

**Szegedi Tudományegyetem**  
**Informatikai Intézet**

**Klasszikus és tandem sakkprogram**

Szakdolgozat

*Készítette:*  
**Hajdú Bence**  
informatika szakos  
hallgató

*Témavezető:*  
**Dr. Siket István**  
egyetemi adjunktus

Szeged  
2021

# Tartalomjegyzék

|  |           |
|--|-----------|
| Feladatkiírás . . . . .                              | 4         |
| Tartalmi összefoglaló . . . . .                      | 5         |
| Bevezetés . . . . .                                  | 6         |
| <b>1. Motiváció</b>                                  | <b>7</b>  |
| <b>2. A sakkról</b>                                  | <b>8</b>  |
| 2.1. A sakk történelme . . . . .                     | 8         |
| 2.2. Sakk a mai világban . . . . .                   | 9         |
| 2.3. A játékról . . . . .                            | 9         |
| 2.4. Tandem sakk . . . . .                           | 11        |
| <b>3. Felhasznált technológiák</b>                   | <b>13</b> |
| 3.1. A Java programozási nyelv . . . . .             | 13        |
| 3.1.1. JavaFX . . . . .                              | 13        |
| 3.2. Maven keretrendszer . . . . .                   | 13        |
| 3.3. JDBC és SQLite . . . . .                        | 14        |
| 3.4. JSON-GSON . . . . .                             | 14        |
| 3.5. Verziókövetés . . . . .                         | 14        |
| <b>4. Az architektúra</b>                            | <b>15</b> |
| 4.1. Futtatási környezetek . . . . .                 | 15        |
| 4.2. Kliens és szerver kapcsolata . . . . .          | 16        |
| 4.2.1. REST végpontok . . . . .                      | 16        |
| 4.2.2. Websocket végpont . . . . .                   | 16        |
| 4.3. Felhasznált tervezési minták . . . . .          | 16        |
| 4.3.1. MVC . . . . .                                 | 17        |
| 4.3.2. Singleton . . . . .                           | 17        |
| 4.3.3. DAO . . . . .                                 | 17        |
| <b>5. Az alkalmazás megvalósítása</b>                | <b>19</b> |
| 5.1. Authentikáció . . . . .                         | 20        |
| 5.2. Várószobák . . . . .                            | 22        |
| 5.3. Modern sakk megvalósítása . . . . .             | 24        |
| 5.3.1. Játék előfeltételeinek megteremtése . . . . . | 24        |
| 5.3.2. Lépések . . . . .                             | 24        |
| 5.3.3. A sakkóra . . . . .                           | 27        |
| 5.3.4. A mérkőzések vége és mentése . . . . .        | 28        |

|           |  |           |
|-----------|--|-----------|
| 5.4.      | Tandem sakk megvalósítása . . . . .        | 28        |
| 5.4.1.    | A figurák bankja . . . . .                 | 29        |
| 5.4.2.    | A leütött figurák elhelyezése . . . . .    | 30        |
| 5.5.      | Játékos statisztikák . . . . .             | 32        |
| <b>6.</b> | <b>Tesztelés és minőségbiztosítás</b>      | <b>33</b> |
| 6.1.      | Egy számítógépen való tesztelés . . . . .  | 33        |
| 6.2.      | Több számítógépen való tesztelés . . . . . | 34        |
| <b>7.</b> | <b>Összefoglalás</b>                       | <b>35</b> |
|           | Nyilatkozat . . . . .                      | 36        |
|           | Köszönetnyilvánítás . . . . .              | 37        |
|           | Irodalomjegyzék . . . . .                  | 38        |

# Feladatkiírás

A szakdolgozat keretein belül egy többszemélyes asztali sakkprogram készült el Java nyelven. A megvalósítás tartalmaz egy asztali klienst a felhasználóknak, valamint egy szerveret, amelyen keresztül képesek kommunikálni.

A felhasználók regisztráció után létrehozhatnak lobbikat, amelyekbe csatlakozva lehetőségük van sakkpartikat indítani. A partik típusait tekintve lehetnek klasszikus sakkmérkőzések, vagy tandem mérkőzések, kettő a kettő ellen felállásban. Játéktípustól függetlenül a mérkőzések közben sakkóra működik, ami megszorítja a játékosok gondolkodási idejét.

A mérkőzések végeztével, azok elmentésre kerülnek szerveroldalon egy adatbázisban. Az egyes játékosok a profil oldalukon tekinthetik meg a statisztikáikat.

