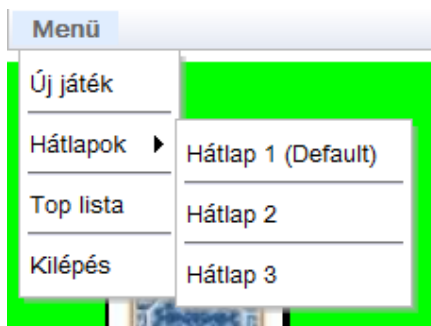
A regisztrációhoz tartozó RPC megvalósításánál hasonlóképpen járunk el, mint a bejelentkezésnél. A legfőbb különbség a szolgáltatás implementációjában lesz. Az adatbázis kapcsolat létrehozása után megvizsgáljuk, hogy létezik-e a megadott felhasználónév a listában.

```
ResultSet qrRec = stm.executeQuery("SELECT USER_ID FROM dt_user WHERE  
USER_NAME= '" + username + "'");
```

Ha létezik, akkor egy `UserAlreadyExistException` kivétellel jelezzük a rendszernek. Ha nincs ilyen nevű felhasználó még az adatbázisban, akkor hozzáadjuk a felhasználónevet és a jelszó MD5-ben lekódolt képét.

```
stm.executeUpdate("INSERT INTO dt_user (USER_NAME,USER_PASSWORD) values (  
'" + username + "', '" + MD5Crypt.crypt(password) + "'");
```

## 5.6. Főmenü



6. ábra : A játék menüje

A játékhoz készítünk egy egyszerű menürendszert a legfontosabb szolgáltatásokkal:

- *Új játék*: a játékos, ha úgy kívánja, kezdhet egy új játékot
- *Hátlapok*: három féle hátlap közül választhat a játékos
- *Top lista*: itt lehet lekérdezni a legeredményesebb játékosok listáját
- *Kilépés*: a játékos, ha úgy kívánja, kiléphet a játékból, ekkor a bejelentkező ablakhoz fog visszalépni

A főmenü csak akkor lesz elérhető, ha elindítunk egy új játékot.



## 5.7. Játéklablak

A játéklablakot a `GameTable` osztály fogja megvalósítani. Sikeres bejelentkezés után megjelenik egy `DialogBox` ablak, amelyen ki tudjuk választani `RadioButton`-ok segítségével, hogy mi vagy a gép kezdjen. A „Start!” megnyomásával meghívódik a `playGame()` metódus, amely elindítja a játékot. Először példányosítjuk a `ZsirAllapotTer` osztályt, amellyel létrehozuk az állapotterünket, utána kirajzoljuk a játékasztalt, a játékosok kártyalapjait és a paklit. Végül a játék futását egy végtelen ciklussal szeretnénk megvalósítani, amiből csak akkor lépünk ki, ha vége a játéknak, azaz elértünk egy célállapotot és szeretnénk időzíteni, hogy másodpercenként vizsgálja meg, hogy milyen változások történtek az állapotterünkben. A GWT `Timer` osztálya nyújt erre egy technikát, amely hasonló a `java.util.Timer` osztályhoz, viszont ennek egy jóval egyszerűbb és biztonságosabb változatát valósítja meg. Működése egyszerű, csupán annyi a dolgunk, hogy létrehozunk egy példányát, amelyben implementáljuk a `run()` metódusát. Majd meghívjuk a `schedule()` vagy a `scheduleRepeating()` metódust, amelyek segítségével tudjuk megvalósítani az ismétlődést.

```
Timer gameTimer = new Timer() {  
  
    @Override  
    public void run() {  
        ...  
    };  
  
gameTimer.scheduleRepeating(REFRESH_INTERVAL);
```

A `REFRESH_INTERVAL` egy `int` típusú konstans, amelynek értéke legyen 1000, ha egy másodpercre szeretnénk állítani az ismétlődést.

A GWT a kártyalapjainkat `Widget`-ekként fogja megjeleníteni. A `Widget` egy speciális GWT típus, amely a legtöbb UI objektum őssztálya. A `Widget`-ekhez eseményeket is tudunk hozzárendelni. A `GameTable` osztály implementálni fog három eseménykezelőt: `MouseOutHandler`, `MouseOverHandler`, `ClickHandler`. Ezek segítségével fogjuk kezelni a játék működésének egy részét, mint például a lap kiválasztást és a kiválasztott lap lehívását.

