**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Programozási nyelvek és Fordító programok Tanszék**

Sakkdolgozat

Szerző: Témavezető:

Gábris Attila Dr. Kozsik Tamás

Programtervező informatikus BSc. Phd., docens

**Budapest, 2023.**

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: **Gábris Attila**

Neptun kód: **GGLOE7**

Képzési adatok:

Szak: **programtervező informatikus, alapképzés (BA/BSc/BProf)**

Tagozat **: Nappali**

**Belső témavezetővel rendelkezem**

*Témavezető neve:* ***Kozsik Tamás***

***munkahelyének neve, tanszéke:*** *ELTE IK, Programozási nyelvek és Fordítóprogramok Tanszék*  ***munkahelyének címe:*** *1117, Budapest, Pázmány Péter sétány 1/C.*  ***beosztás és iskolai végzettsége:*** *Docens, PhD*

A szakdolgozat címe: **Sakkdolgozat**

A szakdolgozat témája:

***(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását )***

Bár már 1996–ban elkészült a Deep Blue, mely képes volt az akkori világbajnokot (Garry Kaszparovot) térdre kényszeríteni, és azóta ezt már messze meghaladta a tudomány, én mégis érdekes kihívásnak tartom egy saját sakk mesterséges intelligencia létrehozását.

Az alapkoncepcióm egy grafikus felhasználói felület létrehozása, mely lehetőséget biztosít a humán játékosnak tudása összemérésére a géppel szemben. A felületet továbbá könnyen kezelhetővé és esztétikailag megkapóvá fogom megtervezni.

Lehetőség nyílik majd az adott állás elmentésére, vagy egy már elmentett állás betöltésére. Továbbá a tökéletes játékélményt az időmérőként is funkcionáló sakkóra fogja biztosítani. Mindezek mellett pedig a játékosnak lehetősége lesz döntetlent ajánlani- , feladni- a partit és új játékot kezdeni egy befejezett után.

A program működésének alapjaként különböző pozíció kiértékelő metódusokat fogok használni, mint például a minimax algoritmus. A munkát valószínűleg Java nyelven fogom elkészíteni.

**Budapest, 2022. 11. 21.**

Tartalomjegyzék

[1. Bevezetés 1](#_Toc154770996)

[2. Felhasználói dokumentáció 2](#_Toc154770997)

[2.1. Célközönség 2](#_Toc154770998)

[2.1.1. Igényelt kapacitás 2](#_Toc154770999)

[2.2. Üzembe helyezés 2](#_Toc154771000)

[2.2.1. Java letöltése 3](#_Toc154771001)

[2.2.2. Futtatási környezet (IDE) letöltése 4](#_Toc154771002)

[2.2.3. Futtatás 5](#_Toc154771003)

[2.3. Felhasználói útmutató, felhasználói utak 6](#_Toc154771004)

[3. Fejlesztői dokumentáció 9](#_Toc154771005)

[3.1. Megoldási terv 9](#_Toc154771006)

[3.1.1. Grafikus interfész terv 11](#_Toc154771007)

[3.1.2. Komponensek 13](#_Toc154771008)

[3.1.3. A komponensek feladatai 15](#_Toc154771009)

[3.2. Megvalósítás 15](#_Toc154771010)

[3.2.1. FEN String avagy a fő kommunikációs eszköz 15](#_Toc154771011)

[3.2.2. A GUI megvalósítása 17](#_Toc154771012)

[3.2.3. A Model és a Megjelenítés kapcsolata 18](#_Toc154771013)

[3.2.4. Az AI alapkoncepciója 18](#_Toc154771014)

[3.2.5. Osztály architektúra 18](#_Toc154771015)

[3.3. Tesztelés 18](#_Toc154771016)

[4. Összefoglalás és további fejlesztési lehetőségek 18](#_Toc154771017)

[4.1. Továbbfejlesztési lehetőségek 18](#_Toc154771018)

[5. Irodalomjegyzék 18](#_Toc154771019)

# Bevezetés

Ahogyan az a témabejelentőben is olvasható, egy sakk program elkészítésére vállalkoztam. Ebben elsősorban a feladatban rejlő informatikai kihívás vonzott, másodsorban pedig szerettem volna tisztelegni az eddigi sakkozói múltam előtt. A cél egy olyan számítógépes asztali alkalmazás elkészítése, melyet bárki kedvére használhat.

# Felhasználói dokumentáció

Ebben a fejezetben szeretném elmagyarázni, hogyan is tudja valaki használni a programomat. Részletes segítséget nyújtok annak telepítéséhez, használatához. A könnyebb megértést segítendő képeket raktam különböző helyekre.

## Célközönség

Bárki, aki szeretne sakkozni. Természetesen tisztában kell lenni azzal, hogy ez a játék valószínűleg nem kerül nagyobb felhasználói tömeg elé, így ha szűkebb csoportot akarok megnevezni, akkor szerintem a játék gondolkodására kifejlesztett algoritmusom felkeltheti néhány inkább matematikával, modellezéssel foglalkozó szakember figyelmét.

### Igényelt kapacitás

Gyakorlatilag egy minimális teljesítményű géppel is használható, de érdemesebb egy minimum 8 GB RAM-ot tartalmazó gépen használni az élvezhető sebesség garantálása érdekében. A foglalt tárhely gyakorlatilag elhanyagolható, minimális. (25 MB)

## Üzembe helyezés

A projekt GitHub linkjének[[1]](#_Irodalomjegyzék) a birtokában bárki élvezheti a játék nyújtotta élvezeteket. A fenti linken elérhető a github repository. (a projekt felhőben tárolt változata) Ez a Code<> vagy Kód gombra (alábbi képen piros színnel kiemelve) kattintás után letölthető különböző formátumokban. Elsősorban a megjelenő link segítségével (alábbi képen kék színnel kiemelve), másodsorban pedig .zip (alábbi képen sárga színnel kijelölve) összecsomagolt formátumban.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

. ábra Magyarázat az üzembehelyezéshez

Az alábbiakban az első variációként említett opciót (~/Chess.git link kimásolása, majd annak segítségével új futtatható állomány készítése) fogom részletesebben, a navigációt segítő képernyőképekkel bemutatni.