# Tartalmi összefoglaló

- **A téma megnevezése:** Klasszikus és tandem sakkprogram.
- **A megadott feladat megfogalmazása:** Olyan asztali alkalmazás fejlesztése, amellyel a felhasználók regisztráció után képesek várószobát létrehozni, ahhoz csatlakozni és mérkőzéseket indítani egymás ellen, valamint két fős csapatokba szerveződve. A játékosok lehetőség szerint képesek legyenek visszanézni a statisztikáikat.
- **A megoldási mód:** Az alkalmazás két részből tevődik össze. A program szerver része biztosítja a kapcsolatot a felhasználók között, valamint elvégzi az adatbázisba történő mentéseket. A felhasználó számára is látható kliens rész foglalkozik a grafikus megjelenítéssel a felhasználó számára, továbbá rendelkezik a kommunikációhoz és sakk szabályaihoz szükséges üzleti logikával.
- **Alkalmazott eszközök, módszerek:** A program Java nyelven íródott, Maven build folyamat menedzselő keretrendszerrel és JavaFX asztali grafikus megjelenítő használatával. A kliensek és a szerver kommunikációja websocket-en, valamint klasszikus HTTP(Hypertext Transfer Protocol) kéréseken/válaszokon keresztül történik. A játékosok és a mérkőzések adatait SQLite adatbázis tárolja szerveroldalon.
- **Elért eredmények:** Megírásra került egy asztali sakkprogram, amellyel több felhasználó párhuzamosan képes játszani. Az alkalmazás különböző operációs rendszereken is tesztelve lett(Windows, Linux, MacOS), ahol azonos teljesítményt mutatott.
- **Kulcsszavak:** *sakk, tandemsakk, webservet, websocket, Java, JavaFX*

# Bevezetés

A sakk egy kétszemélyes táblajáték, és egyben sportág is. Világszerte rendeznek világbajnokságokat 1886 óta. Különbőféle változatai ismertek, Európában és hazánkban is a legelterjedtebb a modern sakk.

A technológia fejlődésével felmerült az igény a sakkprogramok megalkotására. 1947-ben Turing elkészítette a világ első sakk-computer programját, majd 1970-ben megtörtént az első összecsapás különböző sakk-computerek között. Nemsokkal később 1974-ben megrendezték az első sakk-program világbajnokságot, amit a Szovjetunió által tervezett Kaissa nyert.

A levelezési sakk, vagy más néven távsakk a sakkjáték azon formája amikor a játszó felek egymástól távol vannak, és a lépéseket valamilyen közvetítő eszköz segítségével továbbítják ellenfelüknek. A leggyakoribb közvetítést szolgáló eszköz hosszú időközön át a speciális postai levelezőlap volt, de játszottak távsakkot telefonon keresztül, táviratok, faxok váltásával, vagy rádión történő közvetítés révén is.

Ahogy a számítógépek eljutottak az átlagember háztartásába és megindult a még mai nap is tartó informatikai dömping, úgy a szórakoztató, oktató jellegű alkalmazások fejlesztése is felgyorsult. Számos platformra, operációs rendszerre készültek sakkprogramok, sakkot oktató szoftverek. Ezen programok megreformálták a levelezési sakkot, egy új, könnyed kommunikációs eszközt adva a játékosok kezébe. Ezt a vonalat követve, munkám során egy olyan alkalmazást valósítottam meg, melyben a játékosoknak lehetőségük van a sokak által művelt, mondhatni klasszikus, modern sakkipartikat játszaniuk, valamint egy kisebb réteg számára ismert, de ugyancsak versenyek tárgyát képező sakkvariánsban, a tandemben összemérni tudásukat.

# 1. fejezet

## Motiváció

Kisgyermek korom óta érdekelt a sakk. Édesanyám ágáról mondhatni családi hagyomány a táblajáték szeretete, velem sem volt ez másképp. 8 éves koromban elkezdtem oktatóhoz járni, aki megtanított különböző taktikákra, nyitásokra. Ezt követően versenyekre is jártam pár évig. A mai napig érdeklődéssel ülök le egy parti elé és élvezem a kimenetelétől függetlenül.

Egyetemi tanulmányaim kezdetekor keletkezett az ötlet, egy sakkprogram megírásához. A félévek során a szakdolgozati téma választásakor éreztem azt, hogy ennek eljött az ideje. Érdekemben állt olyan technológiát választani amiben otthon érzem magam, viszont szerettem volna túlmutatni az tanulmányaim során tanultakon. Ezért fogalmazódott meg bennem egy olyan alkalmazás elkészítése, amely a későbbiekben akár eszközt adhat a távoli kapcsolatok fenntartásában egy sakkparti képében.

A tandem egy kisebb réteg számára ismert variánsa a játéknak és személy szerint kedvelője vagyok, viszont ritka olyan partnereket találni akik tisztában vannak a szabályaival. Alkalmazásom elkészítésének másik indoka az edukáció volt. Célomként tűztem ki olyan program létrehozását, amellyel a sakkot kedvelőknek egy másik nézetből is megmutathatom a játék szépségeit.

## 2. fejezet

# A sakkról

### 2.1. A sakk történelme

A sakk gyökerei valahol a Távol Keleten eredeztethetők. A sakktörténet kezdete legalább annyira rejtélyes mint maga a táblajáték összetett világa. Az ismert mesék szerint egy Széta nevű brahmin, vagyis tanult ember találta fel a sakkot. Az ősi játékot Indiában csaturanga néven játszották. Végső nevét a perzsa "shäh" szóval rögzítettek miután elterjedt az arab területeken. A szó maga uralkodót jelent, de egyúttal arra a játékhelyzetre is utal amikor az ellenfél királya "ütésben van".