A `MouseOverHandler` eseménykezelőhöz úgy fogjuk implementálni az `onMouseOver(MouseOverEvent event)` metódust, hogy az épp aktuális widgetnek (azaz kártyalapnak) az `y` koordinátáját fogjuk megnövelni.

A `MouseOutHandler` eseménykezelőhöz úgy fogjuk implementálni az `onMouseOut(MouseOverEvent event)` metódust, hogy az épp aktuális widgetnek (azaz kártyalapnak) az `y` koordinátáját vissza fogja állítani alapértelmezett értékre.

Ezzel a kis trükkel el tudjuk érni azt a hatást, hogy amikor rámutatunk egy kártyalapra, akkor az kiemelkedjen a többi közül, és amikor elvisszük róla az egérmutatót „visszacsúszik” a helyére.

A `ClickHandler` eseménykezelőhöz úgy fogjuk implementálni az `onClick(ClickEvent event)` metódust, hogy az épp aktuális widgetnek (azaz kártyalapnak) megfelelően beállítunk egy változónak egy megfelelő értéket, amely segítségével majd a `gameTimer`-ben alkalmazni fogjuk a megfelelő hív operátort az állapotterünkön.

Egy teendőnk maradt még, amikor véget ér a játék ki kell számolnunk a játékos által szerzett pontokat és frissítenünk kell ezt az adatbázisban. A pontszámítás a következőképpen történik:

- Ha nyert a játékos, akkor a megszerzett zsírjainak számának kétszeresét hozzáadjuk az eddig megszerzett pontjaihoz.
- Ha döntetlen lett, akkor néggyel növeljük az eddig megszerzett pontjait.
- Ha vesztett a játékos, akkor az ellenfél által szerzett zsírok számát levonjuk az eddig megszerzett pontjaiból.

A pontszám frissítéséhez egy RPC-t fogunk használni. Az implementációjában kiszámoljuk a pontszám értéket, amennyivel változtatni kell majd a meglévő pontokat, létrehozuk az adatbázis kapcsolatot és a `stm.executeUpdate("UPDATE dt_user SET USER_POINT = USER_POINT + " + zsirDB + " WHERE USER_ID = " + ID);` SQL scripttel módosítjuk az adatbázist.

A klienseket két nagy csoportba lehet sorolni. Vastag és vékony kliens. A csoportokhoz való tartozást nehéz egyértelműen definiálni. Vastag kliens esetén minden adatfeldolgozást és megjelenítést a kliens oldalon végeznénk, és nem használnánk fel a szerver erőforrásait.

Ennek az a legnagyobb hátránya, hogy nem minden kliens rendelkezik a megfelelő erőforrással ennek végrehajtásához. Vékony kliens esetén minden számolást és adatfeldolgozást a szerver oldalon végeznénk kihasználva a szerver erőforrásait és a kliens csak a megjelenítésért lenne felelős. Ez a megvalósítási módszer kezdte el terjedni napjainkban, viszont ha a szerverhez túl sok kliens csatlakozik, akkor akadozhat a kiszolgálás, ezért korlátokat kell bevezetni, pl. a csatlakozható kliensek számában. Mi egy hibrid megoldást választottunk, az állapotterünket és a játékhoz szükséges adatokat kliens oldalon tároljuk, módosítjuk, és a kliensünk végzi a megjelenítést, viszont a keresőnket áthelyeztük szerver oldalra, hogy a bonyolult számolásokat és a hozzá szükséges tárigény terhét levegyük a kliensről.