### Java letöltése

Az alkalmazás java nyelven készült, emiatt először fel kell készíteni a számítógépet, ennek a programozási nyelvnek a fordítására, futtatására. Ehhez látogassuk meg a következő linket:

<https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>

Itt elérhetőek különböző fájlok. Ezekből kiválasztva a megfelelőt, le tudjuk tölteni a számunkra szükséges Java segédfájlokat. Jelen bemutatóban a legelterjedtebb (64 bites Windows) operációs rendszerhez passzoló változat letöltését fogom demonstrálni.

Miután a letöltés befejeződött, a felugró ablakokban sorra válasszuk a megfelelő helyeslő gombokat. Rendre: „Igen”; „Next”; „Next” (Itt meg lehet változtatni a telepítés helyét); „Close”;

### Futtatási környezet (IDE) letöltése

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírásTöltsünk le egy tetszőleges Java IDE-t, aminek segítségével tudjuk futtatni a programot. Itt az IntelliJ IDEA-t fogjuk használni példának. Töltsük le az alábbi linkről a futtatási környezetet, ismét kiválasztva a letöltés után megjelenő ablakokban a helyeslő gombokat. Nyissuk meg a fejlesztőkörnyezetet. Egy ilyet vagy ehhez hasonló ablakot kell látnunk:

2. ábra Amit a futtatási környezet telepítése után látni kell

<https://www.jetbrains.com/idea/download/?section=windows>

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

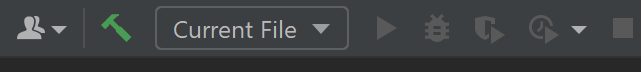
Automatikusan generált leírásVálasszuk ki az alább pirossal jelölt „File” opciót. Ezen belül a „New”-t majd „Project with version control”-t. Ide kell beilleszteni majd a GitHub-ról kimásolt linket. Ezt tudjuk majd klónozni a saját számítógépünkre. Ehhez először szükséges lesz egy Download and Install Git lehetőség választása, amennyiben még a Git verziókezelő nincs telepítve a számítógépre. Ez a lehetőség majd a GitHub link alatt jelenik meg. Ha ez is megvan, akkor sikeresen telepítettük a programot a számítógépre, már csak el kell indítani az alábbiak szerint.

3. ábra Új project készítése Version Control segítségével

### Futtatás

Miután sikeresen letöltöttük a megfelelő Java verziót, és a futtatási környezetet, már csak az van hátra, hogy a futtatási környezeten belül néhány szükséges beállítást elvégezzünk.

Válassza ki fent a „Current File” opción belül az ”Edit configuration”-t. Ezt megnyitva válassza ki az „Add new” majd az Application lehetőséget. Ott meg tudja adni, honnan fogja futtatni a „Main” osztályt. Ezt a „Main” szó gépelésével ki lehet választani. A harmadik képen láthatót válasszuk.



. ábra Futtatási konfiguráció megváltoztatása



. ábra Futtatás applikációként

A képen szöveg, Betűtípus, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

. ábra Megfelelő main osztály kiválasztása

Ezután a következő képen pirossal keretezett Start gombra kattintva tudjuk bármikor kényelmesen használni az alkalmazást. Amennyiben az IntelliJ még felajánl bármilyen lehetőséget a bal alsó sarokban, azokat fogadjuk el, ugyanis azok szükséges file-kiterjesztések letöltését kérvényezik.

A képen szöveg, képernyőkép, Betűtípus, embléma látható

Automatikusan generált leírás

. ábra Alkalmazás indítása

## Felhasználói útmutató, felhasználói utak

Egy ábra segítségével bemutatom milyen lehetőségekkel rendelkezik a játék.

A képen képernyőkép, szöveg, tér, Téglalap látható

Automatikusan generált leírás

. ábra Az ábrán éppen egy új játék veszi kezdetét. Különböző lehetőségek, funkciók kiemelve, megszámozva láthatók

1. „Új játék”: Ezt a gombot kiválasztva tudunk bármikor a program futása alatt új játékot kezdeni. Rákattintva megjelenik a képen középen látható menü, melyben kiválaszthatjuk, hogy a három opció közül milyen játékot szeretnénk játszani vagy nézni.
2. „Világossal szeretnék lenni”: Erre a gombra kattintva Ön van világossal, és amennyiben Ön következik soron, akkor az adott figurára kattintva ki tudja választani, hogy melyik helyre szeretne lépni vele annak lehetőségei közül.
3. „Sötéttel szeretnék lenni”: Erre a gombra ugyanaz történik, csak jelenleg Ön van sötéttel.
4. „Ai vs Ai”: Itt dőljön hátra és nézze ahogy a gép játszik saját magával.
5. A képen szöveg, képernyőkép, Betűtípus, Téglalap látható

   Automatikusan generált leírás„Szünet”: Ez a gomb a játék megkezdésekor nem használható, csak játék közben. Akkor érdemes használni, ha nem Ön van soron, és meg akarja állítani a gép gondolkodását. Rákattintva megjelenik egy felugró ablak, melyen a „Folytatás” - ra kattintva folytatódik a játék, a „Kilépés” – t választva pedig bezárja a programot. A 4. pontban szereplő játékmód esetén is használható.

9. ábra Szünet gomb megnyomása utáni felugró ablak

1. A képen szöveg, képernyőkép, Betűtípus, szám látható

   Automatikusan generált leírás„Mentés”: Ezzel az adott állást tudja elmenteni. Ugyanúgy csak játék közben érhető el, hiszen máskülönben nincs mit elmenteni. Használata után a játék mentése megtörténik egy – a projekten belüli – almappába.

10. ábra Mentés gomb megnyomása után felugró felirat

1. „Betöltés”: Itt bármikor be tud tölteni egyet az elmentett játékok közül a megfelelő fájl kiválasztásával. Az állás megőrzi, hogy ki játszott kivel, tehát a játék onnan fog folytatódni, ahol abbamaradt. Fontos, hogy csak azok a fájlok alkalmasak betöltésre, melyeknek a neve év\_hónap\_nap\_óra-perc-mp formátumú. A többiről (log fájl-ok) később lesz szó.

A képen szöveg, képernyőkép, képernyő, szám látható

Automatikusan generált leírás

. ábra Betöltés gomb megnyomására felugró fájlválasztó segédablak

A fájl kiválasztása után megjelenik egy megerősítésre váró ablak, a kiválasztott fájl nevével. Ha nem választunk, akkor is megjelenik, csak azzal a felirattal, hogy nem a fájl választás meg lett szakítva. Ezek lentebb láthatóak.