Európában először Spanyolországba került be a játék, mór hódítók révén a 8. században. Ezt követte Szicília, majd a vikingek vitték tovább nyugat felé, ugyancsak a hódításaik során. A sor végét Britannia zárta ahová a normannok vitték be, a szigetország 1066-os elfoglalása során. A szabályok több forrásból való lejegyzéséből bizonyítható, hogy már sok helyen ismerték a táblajátékot ekkoriban. Az első olyan Európai játékot jegyző könyv amely említi a sakkot A játékok könyve (Libro de los juegos) volt, mely Bölcs Alfonz megbízásából jelent meg 1283-ban. Az első, kizárólag sakkról szóló könyv pedig az ún. göttingeni kézirat volt 1471-ben. 1497-ben megjelent az első nyomtatott sakk-könyv: a spanyol Lucena műve, majd 1561-ben Ruy López de Segura kiadta az első, teljes sakktankönyvet, amely már tartalmazta az összes eddig ismert megnyitást, és szakszerű leírásokat adott a játékról. A népszerűség növekedés megkérdőjelezhetetlen volt, olyannyira, hogy 1550-ben megalakult Olaszországban a világ első sakk-klubja.

Az első nemzetközi versenyt Londonban rendezték meg, 1851-ben. Ennek egyik partija Adolf Anderssen és Lionel Kiseritzky között játszódtott amit még ma is úgy emlegetnek, mint a "halhatatlan sakkjátszma". 1886-ban megrendezésre került az első hivatalos világbajnokság amit Wilhelm Steinitz nyert meg. A negyedik sakkvilágbajnok Alexander Aljechin haláláig, a világbajnok szabhatta meg a világbajnoki mérkőzés feltételeit. A világbajnokságokat, és magát a versenyszerű sakkozást koordináló nemzetközi sakk-szövetség (FIDE) 1924-es alakulását követően 1948-tól kezébe vette a világbajnokságok rendezését, amit több mint 50 évig a szovjetek egyértelműen uraltak a teljesítményükkel. Ezt a hosszan tartó nyerési szériát a legendás Bobby Fischer törte meg, amikor legyőzte Boris Spassky-t a Reykjavík-i világbajnoki meccsen 1972-ben. A párharcukat méltán nevezik az "évszázad mérkőzése"-nek.



## 2.2. Sakk a mai világban

*"Nobody knows, even roughly, how many people play chess, and nobody should pretend to know."* Edward Winter idézetével élve, nem kell ismernünk pontos számot arról, hányan sakkoznak a világon, viszont a chess.com becslései szerint körülbelül 600 millió játékos ül tábla elé a világszerte. Ebből a hatalmas számból kiindulva kijelenthetjük, hogy sakk napjainkban is az egyik legnépszerűbb táblajáték.

Visszamenőleg a partik lebonyolítása egyértelműen úgy történt, hogy a játszó felek találkoztak, felrakták a táblát és lejátszották a mérkőzésüket. A mai világban a technológia fejlődésének köszönhetően ez megváltozott és nem szükséges fizikai kontaktus a partik lejátszásához. Az internet népszerűségével lehetőség nyílt arra, hogy a partnerek egy szoftvert telepítve az asztali számítógépekre, játszani tudjanak egymással akár a világ két távoli pontjáról is, a laptopok, okostelefonok, tabletek megjelenésével pedig még PC-khez való kötődés is megszűnt. Manapság ezekkel a multiplatform megoldásokkal felgyorsult és nagymértékben leegyszerűsödött egy sakkmérkőzés lebonyolítása, de persze sokan a klasszikus tábla előtti sakkot preferálják.

A mesterséges intelligencia megjelenésével pedig nem csak élő emberekkel mérhetjük össze a tudásunkat. A történelem során több sakkra kiélezett program valamint számítógép készült, ezek közül talán a legismertebb az IBM által fejlesztett Deep Blue ami 1997-ben legyőzte az akkori világbajnokot Gari Kaszparovot. Napjainkban már számos algoritmus és sakkra betanított mesterséges intelligencia ellen játszhatunk a készülékeinken. Ezen algoritmusok és szoftverek megírása nagy feladatot jelent a programozók számára ugyanis a lehetséges lépések száma valahol  $10^{123}$ -on köré tehető. Belátható, hogy a sakk, mint kétszemélyes, lépésváltásos, determinisztikus, zéró összegű játék felettébb nehéz problémának minősül.

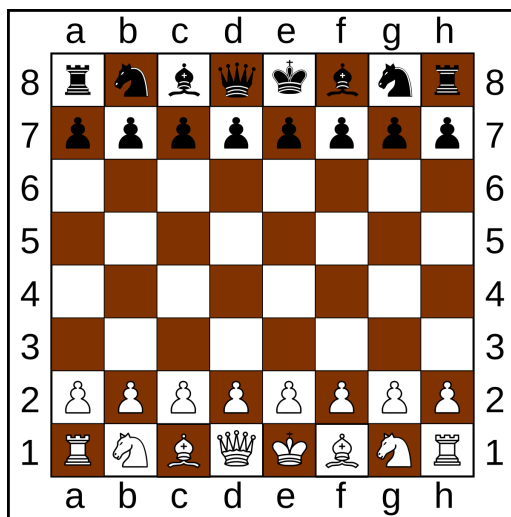
## 2.3. A játékról

Kiindulva a modern sakk népszerűségéből, annak szabályairól és alapjairól csak nagy vonalakban írunk.