## 5.8. Telepítés web szerverre

Eljutottunk odáig, hogy a játék teljes egészében készen van, fel van építve az adatbázis kapcsolat és a fejlesztés során a GWT segítségével ún. „Hosted Mode”-ban dolgoztunk, ami azt jelentette, hogy a GWT saját Jetty web szerverének segítségével jelenítettük meg az alkalmazásunkat a web böngészőben. Mivel azt szeretnénk, hogy ne csak mi tudjuk használni az elkészített alkalmazásunkat, hanem a világon bárki elérhesse, keresnünk kell egy eszközrendszert, amely segít nekünk ebben. Az egyik alternatíva a Google Apps Engine (GAE) lehetne, egy gyors regisztrálás után már fel is tölthetnénk az elkészített alkalmazásunkat ide és a világ már használhatná is, viszont a GAE nem támogatja a JDBC-t, ami számunkra problémát jelent. Egyik megoldás lehetne, hogy nem használjuk a JDBC-t, de egyszerűbb lesz, ha keresünk egy másik web szervert. A választásunk a WampServer programcsomagra esett. A WampServer egy Windows-os web fejlesztő környezetet biztosít nekünk. Segítségével könnyedén tudunk létrehozni web alkalmazásokat Apache, PHP és MySQL adatbázissal. Továbbá rendelkezésünkre bocsájtja a PHPMyAdmin szolgáltatást, amely segítségével könnyedén tudjuk kezelni az adatbázisainkat. A szoftver csomag telepítése nagyon egyszerű, csupán követni kell a telepítési instrukciókat (next, next, ok...) és telepítés után már működésre készen áll a szerver anélkül, hogy bármilyen beállítást végeznünk kellene rajta. Miután elindítottuk a WampServer szolgáltatást az óra mellett megjelenik egy trayicon, amely ha zöldre vált, akkor tudjuk, hogy minden rendben működik és a trayicon-ra rákattintva tudjuk kezelni a beállításokat is.

Most már készen áll a web szerverünk, viszont az Apache nem támogatja a Java kódokat, ezért szükségünk lesz még egy Tomcat szerverre is, amelyet az Apache helyett fogunk használni. A telepítés után a Tomcat sem igényel különösebb beállításokat, így rögtön fel is tölthetjük az alkalmazásunkat a szerverre a következő módon:

1. A Tomcat telepítése könyvtárában a webapps alkönyvtárban létrehozunk egy mappát a projektünk nevével, esetünkben legyen a mappa neve: zsir
2. Ebbe a mappába másoljuk bele a projektünk war könyvtárában található összes fájlt könyvtárakkal együtt

Hogy működni is tudjon, a szerver be kell állítanunk a Java környezeti változóit, ha korábban nem tettük még meg:

- Vegyünk fel egy JAVA\_HOME környezeti változót, amelynek értéke mutasson a jdk1.6... könyvtárra (amelyiknek a bin közvetlen alkönyvtára)  
pl.: C:\Program Files\Java\jdk1.6.0\_18\
- Adjuk hozzá a PATH környezeti változóhoz a jdk.../bin könyvtárat (amelyben a java, javac stb. fájlokat látjuk)  
pl.: C:\Program Files\Java\jdk1.6.0\_18\bin

A Tomcat szerver a bin könyvtárban található startup.bat állománnyal indíthatjuk el és a shutdown.bat –al állíthatjuk le.

Ezután elvégezhetünk néhány tesztet, hogy minden rendben működik-e:

- Nyissuk meg a böngészőben a <http://localhost/> címet. Ekkor láthatjuk a WampServer kezdőlapját.
- Nyissuk meg a böngészőben a <http://localhost/phpmyadmin/> címet. Ekkor láthatjuk a PHPMyAdmin szolgáltatás kezelőfelületét.
- Ha elindítottuk a Tomcat szervert és a böngészőben megnyitjuk a <http://localhost:8080> címet, látni fogjuk a Tomcat kezdőlapját.
- Végezetül, ha megnyitjuk a <http://localhost:8080/zsir/> címet, akkor elindul az alkalmazásunk.

## 6. Összefoglalás

Sikeresen megalkottuk a klasszikus zsír kártyajáték online működő és használható változatát. A kevés játékszabály ellenére nagy hangsúlyt kellett fektetnünk a gép „gondolkodásának” megvalósítására, így egy igen összetett és bonyolult szabályrendszert sikerült létrehozunk arra, hogy a gépnek mikor mit érdemes lépnie. Ezeket a saját játéktapasztalataink alapján írtuk össze és implementáltuk. Talán egy pontosabb heurisztika vagy jobban megkonstruált keresőalgorithmus segítségével komolyabb ellenfelet tudnánk biztosítani a játékosok számára. Az általunk használt eszközök kiválóan teljesítették a tőlük elvártakat, segítségükkel könnyedén és hatékonyan tudtuk fejleszteni az alkalmazásunkat lépésről lépésre. A fejlesztés alapeszköze a GWT volt, hiszen web alkalmazás fejlesztése volt a cél. A GWT fejlesztői rendszere tökéletesen bevált, sikerült kezdő JavaScript tudás nélkül is összetett és bonyolult weblapot létrehozunk, amire egyébként nem lett volna lehetőségünk.