A képen szöveg, képernyőkép, Betűtípus, Téglalap látható

Automatikusan generált leírás A képen szöveg, képernyőkép, Betűtípus, Téglalap látható

Automatikusan generált leírás

. ábra Sikeres fájl választás

. ábra megszakított fájl választás.

1. A képen szöveg, képernyőkép, Betűtípus, Téglalap látható

   Automatikusan generált leírás„Feladás”: Ennek segítségével tudja feladni az adott partit, amennyiben Ön következik. A továbbiakban nem tud lépni már az adott állásban.

. ábra Feladás esetén megjelenő jelzés

1. „Döntetlen”: Ezzel a gombbal tud döntetlent ajánlani, melyről a gép dönt, hogy elfogadja-e vagy sem.
2. „Log – Játszma jegyzés”: Ezen az oldalsó fehér felületen történik az adott lépés lejegyzése. **Lépés sorszám. Figura Honnan - Hová** formátumban. A sáncokat megfelelően 0-0 vagy 0-0-0 módon jegyezzük.

A képen szöveg, képernyőkép, Téglalap, Betűtípus látható

Automatikusan generált leírás

. ábra Példa log

1. A képen képernyőkép, szöveg, Játékok, társasjáték látható

   Automatikusan generált leírás„Levett figurák”: Ez egy egyszerű – a játékos által használhatatlan – panel sor, melyekben a játék során levételre került figurákat tároljuk.

16. ábra Jó játékot! (Példa állás látható)

# Fejlesztői dokumentáció

A játék letöltését ezen a projekt GitHub linkjén[[1]](#_Irodalomjegyzék) tudják elérni. Kell hozzá egy előre telepített Java, valamint egy Java programok futtatását támogató IDE. Ajánlom az IntelliJ-t, aminek a segítségével maga a program is készült. Ezek hiányában nem használható a program. Telepítésük hiányában nem lehet futtatni. Telepítésükhöz segítség a [2.2.1](#_Java_letöltése)-es és a [2.2.2](#_Futtatási_környezet_(IDE))-es pontokban található.

A játék maga egy asztali alkalmazás, ami egy sakkjátékot valósít meg. Van lehetőségünk világossal és sötéttel is kihívnunk a készített programot. Ezen felül székünkben hátradőlve nézhetjük, ahogy a számítógép saját maga ellen játszik. Van lehetőségünk a játékot feladni, döntetlent ajánlani. Továbbá szüneteltetni, elmenteni vagy egy elmentett játékot betölteni. Legutóbbi esetben az adott játék onnan folytatódik, és olyan attribútumokkal, ahol és amilyenekkel abbahagytuk.

## Megoldási terv

A feladat egy humán játékos ellen szabályos lépésekkel játszó sakkprogram létrehozása. Gyakorlatilag egy kvázi AI-ról beszélünk. Emiatt a továbbiakban az AI megnevezés az engine-t takarja, amely különböző számítások mentén gondolkodik, és hoz meg egy adott döntést. Ezt az AI-t úgy kell létrehozni, hogy a felhasználónak – legalább – elfogadható élményben legyen része. Magyarán szólva képesnek kell lennie relatíve jó döntések meghozatalára. Ezekhez a döntésekhez tehát szükségünk lesz egy kiértékelésre, ami ezeket a képességeket garantálja.

Mint említettem, a feladat nem csupán egy – az optimális lépést kiválasztani képes – AI létrehozása, hanem egy ehhez kapcsolódó grafikus interfésszel rendelkező alkalmazás is. Ezt, mint már említettem, asztali alkalmazás formájában szeretném kivitelezni. Elkerülve ezzel egy adott webes felülettel járó további bonyodalmakat, mint a szerverrel történő kommunikáció, adatbázis, stb. A jobb felhasználói élményt elősegítendő az alkalmazásnak képesnek kell lennie adott állások mentésére, régiek előhívására. Végül, de nem utolsó sorban pedig alapvető játékélvezhetőségi tényezőként a játéknak elviselhetően gyorsnak kell lennie. Értem ezalatt, mind a géptől érkező válaszlépés – számítási mennyiséghez arányos - relatív gyorsaságát, mind pedig a már említett funkciók hatékonyságban mutatott eredményességét. Ennek a hatékonyságnak a megfelelő abszolválásához szükségünk lesz pár olyan adatszerkezetre, amelyeknek a kezelése nagy sebességgel történik. Ebbe a kezelésbe beletartozik a gyors generálhatóság, valamint a struktúrában eltárolt információk hatékony kinyerése is.

A grafikus felület kialakításában kiemelten figyelni kell a különböző funkciók jól elkülöníthető kialakítására. Ezenkívül a grafikus partícióknak garantálniuk kell a hozzájuk tartozó funkciók mibenlétének azonnali felismerését is. Érdemes tehát olyan kialakítást létrehozni, ami kellően nagy kontrasztot mutat a különböző részek között.

A fejezet további alpontjaiban a fent említett követelményekhez készítek részletesebb terveket.

### Grafikus interfész terv

A következő ábrán látható egy vázlat, amely egy vázlata az elkészíteni kívánt felületnek.