A sakk kétszemélyes táblajáték amit egy 8x8-as négyzet alapú világos-sötét pepita mintájú táblán játszanak. A tábla mezői meg vannak címezve általában olyan módon, hogy az oszlopokat az angol ábécé első 8 betűjével, a sorokat pedig arab számokkal egytől nyolcig indexelik, esetlegesen megcserélve a sorokra-oszlopokra való címezést. Ez a címezés a mérkőzések lejegyzéséhez használatos. A tábla akkor áll helyesen ha a bal alsó sarokban sötét, a jobb alsóban pedig világos színű mező van, továbbá a világos játékos bal keze felől eső mező az A1. A mezők színe váltakozó. A tábla közepét, ami a D4, D5, E4, E5 mezőket foglalja magába *centrumnak* nevezik. Az A1 - H8, illetve A8 - H1 átlókat *nagyátlónak* hívják. Az A - D vonalak neve *vezérszárny* az E - H vonalaké pedig a *királysárny*.

A sakkot a játékosok bábukkal (más néven figurákkal) játsszák, melyből mindkét játékos rendelkezik 16 darabbal, amit a feléjük eső alsó két sorban helyeznek el meghatározott módon. Alulról a második sorokban foglalnak helyet a gyalogok, a mögöttük lévő utolsó sorban pedig a többi bábu az alábbi felállásban: bástya, huszár, futó, király/vezér, vezér/király, futó, huszár bástya. A vezér és a király elhelyezése a játékos színétől függ, mindig a vezér áll a saját színén. Minden bábu értékkel bír, melyet lépéslehetőségeikhez mérten

határoznak meg. Egyszerűbb esetben szokás úgy becsülni, hogy minden gyalog 1 egységet ér, a futó és a huszár 3 egységet, a bástya 5-öt, a vezér pedig 9-et. A figurák lépéseinek részletezése előtt fontos megjegyezni, hogy a bábukkal csak úgy léphetünk, ha más bábút nem ugrunk át. Ez alól kivétel a huszár. A saját figuráinkkal leüthetjük az ellenfelét, abban az esetben ha az adott bábunk egy olyan mezőre léphet, ahol egy az ellenfél által birtokolt figura áll.



2.1. ábra. A sakktábla

Forrás[14]

- **Király**: A király a játék legfontosabb figurája, hiszen az egész játék a támadásáról szól. Bármely irányba léphet(horizontálisan, vertikálisan, átlósan), de csak egyet, azaz a közvetlen szomszédos mezőkre.
- **Vezér**: A vezér mozgása a legkevésbé korlátolt a figurák közül, ezért is ő a játék legértékesebb figurája. A királyhoz hasonlóan minden irányba léphet, viszont nála nincs meghatározva a lépett mezők száma.
- **Bástya**: Bármennyi mezőt léphet függőleges vagy vízszintes irányban. Van egy hozzá kapcsolódó speciális lépés a *sáncolás*, ez kifejtésre kerül.
- **Futó**: A futók bármennyit léphetnek átlós irányban. Mindkét játékos rendelkezik kettő futóval melyek közül az egyik a sötét, másik pedig a világos mezőkön képes lépni.
- **Huszár**: A huszár, vagy más néven ló, a róla elnevezett mozgással *lóugrásban* képes a tábla mezőin lépni. Ez a lépés vízszintesen jobbra vagy balra két mezőt, majd függőlegesen fel vagy le egy mezőt jelent. Ennek a fordítottja is megengedett amikor vízszintes irányban valamerre egy mezőt, függőleges irányba pedig két mezőt lép.
- **Gyalog**: A gyalog csak előre léphet. Alapállásban a játékos eldöntheti, hogy egyet vagy kettőt lép vele, de a továbbiakban csak egyre van lehetősége. A többi bábuval ellentétben a gyalog ütni nem a haladási irányában képes, hanem átlósan, egy mezőnyi távolságra.

Van lehetőségünk speciális lépésekre a figuráinkkal. A legismertebb ezek közül a sáncolás, amikor is a király a kiválasztott bástya irányába lép kettőt, a bástya pedig a király által átlépett, tehát a királlyal szomszédos mezőre lép. Ezen lépés feltétele, hogy egyik bábuval se lépett még a játékos, nem áll kettőjük között más figura, és a sáncolás előtt és után sincs sakkhelyzet. Speciális lépés még a menet közbeni ütés (*en passant*) amikor is az érintetlen gyaloggal kettőt lépve áthaladunk egy ellenséges gyalog ütésmezején. Ebben az esetben az ellenfél gyalogja a következő lépésében leütheti a miénket. Az utolsó speciális lépés az *átalakulás* amikor a gyalogunkkal az ellenfél alapsorába lépve lecserélhetjük azt egy tisztre függetlenül azok táblán lévő számától.

A parti során a játékosok felváltva lépnek a bábuikkal. A játék célja az ellenfél királyának támadása, amit úgy tehetünk meg, hogy *sakkot adunk*. A sakkhelyzetkor az ellenfél lépését követően olyan helyzet alakul ki, hogy amennyiben ő következne leüthetné a királyunkat. Ezt többféleképpen lehet kivédeni. A legintuitívabb megoldás ha olyan mezőre lépünk a királyunkkal amit nem támad éppen az ellenfél figurája. Egy másik megoldás lehet ha olyan bábút helyezünk a király és az őt támadó figura közti valamelyik mezőre amellyel blokkoljuk az útját és ezzel megvédjük a királyunkat. A harmadik megoldás ha leütjük a királyt támadó bábút. A játéknak akkor van vége ha kifogytunk azokból a lépésekből amelyekkel a sakkhelyzetet hárítani tudnánk, ilyenkor *mattot* kaptunk. A játék végződhet mattal, feladással, döntetlen ajánlásának elfogadásával vagy patthelyzet kialakulásával, amikor a soron következő játékos nem tud szabályos lépést tenni, de a király a nem áll sakokban, ilyenkor a játék automatikusan döntetlen.