A fejlesztési folyamat során tudomást szereztünk néhány hibáról és hiányosságról is. Hatékonyabb biztonságot érhetünk el, ha a jelszó MD5-re való lekódolását kliens oldalon végeznénk el, de a GWT nem tud JavaScript kódot készíteni a Java.Security csomagból. Célszerű lenne a teljes algoritmust lekódolni Java-ban kliens oldalon, de idő és forrásanyag hiányában erre nem került sor. Észrevettünk egy apró grafikai hibát is, amikor a kártyalap határára mozgatjuk az egeret, el lehet érni, hogy a kártyalap „bepörögjön”. Ez sajnos a GWT Widgetek eseménykezeléséből adódik, ugyanis rosszul határozza meg a határvonalakat.

Remélem sikerült felkelteni az olvasó érdeklődését akár a mesterséges intelligencia tudományágára, akár a GWT nyújtotta lehetőségek használatára. A játékhoz pedig jó szórakozást és kellemes időtöltést kívánok!

## Függelék

A speciális heurisztikus minimax keresőalgorithmus forráskódja. [1]

```
public class MiniMaxKereso {

    public ZsirAllapotTer aktAllapot;
    public int korlat;

    public List<Operator> felhasznaltOperatorok =new ArrayList<Operator>();

    public MiniMaxKereso(ZsirAllapotTer allapot, int korlat) {
        this.aktAllapot = new ZsirAllapotTer((ZsirAllapotTer) allapot);
        this.korlat = korlat;
    }

    public Operator keres(){

        Operator operator = null;
        int ertekek;
        int max = Integer.MIN_VALUE;

        for (Operator op : aktAllapot.operatorok){
            if (aktAllapot.elofeltetel(op)){
                ZsirAllapotTer ujAllapot = new ZsirAllapotTer(
                    (ZsirAllapotTer) aktAllapot.alkalmaz(op));
                ertekek = miniMax(ujAllapot, korlat,
                    ujAllapot.heurisztika());
                if (ertekek >= max){
                    max = ertekek;
                    operator = op;
                }
            }
        }
        return operator;
    }

    public int miniMax(ZsirAllapotTer jatszma, int korlat,
        int heurisztika){

        if (jatszma.celAllapot() || korlat == 0) return heurisztika;

        int melyseg = 7;
        int jatszmaErteke = 0;

        if (jatszma.aktualisJatekos.equals(jatszma.Ajatekos)){
            int max = Integer.MIN_VALUE;
            ZsirAllapotTer ujJatszma = new ZsirAllapotTer(jatszma);
            felhasznaltOperatorok = new ArrayList<Operator>();
            jatszmaErteke = jatszmaErtek(ujJatszma, melyseg,
                ujJatszma.heurisztika());
        }
    }
}
```

```

        if (jatszmaErteke >= max){
            max = jatszmaErteke;
            for (int i = felhasznaltOperatorok.size()-1; i >= 0; i--){
                jatszma = new ZsirAllapotTer(
                    (ZsirAllapotTer)jatszma.alkalmaz(felhasznaltOperatorok.get(i)));
            }
        }

        for (Operator op : jatszma.operatorok){
            if (jatszma.elofeltetel(op)){
                ZsirAllapotTer ujAllapot = new ZsirAllapotTer(
                    (ZsirAllapotTer)jatszma.alkalmaz(op));
                jatszmaErteke = miniMax(ujAllapot, korlat - 1,
                    ujAllapot.heurisztika());
                if (jatszmaErteke >= max){
                    max = jatszmaErteke;
                }
            }
        }

        return max;
    }
}

else {
    int min = Integer.MAX_VALUE;
    ZsirAllapotTer ujJatszma = new ZsirAllapotTer(jatszma);
    felhasznaltOperatorok = new ArrayList<Operator>();
    jatszmaErteke = jatszmaErtek(ujJatszma, melyseg,
        ujJatszma.heurisztika());

    if (jatszmaErteke <= min){
        min = jatszmaErteke;
        for (int i = felhasznaltOperatorok.size()-1; i >= 0; i--){
            jatszma = new ZsirAllapotTer(
                (ZsirAllapotTer)jatszma.alkalmaz(felhasznaltOperatorok.get(i)));
        }
    }

    for (Operator op : jatszma.operatorok){
        if (jatszma.elofeltetel(op)){
            ZsirAllapotTer ujAllapot = new ZsirAllapotTer(
                (ZsirAllapotTer)jatszma.alkalmaz(op));
            jatszmaErteke = miniMax(ujAllapot, korlat - 1,
                ujAllapot.heurisztika());
            if (jatszmaErteke <= min){
                min = jatszmaErteke;
            }
        }
    }

    return min;
}
}