Kezdőablak: A képen diagram, tér, Téglalap, sor látható

Automatikusan generált leírás

. ábra Grafikus interfész vázlat

1. Az „Új játék” gomb tervezett helye. A gomb megnyomására nem áll meg az adott játék, ám miután kiválasztottuk azt a játékmódot, amiben a következő meccset szeretnénk játszani, automatikusan új játék inicializálódik a kívánt tulajdonságokkal. Fontos, hogy jól elkülöníthető, ráutaló felirattal ellátott gombot hozzunk létre, mely minden esetben teljesíti a rászabott funkciót.
2. A „Szünet” gomb tervezett helye. A gomb megnyomására minden éppen futó háttérfolyamat megáll. Ennek leggyakoribb valószínűsített működése az AI számolásának megszakítása. A megfagyasztott folyamatok egészen addig váratnak magukra, amíg nem adjuk ki a manuális megnyilvánulását a döntésünknek, azaz amíg nem indítjuk tovább a játékot. Ezalatt minden más lehetőségnek elérhetetlennek kell lennie.
3. A „Mentés” gomb tervezett helye. A gomb megnyomására semmilyen folyamat nem szakad meg, ám az éppen aktuálisan megjelenített táblát el kell mentenünk valamilyen formátumban. Ennek a megtörténtét, vagy sikertelenségét a felhasználónak tudomására kell hoznunk egy jól észlelhető jelzéssel.
4. A „Betöltés” gomb tervezett helye. A program indításakor ez és az „Új játék” azok, amiknek egyedül vannak használható funkcióik. A többi gomb a program indításakor le van tiltva. Ugyanis az új játékkal egyetemben ez a gomb is azt az opciót valósítja meg, hogy egy új partit tudunk vele kezdeni. No persze itt nagy valószínűséggel nem egy kezdő állás fog létrejönni a játéktáblán, hanem egy annál bonyolultabb – félbehagyott és elmentett – pozíció fog megjelenni középen.
5. A „Feladás” gomb tervezett helye. A gomb által kiadott utasításnak pont olyan egyértelműnek kell lennie, mint a feliratának. Amennyiben a játékos ezt választja a saját köre során, jelezzük felé, hogy a játéknak vége. A táblán maradt figurák a fájó hibára való emlékeztetőül ne tűnjenek el, de a hozzájuk csatolt lépéslehetőségek szűnjenek meg, hogy ne tudjon semmit csinálni a vesztes. Csupán az új játék inicializálására, helyzet mentésére alkalmas gombok funkciói legyenek elérhetőek.
6. A „Döntetlen” gomb tervezett helye. A humán játékosnak a saját köre alkalmával – ennek a gombnak a megnyomásával - legyen lehetősége döntetlent ajánlani. Jelezzük felé látható módon, ha az AI ezt az ajánlatot elfogadta. Amennyiben ez bekövetkezik, járjunk el ugyanúgy, ahogy a feladás esetében is.
7. A játék lejegyzésére szolgáló egyszerű szövegdoboz számára fenntartott hely. Legfőbb tulajdonságai között szerepeljen a read-only. Minden egyes megtett lépés egyazon formátumban legyen lejegyezve. Függetlenül attól, hogy a lépést generáló fél maga a játékos, vagy az AI. Minden játék kezdetekor generálódjon egy szövegfájl amibe ezeket a - logolásként felfogható - lépésjegyzeteket elmentjük.
8. A játéktábla tervezett helye:
   1. Megjelenítésében legyen jól elkülöníthető az ablak többi részétől.
   2. Az egyes mezőkre rákattintva különböztessük meg azokat a mezőket, amikre a kiválasztott mezőn lévő figura léphet. Amennyiben nincs ilyen mező, vagy éppen nincs figura a választott mezőn (véletlen kattintás – vagy csak vaktában kattintgat a felhasználó) ne rajzoljunk ki semmit. Illetve, ha korábban volt olyan mező, ami egy korábban választott figura lépéslehetőségei közé esett valamilyen kiemelt jelzés által, szüntessük meg ezt a jelzést.
   3. Ha történt már olyan figuraválasztás, aminek van lépéslehetősége, és a következő választott mezőnk ezen mezők egyike, detektáljuk lépésként és végezzük el a lépéshez tartozó adatváltoztatásokat. Ezek esetleges plusz hatásait szintén rajzoljuk ki a táblán.
9. Egyszerű mezők, melyeknek a funkciója annyiban kell álljon, hogy az ütésekben elesett figuráknak szolgálnak majd végső nyughelyül. Avagy ha ütés következik be jelenjenek meg rendre ezekben a mezőkben a leütött bábuk.

Ennek a megjelenítésnek pedig abszolválnia kell a következő felhasználói utakat.

A képen diagram, szöveg, kör, tervezés látható

Automatikusan generált leírás

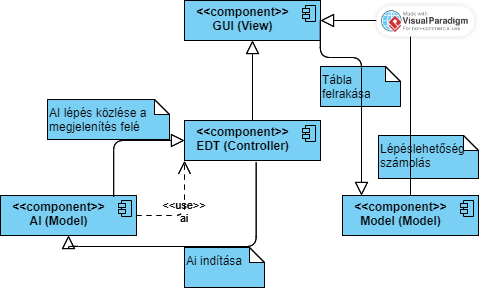
. ábra Felhasználói lehetőségek teljes diagramm

### Komponensek

A program alapvetően négy fő komponensből állni. Ezek MVC (Model-View-Controller) architektúrába rendeződnek 2 – 1 – 1 megoszlásban. A megjelenítésnek egy fő komponense lesz tehát, nevezzük ezentúl GUI-nak (Graphical User Interface) vagy még egyszerűbben UI-nak. Az alkalmazás backend-ét pedig két fő komponensre osztjuk. Az egyik az alapvető szabályokat megvalósító belső modell komponens. A másik pedig az AI működésért felelős – meglepő módon – AI névre keresztelt komponens. A négy réteg közti kommunikáció a következő képpen zajlik:

Minden a megjelenítéssel kezdődik, a felhasználó kiválasztja, hogy melyik játékmódot szeretné használni. Ez alapján megtörténik a kezdő állás felrakása. Ezután két variáció ciklikusan követi egymást.

* 1. Amennyiben a soron következő játékos humán játékos a modell kerül a képbe: ugyanúgy inicializál saját maga számára egy a megjelenítettel egyező állást, és kiszámolja az abban megtehető lépéslehetőségeket. Ezen lépéslehetőségekkel felruházza a UI figuráit, melyek után ezek már saját lehetőségeikből tudnak választani, és kirajzolni a játékos választásait. Miután a játékos választott és a UI-ban megtörtént a lépés, a Controller elindítja útjára az AI-t.
  2. Ha a soron következő fél az AI, akkor lép működésbe a gép. A saját reprezentációját megalkotva elkezdi kiszámolni a lehetőség szerinti legjobb lépést. Ehhez a rekurzív lépésgenerációt használó miniMax algoritmust alkalmazza. Miután így kap egy végeredményt, mely a lehetőség szerinti legjobb lépése, fogja a saját reprezentációját és a Controller segítségével közvetíti azt a UI felé. Ami, miután érzékelte ezt, megteszi a keletkezett – AI szerint legjobb – lépést.