## 2.4. Tandem sakk

A tandemsakk egy kisebb réteg által ismert, de népszerű változata a táblajátéknak. A játék két táblán történik, négy fővel két-két fős csapatokban. Nevezhető a modern sakk kiegészítésének. A szabályok változatlanok, leszámítva azt, hogy az egyik táblán leütött bábukat a játékosok továbbadhatják a csapattársuknak, aki azt elhelyezheti a tábláján. A mérkőzések során sakkórát használnak, igen rövid időkorláttal. Méltán kapta a *bughouse*, azaz bolondokháza nevet a játék variáns, mert a bábuk cseréjétől és kevés gondolkodási időtől a játékmenet véletlenszerűbbé, kaotikussá képes válni.

A csapatok egyik tagja világos, míg a másik sötét figurákkal játszik. Értelemszerűen a csapatokból az ellentétes színű játékosok sakkoznak egymás ellen. A leütött figurákat a sakkozók elhelyezhetik bármely üres mezőre, beleértve azokat is amelyekkel sakk vagy matt helyzetet alakítanak ki. A gyalogok elhelyezése annyiban van megkötve, hogy az első és utolsó sorokba nem rakhatóak le. A mérkőzés közben a leütött bábuk az összes játékos számára láthatóak. A gyalogváltozás során az alapsorra érő gyalog tetszőleges bábura cserélhető, leütésekor viszont eredeti formájában, gyalogként kerül be a *bankba*. Ezen speciális lépés szabályairól több változat is ismert, munkám során elkészült alkalmazásban a tandemsakokban nincs gyalogváltozás.

Az idő kulcsfontosságú tényező a tandemsakk során. Modern sakokban is egy olyan játékelemről beszélhetünk, mely megköti a játékosok gondolkodási idejét, tandemben ez hatványozottan igaz, amire azért van szükség, hogy a játékosoknak ne legyen "korlátlan" ideje várni csapattársuk ütésére. Az órákat egyszerre indítják a mérkőzés elején. Van lehetőség óra nélkül is játszani, ilyenkor valamilyen megkötés van érvényben, például addig várhat a játékos amíg a csapattársa nem tesz meg három lépést.

A mérkőzést játszó felek az egész parti során kommunikálhatnak egymással. Mivel a játékot egymás melletti táblákon játsszák, mindenki hall mindenkit, a bábukérés és játékjavaslatok tehát nem titkoltak. Vannak társaságok akik viszont kommunikáció nélkül tandemeznek, ilyenkor a másik tábla áttekintésével kell tájékozódniuk a lehetőségekről.

A csapatjátékból kifolyólag a partinak akkor van vége, ha az egyik táblán valaki győztesként kerül ki. Ez lehet matt adással vagy az ellenfél idejének lejáráásával, feladással.

Fontos megjegyezni, hogy a tandem egy kevésbé kötött játékstílus, ezért világszerre számos változata ismert, van ahol például a győzelemhez egy csapat mindkét tagjának nyernie kell, megnövelve a döntetlenek gyakoriságát. A fentebb említett szabályok a leggyakoribb változatot tárgyalják. Rendeznek belőle versenyeket is, viszont gyakrabban szórakozás céljából játsszák a sakkozók, hogy el tudjanak rugaszkodni az alap játékmenettől. Jó gyakorlása lehet például a szűk időkerettel rendelkező szituációknak.

## 3. fejezet

# Felhasznált technológiák

Az alkalmazás elkészítéséhez több, a munkaerőpiacon is használt technológiát alkalmaztam. Nagy részük az alkalmazás működésért felelős, de vannak olyanok is amik a fejlesztési folyamat gördülékenységéhez használatosak. Az alábbi fejezetben ezeket szeretném bemutatni nagy vonalakban.

### 3.1. A Java programozási nyelv

A Java széles körökben használt általános célra fejlesztett objektum orientált programozási nyelv. A Sun Microsystems fejlesztette 2009-ig, ezt követően felvásárolta az Oracle. Szintaxisában leginkább a C++-ra hasonlít, viszont objektummodelljei egyszerűbbek annál, ezért is preferáltabb sokak számára. A Java egyik, és talán legnagyobb előnye a platformfüggetlenség. Fordításkor a programkódból egy köztes reprezentáció, bájtkód keletkezik, amit a JVM (Java Virtual Machine) futtat. A végfelhasználóknak a számítógépükön elég telepíteni a JRE-t (Java Runtime Environment) az alkalmazások futtatásához.

#### 3.1.1. JavaFX

A felhasználói élmény biztosításához elengedhetetlen, hogy az alkalmazás rendelkezzen grafikus felülettel. Kezdetben a JavaFX fejlesztésének a célja az akkor használt Swing lecserélése volt, mint a Java platform standard grafikus interfész könyvtára. Jelenleg nem képi részét a beépített Java könyvtáraknak, különálló attól. Segítségével könnyedén fejleszthetünk asztali, webes és mobil környezetre alkalmazásokat. Grafikus eleminek tárháza igen gazdag, ezzel is számos lehetőséget adva a fejlesztők kezébe amivel kielégíthetik a felhasználói igényeket. Egyik nagy előnye, hogy a grafikus felületet definiáló fájl XML alapú, és teljes mértékben elkülöníthető a kódtól. Az ilyen fájlok .fxml kiterjesztéssel rendelkeznek. Ahogyan a Java, a JavaFX is cross-platform, a felhasználásával írt alkalmazások futtatásához JVM szükséges.