```

```

public int jatszmaErtek(ZsirAllapotTer allapot, int melyseg,
                        int heurisztika){

    if (allapot.jatszmaVege() || melyseg == 0) return heurisztika;

    int ertekek = 0;
    Operator operator = null;

    if (allapot.aktualisJatekos.equals(allapot.Ajatekos)){
        int max = Integer.MIN_VALUE;
        for (Operator op : allapot.operatorok){
            if (allapot.elofeltetel(op)){
                ZsirAllapotTer ujAllapot = new ZsirAllapotTer(
                    (ZsirAllapotTer)allapot.alkalmaz(op));
                ertekek = jatszmaErtek(ujAllapot, melyseg - 1,
                                       ujAllapot.heurisztika());
                if (ertekek >= max){
                    max = ertekek;
                    operator = op;
                }
            }
        }
        if (!felhasznaltOperatorok.contains(operator))
            felhasznaltOperatorok.add(operator);

        return max;
    }
    else {
        int min = Integer.MAX_VALUE;
        for (Operator op : allapot.operatorok){
            if (allapot.elofeltetel(op)){
                ZsirAllapotTer ujAllapot = new ZsirAllapotTer(
                    (ZsirAllapotTer)allapot.alkalmaz(op));
                ertekek = jatszmaErtek(ujAllapot, melyseg - 1,
                                       ujAllapot.heurisztika());
                if (ertekek <= min){
                    min = ertekek;
                    operator = op;
                }
            }
        }
        if (!felhasznaltOperatorok.contains(operator))
            felhasznaltOperatorok.add(operator);

        return min;
    }
}
}

```



## Irodalomjegyzék

- [1] Google Web Toolkit: <http://code.google.com/intl/hu-HU/webtoolkit/overview.html>
- [2] GWT Communicate with a Server: <http://code.google.com/intl/hu-HU/webtoolkit/doc/trunk/DevGuideServerCommunication.html#DevGuideSerializableTypes>
- [3] GWT: <http://java.ociweb.com/mark/programming/GWT.html>
- [4] WampServer: <http://www.wampserver.com/en/>
- [5] MySQL Connector/J: <http://dev.mysql.com/downloads/connector/j/>
- [6] Tomcat: <http://tomcat.apache.org/>
- [7] WAMP using Tomcat, PHP, MySQL: <http://tomcatenv.awardspace.com/>
- [8] MD5: <http://hu.wikipedia.org/wiki/MD5>
- [9] MD5 használata: <http://www.sourceformat.com/obfuscate-code-java-2.htm>
- [10] Dr. Várterész Magda: Mesterséges intelligencia 1:  
<http://www.inf.unideb.hu/~varteres/mi1folia/fofiafo.pdf>
- [11] Kétszemélyes, teljes információjú játékok:  
<http://www.inf.unideb.hu/~varteres/mi/part5/jatek.htm>
- [12] Informált keresési stratégiák:  
[https://wiki.sch.bme.hu/bin/view/Infoalap/MIOsszefoglalo?CGISESSID=fa51c2233a5d1bfab8c5d0a3e3dc4a56#Informált\\_keresési\\_stratégiák](https://wiki.sch.bme.hu/bin/view/Infoalap/MIOsszefoglalo?CGISESSID=fa51c2233a5d1bfab8c5d0a3e3dc4a56#Inform%C3%A1lt_keres%C3%A9si_strat%C3%A9gi%C3%A1k)
- [13] Jeszenszky Péter: Minta állapottér-reprezentáció:  
<http://www.inf.unideb.hu/~jeszy/download/mestint/blocks.pdf>
- [14] A játék fogalma: <http://www.tankonyvtar.hu/konyvek/magyar-neprajz/magyar-neprajz-jatek>

## **Köszönetnyilvánítás**

Szeretném köszönetemet nyilvánítani Csomor Benő témavezetőmnek, barátaimnak megértésükért, hasznos és építő jellegű tanácsaikért és a DotA partikért. Külön köszönetet érdemel Hucker Dávid és Sarkadi Sándor, akik segítettek a játék tesztelésében és a felhasznált fejlesztői eszközök megismerésében.