. ábra A komponensek és a köztük történő kommunikáció diagrammja

### A komponensek feladatai

* Controller: Ez a komponens gyakorlatilag két dologra kell, hogy képes legyen. Az egyik az egyéb komponensek közti kommunikáció, a másik pedig az itt használt eszköz megteremtése, konvertálása.
* GUI: Főbb feladatai, a user felé történő változások megjelenítése és induláskor a játék típusának beállítása. Mellék, mondhatni alapvető feladatai a különböző gombnyomások által kiváltott események megvalósítása, azok esetleges más komponensekre történő hatásának beállítása. Pl.: Új játék.
* Model: Fő feladatai, adott állásban a lépéslehetőségek kiszámolása a játékos számára és ezek közvetítése a megjelenítés felé. Amennyiben nincs ilyen, kiváltani / létrehozni a játék végével járó teendőket.
* AI: A gyakorlati feladata lényegileg annyi, hogy a Controller-rel kommunikálva kapja meg a UI-ból általa a számára döntéshelyzetet jelentő pozíciót. Miután kiszámolta a számára lehető legjobb opciót, a Controller segítségével közli ezt a UI-al.

## Megvalósítás

A megvalósítás leírásában olyan adattárolásokra, megoldásokra fogok koncentrálni, melyek elengedhetetlenek ahhoz, hogy hatékony programot tudjunk létrehozni. Kezdjük az adattárolással. Bármelyik komponensben gondolkodunk, mindegyiknek a játéktábla reprezentációjával kell megbirkóznia elsőként. Ez komponensenként eltérő megoldásokat kíván, viszont ha úgy tetszik, mindennek az alapja egy ún. FEN string, mely az adott állás hatékony tárolását szolgáltatja, egyetlen karaktersorozatként.

### FEN String avagy a fő kommunikációs eszköz

A kívánt adatstruktúrát hál’ Istennek nem kellett sokáig keresni. A táblát ugyanis elég könnyen tudjuk tárolni egy ún. FEN (Forsthy-Edwards Notation[[2](#_Irodalomjegyzék)]) string segítségével. Ez a sakkprogramozásban közkedvelt reprezentáció egyetlen sor string-ben eltárolja az összes szükséges adatot, ami alapján fel tudunk rakni egy adott sakkállást, továbbá ki tudjuk számolni az abból keletkező lépéslehetőségeket. Íme egy, a kezdőállást reprezentáló FEN string:

**„RNBQKBNR/PPPPPPPP/8/8/8/8/pppppppp/rnbqkbnr w KQkq - 1 0”;**

A karaktersorozatból történő adatkinyeréshez először is szóközönként darabokra kell szednünk.

* 1. RNBQKBNR/PPPPPPPP/8/8/8/8/pppppppp/rnbqkbnr : Ez az így kapott első rész, mely egy adott táblát reprezentál, a számolás a1-ről kezdődik R (Rook – Bástyával) figurával. Majd rendre: N (Knight – Huszár), B (Bishop – Futó), Q (Queen – Vezér), K (King – Király), B, N R következik. Ez a kezdőállás világos figurákkal megtöltött alapsora. A következő sor kezdetét a / elválasztójel végzi. Ezután nyolc db. P (Pawn – Gyalog) következik. A következő sorban egy egyedüli 8-as szám árválkodik, jelezve, hogy a következő sorban 8 üres mezőt találunk egymás után. A rákövetkező három sor, szakaszott ennek a mása. Majd ismét nyolc darab gyalog következik, csupán annyi különbséggel, hogy sötét színűek – kis betűvel vannak jelölve a FEN-ben.
  2. w : Ez az egy darab w betű jelöli, hogy az adott állásban világosé a következő lépés megtételének lehetősége.
  3. KQkq : Annak a jelzése, hogy világos és sötét is tud még király (K – k) és vezér (Q - q) oldalra sáncolni. Nem feltétlenül van rá lehetőségük – a kiemelt példában sincs – csupán még nem történt olyan lépés egyik bástyával, és a királlyal sem, ami meggátolta volna a sáncolás lehetőségét valamely oldalra. Minden egyes esetben amikor egy ilyen lehetőség elvész, az adott oldalt reprezentáló karaktert ’-’ karakterre cseréljük. Például -Qk- itt világosnak van még lehetősége hosszú, sötétnek pedig rövid sáncot végrehajtani.
  4. - : Ez a karakter jelöli, hogy van-e az adott pozícióban en passant lépésre lehetőség. Ha nincs akkor ez a karakter, mint a példánkban is. Ellenben, ha van, akkor az első karakter jegyzi, hogy melyik sorban van elérhető en passant lehetőség. A második karakter pedig ugyanennek az oszlopszámát jelzi. Pl.: „23” A második sor, harmadik helyére lépve egy gyaloggal a soron következő játékos en passant lépést képes végrehajtani.
  5. 1 : Végtelenül egyszerűen ez egy lépésszám jelző.
  6. 0 : Ez a szám pedig váltakozóan 0 vagy 1, attól függően, hogy világos vagy sötét következik az adott pozícióban.

Ez az ábrázolás a kulcsa a különböző komponensek közti kommunikációnak. Bármilyen helyzetet bármikor tudunk vele ábrázolni. Emiatt a program egyik fő feladata a különböző komponensek táblareprezentációját egy FEN alapján megalkotni.

A vizualizáció, annak modellje, és főképpen az AI eltérő módon ábrázol pozíciókat. Az ezek egymásra történő átfordítása viszont ezt a leképzést használja.

### A GUI megvalósítása

Ennek a komponensnek a megvalósításakor az elsődleges cél - a külalak mellett – a könnyű és gyors használhatóság volt. A felhasználói felülethez egy már – már réginek tekinthető, de annál egyszerűbben használható könyvtárat alkalmaztam a javax.swing[[3]](#_Irodalomjegyzék)– et.

Ebből a következő lehetőségeket hasznosítottam:

1. Maga a felület egy JFrame, amin a tervnek megfelelően elhelyeztem különböző eseménykezeléseket megvalósító JButton-ket, valamint egy plusz JPanelt, amin

8 x 8 JButton-k által reprezentált játéktábla látható. Ehhez a panelhez tartozik továbbá a tábla címkézését szolgáló JLabel lista. Az egyes címkék feliratai attól függően változnak értelemszerűen, hogy a felhasználó sötét vagy világos színnel van.