### 3.2. Maven keretrendszer

Az Apache Maven (továbbiakban csak Maven) egy build keretrendszer, amely Java nyelvű alkalmazások build-eléséhez ad segítséget. Ez magában foglalja a függőségek keze-

lését, a Java fájlok könyvtárba rendezésének egyszerűsítését, dokumentáció generálást és még sok mást. Rendelkezik egy központi tárolóval (angolul repository), a Maven Central Repository-val, ahonnan beszerezhetjük alkalmazásunk függőségeit és akár mi magunk is publikálhatunk csomagokat. A Maven-t használó projektek rendelkeznek egy leíró fájlal a pom.xml-el, ami tartalmazza a függőségek listáját, valamint a build folyamathoz szükséges beállításokat.

### 3.3. JDBC és SQLite

Alkalmazásfejlesztés szempontjából az esetek nagy részében szükségünk van arra, hogy fejlesztői, vagy felhasználói oldalról, hogy konzisztensen tároljunk bizonyos adatokat, például bejelentkezési információkat, mérkőzések eredményeket és még sok mást. Ehhez szükségünk van egy adatbázisra ami tárolja az adatokat. A Java nyelv részét képezi a JDBC (Java Database Connectivity), ami egységes interfészt biztosít különböző adatbázisok kezeléséhez.

Az SQLite egy relációs adatbázis kezelő rendszer, amit egy C programkönyvtár tartalmaz. Alkalmas kis méretű adatbázisok kezeléséhez, és JDBC-vel minimális konfigurálással, könnyen használható, egyszerűen csak hozzá kell adnunk a projektünk függőségi listájához a driver-ét. Akárcsak a Java, platformfüggetlen.

### 3.4. JSON-GSON

A JavaScript Object Notation(JSON) egy ember számára is olvasható szöveg alapú szabvány, ami a JavaScript programozási nyelvből alakult ki és adatcserét hivatott megvalósítani. Objektumok reprezentációjára alkalmas, amik nem mások, mint asszociatív tömbök vagy egyszerű adatstruktúrák. Míg JavaScriptben JSON-ből egyszerűen készíthetünk objektumokat, Java-ban ez kissé bonyolultabb. Erre ad megoldást az alkalmazásban is használt GSON ami egy, a Google által biztosított könyvtár, melynek segítségével Java objektumokat szerializálhatunk és deszerializálhatunk JSON-be, valamint JSON-ből.

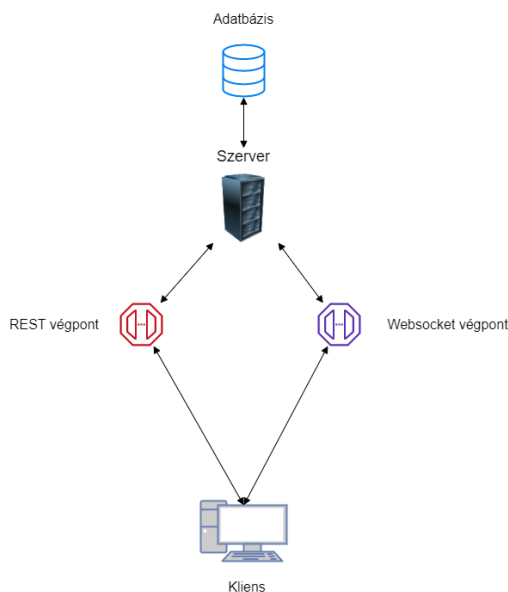
### 3.5. Verziókövetés

A verziókezelés fontossága a szoftverfejlesztés során kritikus pont lehet, nélküle a projektek organizálása, kiadása kaotikussá válna. Ebben segítenek a verziókövető rendszerek (VCS), mint például a Git, vagy az SVN. Ezek szoftveres segédprogramok, melyek képesek nyomon követni a fejlesztő fájlrendszerében történt változásokat. Ezeket lehetőségünk van feltölteni egy központi tárolóba (repository). Minden feltöltés egy külön *commit* üzenet. Lehetőségünk van továbbá a fejlesztés különböző állapotaiból ágakat képezni, akár commit-okból is. A fejlesztők általában külön branch-en dolgoznak, és amikor feladataikkal végeztek, megtörtént a munkájuk felülvizsgálása, akkor az ág amin dolgoztak, egyesíthetővé válik más ágakkal. Amennyiben többen írnának egy fájl, konfliktus keletkezik, amit a verziókövetők le tudnak kezelni, viszont feloldaniuk a fejlesztőknek kell őket. Munkám során Git-el dolgoztam.

## 4. fejezet

# Az architektúra

Az alkalmazás két fő részre bontható. A kliens része az amit a végfelhasználók használnak, és ezek kommunikációjához, valamint az adatbázissal való kapcsolattartáshoz tartozik egy szerver rész. Az egyes részek különálló Java alkalmazások, portabilisak minden operációs rendszerre.



4.1. ábra. Kommunikációs csatornák

## 4.1. Futtatási környezetek

A build folyamat során a projektekből egy-egy JAR fájl keletkezik. A JAR olyan ZIP tömörítés, mely tartalmazza a Java osztályokat és a hozzájuk tartozó metaadatokat. Ezek készítéséhez, magához a build-hez JDK(Java Development Kit) szükséges, a JAR-ok futtatásához viszont elegendő egy JRE. Az alkalmazások 11-es Java verzió használatával kerültek megírásra, ezért a runtime verziójának el kell érnie ezt a szintet, vagy magasabbat.

A JAR fájlok úgy lettek elkészítve, hogy tartalmazzák minden függőségüket, és a tömörítés részét képezik, egyetlen fájlban. Ennél a pontnál fontos megjegyezni, hogy bár

a Java portabilis nyelv, a build-eléskor a függőségek közé csatolni kellett külön a .so és .dll fájlokat. Az előbbi az Unix alapú operációs rendszerekhez (Linux disztribúciók, MacOS), utóbbit pedig Windows-hoz.

Futtatás szempontjából két lehetőségünk van. Indíthatjuk az alkalmazásokat grafikusan, dupla kattintással, mit bármelyik futtatható állományt, vagy parancssorból a *java -jar <filename>* paranccsal. A szerver részt célszerűbb parancssorból futtatni, ilyenkor láthatjuk a különböző log üzeneteket, mint például ha egy felhasználó bejelentkezett.

## 4.2. Kliens és szerver kapcsolata

### 4.2.1. REST végpontok

A REST (Representational state transfer) egy szoftverarchitektúra típus, amely elosztott kapcsolatot képes biztosítani nagy internet alapú rendszerek számára. A kliensek ezeken a végpontokon keresztül végzik a szerverrel a kommunikáció azon részét, ahol a sebesség nem kritikus pont. Ilyenek például az adatbázissal való közvetett interakciók, vagy a partnerlista frissítése. A kommunikáció HTTP kérésekkel történik a definiált végpontok felé. A kérések body részébe GSON segítségével átalakított Java objektumokat küldenek a kliensek, és a szerver válasza is ilyen formában érkezik, növelve az alkalmazás konzisztenciáját.

### 4.2.2. Websocket végpont

A websocket egy kommunikációs protokoll, ami duplex kommunikációs csatornát biztosít TCP kapcsolat felett. A websocket kapcsolat sebességét tekintve gyorsabb a REST felé indított kéréseknél, széles körben alkalmazzák például csevegő alkalmazásokhoz, játékokhoz. A kétirányú kommunikációnak köszönhetően lehetőségünk van üzenetszórásra (broadcast), amely során a kliens által elküldött üzenetet a szerver képes elküldeni a többi kliens számára is, minimális késleltetéssel. Ez a kapcsolat az alkalmazás termináláig fenn áll.

Az alkalmazásban a kliensek sikeres autentikáció után felcsatlakoznak a websocket csatornára, és a sakkpartik során itt történik a kommunikáció. A játékos lépése helyességének validálása után az elküldésre kerül websocketen keresztül, amit a szerver üzenetszórással továbbít a többi kliens felé. Ezen a ponton szükséges, hogy a kliensek információval rendelkezzenek az ellenfelek identitásával, képesek legyen eldönteni, hogy a beérkező üzenet a partnerüktől érkezett-e. Amennyiben igen, frissítésre kerül a grafikus felület.

## 4.3. Felhasznált tervezési minták

A szoftverfejlesztésben tervezési mintáknak (angolul: design pattern) nevezzük azokat a paradigmákat, melyek egy általános, újrafelhasználható megoldást adnak gyakran jelentkező problémákra, helyzetekre. Felfoghatjuk őket sablonként is, egy szituáció megoldására. Segítségükkel javíthatjuk a megírt szoftver minőségét. Az objektum orientált világban tipikusan kapcsolatokat és interakciókat határoznak meg egyes osztályok és objektumok



között, anélkül, hogy specializálnák a konkrét osztályokat vagy objektumokat. Munkám során három tervezési mintát használtam fel, az MVC-t, a Singleton-t, valamint a DAO-t.

### 4.3.1. MVC

Az MVC (Model-View-Controller) gyakran használt tervezési minta amit grafikus interfésszel rendelkező alkalmazások során szoktak használni a fejlesztők. Alkalmazásával lehetőségünk van hatékonyan szeparálni egymástól a program üzleti logikájáért felelős részt három részre.

A View réteg felel a grafikus interfész megjelenítéséért, és az ezzel történő interakciók kezeléséért. A JavaFX használata ezt maximális mértékben támogatja, hiszen a grafikus felületet leíró rész külön fájlban, az fxml-ben található.

A központi komponense a mintának a Model réteg. A felhasználói interfésztől független, meghatározza az adatstruktúrát, annak kezelését.

A Controller réteg felelős az előbb említett rétegek közötti kommunikációért. Validálja az inputokat, és képes a Model réteg aktualizálására, a grafikus felület frissítésére.