1. Az egyéb plusz partíciók a logolást megvalósító egyszerű read-only szövegdoboz, és még plusz 16 gomb, amik a levett figuráknak szolgáló helyért felelősek. A szövegdoboz egy egyszerű JTextArea.
2. A mezők megvalósításának leírásánál annyiban csaltam, hogy a mezők maguk leszármazottjai a JButton-nek. Minden képességével rendelkezdnek, de a játékban betöltött szerepük miatt egyéb tulajdonságokkal is fel vannak ruházva. Egészen konkrétan az eseménykezelőikben kellett némi plusz munkát végrehajtanom. A rajtuk lévő figurák megjelenítése pedig a gombok ikon képével történik. Az ikonok egyik gombról a másikra vándoroltatásának szabályos működésének felügyeletét egy véges determinisztikus automatával valósítottam meg, melynek a pseudo kódja alább olvasható.
3. **if (CLICK\_COUNTER == 0 AND clicked.isGotPiece AND clicked.piece.isWhite = whiteToPlay AND NOT clicked.piece.range.isEmpty())**

**THEN**

**changeColor(clicked)**

**lastClicked = clicked**

**pieceToChange = clicked.piece**

**CLICK\_COUNTER = CLICK\_COUNTER + 1**

**ELSE IF (CLICK\_COUNTER == 1 AND locationCollectionContains(pieceToChange.getPossibleRange(), clicked.loc)) THEN**

**changeColor(lastClicked)**

**moveToClicked(clicked)**

**pieceToChange = NULL**

**lastClicked = NULL**

**CLICK\_COUNTER = CLICK\_COUNTER - 1**

**switchWhoComes()**

**ELSE IF (CLICK\_COUNTER == 1) THEN**

**changeColor(lastClicked)**

**pieceToChange = NULL**

**lastClicked = NULL**

CLICK\_COUNTER = CLICK\_COUNTER - 1

END IF

Az volt a feladat, hogy a megjelenítés a program többi részétől függetlenül tudjon működni. Ehhez első sorban arra volt szükség, hogy amíg az AI számolja a saját lépéseit, a játékos által megtett lépés, és annak hatásai jelenjenek meg. Tehát szükség volt egy olyan lehetőségre, ami a program egyes részeinek elindítását szeparáltan tudja megtenni a többitől. Ekkor jutottak eszembe a szálak. Ugyanis némi használat után világossá vált, hogy amennyiben egy szálon, szekvenciális megoldással szeretném elkezdeni a program fejlesztését, elég gyorsan bele fogok futni egy problémába aminek a vizuális megjelenése roppant zavaró. Nevezetesen abba, hogy miután a kiválasztottam a figurámat és annak az új helyét, az AI elindul, és amíg a gondolkodási folyamata le nem zárult, a megjelenítés nem tudja végrehajtani a – még játékos által – megtett lépés vizualizációját. Egészen addig várat ez a művelet magára, amíg a háttérben az AI le nem futott. Ez roppant bosszantó, hiszen minden lépésnél gyakorlatilag egy, maximum kettő JButton ikon változtatás történik. Szükség volt tehát egy megoldásra, ami nem csupán szeparálja a két komponens működését, hanem megvárja, amíg a megjelenítés folyamatai befejeződnek és csak azután kezdi el a gép háttérmunkáját.

Erre szolgált a Swing.invokeLater metódus. Pontosan azt a feladatot látja el, amire nekünk szükségünk van. Ha egy külön szálon futtatjuk ugyanis az AI által végzett műveleteket, akkor ennek a metódusnak a segítségével az említett függvényhívásokat be tudjuk ütemezni, késleltetni amíg a vizuális folyamatok le nem zárultak.

### A Model és a Megjelenítés kapcsolata

Bármennyire is eltérő sok szempontból a model által végzet háttérmunka, és a megjelenítésben történő logika, a főbb feladataikban nincs eltérés.

* Mezők, figurák, és a tábla ábrázolása
* Adott állás felrakása egy FEN string alapján
* Lépéslehetőségek kiszámítása adott figurához.

Emiatt a model, és a GUI táblareprezentációja közös őssel rendelkezik, és a lényegi különbség a lehetséges lépések kiszámításának módjában van. Ugyanis míg a model tényleges számításokat végez ehhez kapcsolódóan, addig a GUI csupán megkapja a model által kiszámolt lehetőségeket.

### Az AI alapkoncepciója

A sakkprogramozás régre visszatekintő múltjában valószínűleg mindenkinek felvetődött két kérdés a következő sorrendben:

* Lehet-e olyan kiértékelő függvényt írni, amely 100% bizonyossággal képes ítéletet hozni arról, hogy mi az adott állásban a soron következő játékosnak a lehető legjobb lépése. Mindezt anélkül, hogy számolnánk az arra adandó válaszlépések következményeivel. Azaz lehetséges-e csupán a figurák értékéből, abszolút- és egymáshoz viszonyított relatív- pozíciójukból egy folyamatosan releváns, állásonként változó konstans értéket létrehozni.
  + Erre a kérdésre a természetesen rengeteg válasz érkezett már. Ám bármily nagy számítást tudtak maguk mögött ezek a kiértékelő függvények, mindegyik megbicsaklott, ha egy elég erős játékossal ültették le játszani.
  + Tehát a válasz eddig egy kategorikus nem.
* Lehetséges-e hatékonyan értékelni a feltételezhetően rekurzívan keletkező állásokat, ha adott állásból úgy keletkezik egy másik, hogy az eredeti egyik lépéslehetőségét a sok közül megtesszük.
  + Erre már érkeztek egészen kielégítő válaszok. A leggyakrabban használt változat az ún. Minimax algoritmus. Melynek az alapötletét nem más, mint Neumann János játékelmélete képezi.

#### Minimax

function minimaxAlphaBeta(node, depth, alpha, beta, maximizingPlayer):

**if depth is 0 or node is a terminal node:**

**return evaluate(node)**

**node.generateChildren**

**if maximizingPlayer:**

**maxEval = -INFINITY**

**for each child in node.children:**

**eval = minimaxAlphaBeta(child, depth - 1, alpha, beta, False)**

**maxEval = max(maxEval, eval)**

**alpha = max(alpha, eval)**

**if beta <= alpha:**

**break // Alpha cut-off**

**return maxEval**

**else:**

**minEval = +INFINITY**

**for each child in node.children:**

**eval = minimaxAlphaBeta(child, depth - 1, alpha, beta, True)**

**minEval = min(minEval, eval)**

**beta = min(beta, eval)**

**if beta <= alpha:**

**break // Beta cut-off**

**return minEval**

**// Initial call**

**minimaxAlphaBeta(root, depth, -INFINITY, +INFINITY, true)**

Az algoritmus pseudo kódjában látható, hogy adott állásban először megtörténik az adott állás (node) lehetséges lépéseinek létrehozása (generateChildren), majd attól függően, hogy világos (maximizingPlayer) vagy sötét (!maximizingPlayer) következik az adott állásban az adott csúcsnak a gyerekei közül kiválogatott legnagyobb illetve legkisebb értékű végkiértékelést adjuk értékül. Ezt lefolytatjuk rekurzívan amíg el nem érjük a kívánt mélységet, vagy az adott állásban véget nem ér a játék. Így keletkezik egy fa, melyről annyit biztosan tudunk, hogy exponenciálisan növekszik a mérete. Ezzel pedig arányosan nő az algoritmusunk futásideje, rossz esetben a felhasznált dinamikus memória is túlcsordul. Ez ellen az egyik védekezés az ún. alfa béta vágás, amely a már eltárolt csúcsértékek alapján meghatározza, hogy érdemes-e adott ágban kiértékelni további gyerekeket. Ha nem, akkor egyszerűen levágja azt a csúcsot, és nem megy arra tovább a lépés generálás. A kérdés tehát az, hogyan tudunk hatékonyan lépéseket generálni?

#### Generate Moves

A probléma az egy állásból létrejöhető lépések összegyűjtésével a következő; miután megtettünk egy lépést, a következő létrehozásához az eredeti pozíciónkból kell kiindulni, tehát a legutóbb megtett lépést vissza kell forgatni. Ezt viszont csak úgy tudjuk csinálni, ha az eredeti pozíciót valamilyen formában eltároljuk és a visszalépés meghívásakor az egész táblát visszaállítjuk az eltárolt verzióra. Ekkor jött a következő ötlet. Tároljuk el csupán az adott pozíció FEN-jét és amikor meghívjuk a visszalépés műveletet akkor az egyedüli dolgunk az állás visszarakása eredeti állapotára. Így pedig a modellben egyetlen tábla objektumra lesz szükségünk. Emiatt ezeket az osztályokat érdemes singleton[4] módon definiálni. Hiszen folyamatosan csak le és fel rakunk rá állásokat.

Igen ám, csak a folyamatos le-fel rakás és String parsolás elég sok időt vesz igénybe. Emiatt kellett egy másik reprezentációt is keresni, amit kizárólag az AI fog hasznosítani. Miért? Mert így a tesztelés során hatékonyan tudom adott állásokban ellenőrizni a lépésgenerációt. Ezt bővebben a [3.3.](#_Tesztelés) pontban fejtem ki.

#### A bitBoard-ok, avagy a sakk-programozás legelegánsabb része.

Köztudott tény, hogy a számítógép 2-es számrendszerben gondolkodik. A CPU-ban a műveletek, 0 és 1-es értéket felvenni képes bitek alapján történnek. A számokat és különböző típusokat is bitek ábrázolnak. Gyakorlatilag mindent. Miért fontos ez nekünk? Mert egészen véletlenül van Java-ban egy típus, a long, ami egészen véletlenül 64 biten van ábrázolva. Ennek tudatában a következőképpen tudunk ábrázolni egy adott állást. A példában az alapállást mutatom be.

Világos gyalogok Sötét gyalogok

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Világos huszárok Sötét huszárok

0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0

Világos futók Sötét futók

0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0

Stb. stb. Ezeknek a segítségével például a világos gyalogok lépéslehetőségeinek meghatározásának alapját képező logika egyszerűen a tábla shitelése balra (<<) attól függően, hogy melyik lépéslehetőségekre vagyunk kíváncsiak.

Pl.:

Világos gyalogok Világos gyalogok

eredeti két lépés előre ( << 16)

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Ilyen, és ehhez hasonló bitműveletekkel ügyesen machinálva a lépésgeneráció idejét (nagyon nagy) nagyságrendileg csökkentettük, hatékony eszközt adva ezzel a Minimax algoritmusunk kezébe.

#### Adott levél csúcs statikus kiértékelése

Ennek a résznek a megoldásában a következő dolgokat használtam fel.

1. Figurák alapértékeinek pozitív (világos) illetve negatív (sötét) előjellel vett összegét. Azaz:
   * 1. Gyalog v: 1, s: -1
     2. Huszár, futó v: 3, s: -3
     3. Bástya v: 5, s: -5
     4. Vezér v: 9, s: -9
     5. Király v: 100, s: -100
2. Különböző figurák különböző mezőkön más-más eséllyel vesznek részt a játékban. Ehhez készítettem pár vizuális példát.

A képen tér, kék, Téglalap, Színesség látható

Automatikusan generált leírásA képen tér, Téglalap, kék, Színesség látható

Automatikusan generált leírásA képen tér, kék, Téglalap, Színesség látható

Automatikusan generált leírás

. ábra Huszár mező értékek

. ábra Világos Bástya mező értékek

. ábra Világos Futó mező értékek

1. Attól függően tehát, hogy milyen mezőn foglalnak helyet, különöző értékek adódnak a figurák alapértékeihez. Értelemszerűen pozitív illetve negatív előjellel. A vizualizációban látható egyre sötétebb színű mezők érnek adott figura számára a egyre kevesebbet. A legsötétebbek sok esetben a figura alapértékével ellentétes előjelű pontot érnek.
2. Végül pedig a játékos következő lépésében lehetséges lépésszámának 10%-át hozzátoldottam még a kiszámolt értékhez, a megfelelő előjellel.

### Osztály architektúra

Ebben az alfejezetben az eddig felvázolt szempontok alapján létrejött osztály architektúrát fogom bemutatni részletesen.

#### A Contorller osztály architektúrája

Mint mindig, most is először ezt a komponenst veszem górcső alá először. Két osztály alkotja, egyikük az ütemezésért, (EDT.java – alias Event Dispatch Thread) a másik pedig a komponensek közti kommunikációért felelős Fen stringel kapcsolatos műveleteket valósítja meg (FenConverter.java).

Az EDT-nek mivel csak ütemezési feladatokat lát el, egyedül a startAi(), a receivedMoveFromAi() és az initialization() metódusai lényegesek. Ezek a metódusok azok amelyeket más komponensek felhasználnak arra, hogy a többiek felé közvetítsék a feladatuk befejezését, vagy éppen egy feladat elkezdését. Az initialization pedig értelemszerűen egy új játék kezdetét jelenti.

A FenConverter-nek ugyanúgy kettő fő feladata van. A különböző komponensek által használt táblareprezentációkat alakítsa FEN-é vagy fordítva. Ennek a megvalósítását végzik a FenToBoard() és BoardToFen() műveletek, melyek egymás inverzei.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

. ábra A Controller komponens Osztálydiagrammja

#### A Megjelenítés és a Model osztály architektúrája

A már említett programozási döntés miatt (közös alapokon nyugvó táblareprezentáció) ezt a gyakorlatilag másfél komponenst együtt tartom érdemesnek tárgyalni. Ugyanis a megjelenítésben és a modelben alkalmazott lényeges, a játék folyamatában aktívan résztvevő függvények nagy többségében ugyanazok. Ilyen a board- és piece- setUp metódusok, valamint a cleanBoard(), rangeUpdater() és egyéb getter függvények. Ezeket az IBoard interfész inicializálja. Ugyanilyenek a táblareprezentációhoz szervesen kötődő másik két interfész; IPiece és IField. Mindkettőnek megvan az őt megvalósító osztálya a view-ban és a modelben egyaránt. Ilyenek az updateRange az IPiece-ben, továbbá a clean() az IField-ben. Ugyanúgy tartozik mindkettőhöz getter és setter metódusok sora. ű A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

. ábra A megjelenítés és a model rendezett, egyszerű osztálydiagrammja

A Model tartalmaz továbbá két osztályt, melyek mindkét helyen ugyanúgy felhasználhatóak. Ezek a GameOverOrPositionEnd és a Move osztály. Az utóbbi egy táblán (függetlenül annak típusától) végzi a figurák mozgatását (főképpen Step() metódus), míg a másik a benne definiált függvényekkel felelős a játék végének érzékeléséért, megfelelő események kiváltásért (főképpen GameOverDecision() metódus).

A képen szöveg, képernyőkép, diagram látható

Automatikusan generált leírás

. ábra Ez pedig az eggyel bonyolultabb, különböző függőségeket jelző osztálydiagramm

#### AI osztályarchitektúra

Főképpen három osztályról kell említést tenni itt. Az első mindenféleképpen maga az AI.java. Az Ai motorja, ami egy szál leíró objektumként lett definiálva. Ennek okáról a [3.2.2.](#_A_GUI_megvalósítása) - es pontban beszéltem bővebben. A fő munkát végző függvénye a már tárgyalt Minimax. Ez a run metóduson keresztül érhető el. Tehát minden egyes AI futás az EDT-ben szereplő egyedüli AI instance újra inicializálásával, futtatásával történik. Így megakadályozva a szálak összegabalyodásának esélyét. A második ami mellett nem mehetünk el említés nélkül a már emlegetett BitBoard-okat létrehozó, kezelő osztályok. Ezek egyike főképpen a BitBoards.java, melyben az általános bitművelet definíciók - , bitboardok kreálása - , azok alakítása FEN stringé és vissza -, szerepel(nek). A másik lényeges pedig a BitBoardMoves.java osztály, melyben statikus metódusok garmadája igyekszik adott állásból minél gyorsabban a lehetséges lépések meghatározását elvégezni. Az itt található fő metódus a generateMoves(). Az itt egész számként reprezentált lépéseket egy map-ben tárolja el, aminek a kulcsai az adott lépés által létrejött állás konstansan meghatározott értéke. Az utolsó itt szereplő osztály pedig az A képen szöveg, képernyőkép, Multimédiás szoftver, szoftver látható

Automatikusan generált leírásEvaluator.java, mely egyes állásoknak határozza meg az értékeit.

. ábra Az AI komponens osztály architektúrája

## Tesztelés

A tesztelésben az egyszerűségre törekedtem. Módszertanilag alkalmaztam egyszerűbb egységteszteket a különböző függvények eredményeire, objektumok helyes létrejöttére, végeredmények helyes kezelésére. Továbbá kell egy nagy, átfogó közös tesztelés a két háttérkomponensre. Ennek lényege, hogy bár két féle megvalósítás létezik a lépéslehetőségek meghatározására, ezek eredményei egy adott pozícióban szükségszerűen egyezni fognak. Tehát adott FEN string-ekből meghatározom egyik, majd másik módszerrel a lehetséges lépések számát, majd azokat vetem össze. Ez a kódban a következő módon szerepel:

public class CombinedMoveGenerationTest { …. }

A tesztek továbbá csoportosítva vannak komponensek szerint is. Ezen felül a tesztekben megtalálhatónak kell lenni a következő szisztematikus rendezésnek:

* 1. UI tesztek esetében a kívánt UI rész helyes létrejötte, értékeinek helyessége.
  2. Back-end tesztek esetében:
     1. a függvény eredmények helyessége.
     2. a megfelelő helyen kiváltandó kivételek ellenőrzésének

# Összefoglalás és további fejlesztési lehetőségek

A program sajnos hiányt szenvedett olyan értelemben, hogy nem voltam képes elég erős döntési algoritmust létrehozni. Ez egy jobban sakkozó felhasználó szemében tekintélyes csorba. Mindazonáltal alapvetően a kitűzött cél szerintem teljesült, és ez a lényeg.

## Továbbfejlesztési lehetőségek

A programon rengeteg továbbfejlesztési lehetőség van. Ilyen például egy olyan kiértékelő program, mely egy adott állás mibenlétéről, pusztán az állás felhasználásával elég jó végeredményt tud nyújtani. Ezzel helyettesítve az alapvetően egyszerű figura és mezőértékekre épülő jelenlegi kiértékelőt.

Érdemes lenne továbbá egy olyan játékállások elemzésére képes teszt környezet létrehozása, amelyben a játékos szabadon tud adott figurákat levenni és visszatenni, lépéseket tenni oda és vissza. Saját kezűleg megválasztani, hogy AI általi válasz lépést szeretne adott helyzetben, vagy még ő maga gondolkodik inkább. Egy, az ezekkel, és ehhez hasonló teszt környezet a játékon belül nagyban megkönnyítené sok sakkozó életét, hiszen gyakorlatilag mint munkaeszközt is képesek lennének használni.

# Irodalomjegyzék

1. A projekt GitHub linkje: <https://github.com/gabrisattila/Chess.git> 2023.12.29.
2. FEN String <https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation> 2023.12.28.
3. Javax.swing <https://en.wikipedia.org/wiki/Swing_(Java)> 2023.12.28.
4. Singleton <https://hu.wikipedia.org/wiki/Egyke_programtervez%C3%A9si_minta> 2023.12.28.