Az MVC mintát alkalmazva, alkalmazásomban is kialakult a három réteg. A View réteg felelős az fxml állományok betöltéséért, és azoknak kezeléséért. Az fxml-ek tartalmaznak előre definiált formázásokat, stílusokat, viszont vannak olyan helyek ahol kód segítségével történik a grafikus interfész kezelése. Ilyen például a sakktábla mezőinek színezése, vagy a bábuk elhelyezése a játékos színétől függően. A Model réteg definiálja az adatbázisban elmentésre kerülő adatokat, mint a játékosok adatai, mérkőzések, vagy a websocket kommunikáció során használt socket üzenet osztálya. A Controller réteg kliens oldalon kezeli a grafikus interfésszel való interakciókat, valamint felel a szerverrel való kommunikációért. Szerver oldalon ez a réteg szólítja meg a Model réteget, és dolgozza fel az onnan érkező adatokat, amit utána visszaküld a kliensnek.

### 4.3.2. Singleton

A Singleton ("egyke") tervezési minta az objektumok példányosítását köti meg, hogy a memóriában egy bizonyos osztály példányából csak egy létezhesen. Erre akkor van szükség ha az alkalmazásunkban egy olyan objektumot akarunk létrehozni amely központi és a többi objektumnak ezt az egy példányt célszerű használnia.

Munkám során fontosnak tartottam használni ezt a mintát, olyan rendszer szintű műveletek használatához, mint a központi információk tárolása, alkalmazás property-k kezelése, további menedzser objektumokkal való kommunikáció.

### 4.3.3. DAO

A DAO(Data Access Object) tervezési mintát széles körben használják az egyes programozási nyelveknél, viszont leginkább a Java nyelvvel és a relációs adatbázisokkal asszociált. Az MVC mintán belül a Model réteghez sorolható. A minta lényege, hogy absztrakt interfészt biztosítson az adatbázisbeli objektumokkal való interakciókhoz. Lehetőségünk van az adatokat szolgáltató réteget olyan módon szeparálni, hogy az ne befolyásolja az alkalmazás üzleti logikáját, továbbá az adatbázis is könnyebben cserélhetővé válik.

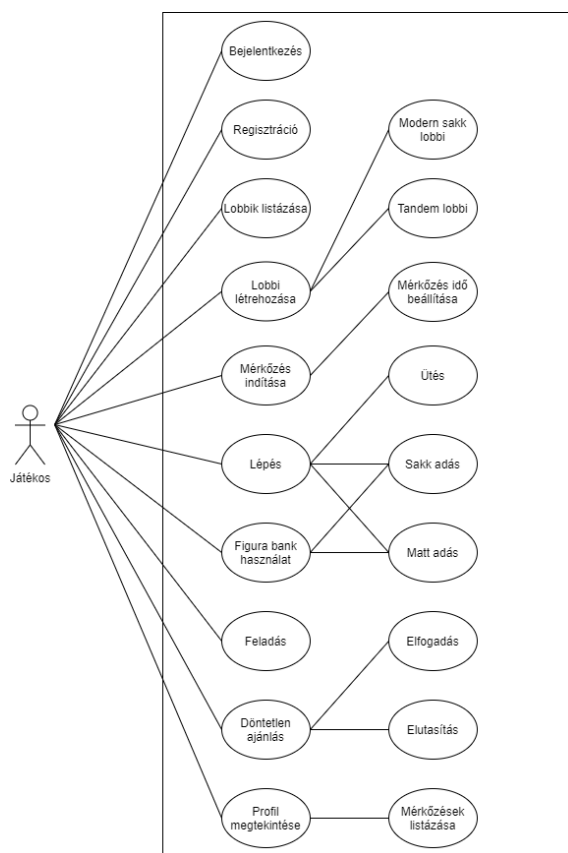
Alkalmazásomban DAO-kat használtam a játékosok és mérkőzéseik adatbázisba való mentésekor. A szerver részét képezik és az interfészeket megvalósító osztályok tartalmazzák az adatbázissal való kapcsolódáshoz szükséges információkat, mint az adatbázis neve, SQL parancsok, valamint a szükséges metódusok.

Mivel ebben a rétegben van implementálva az adatbázissal való kapcsolódás, fontos figyelmet fordítanunk az alkalmazás, és adatok védelmére. Relációs adatbázist használó alkalmazásoknál a leggyakoribb támadási forma az SQL Injection, amely során a támadók a futtató környezet számára értelmezhető kódot helyeznek el a beviteli mezőkben vagy kérésekben. Igyekeztem ennek kockázatát minél jobban csökkenteni. Az adatbázis felé irányuló parancsok konstruálásakor PreparedStatement-eket használtam, amely képes kiszűrni a kártékony kódot a szolgáltatott adatokból.

## 5. fejezet

# Az alkalmazás megvalósítása

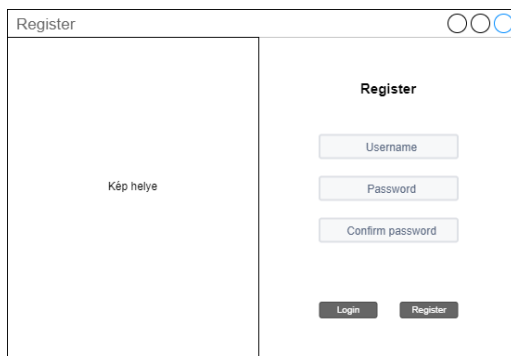
A fejezetben az alkalmazás megvalósítását foglalnám össze. A fejlesztés során igyekeztem az egyes részeket olyan módon megírni, hogy azok egyszerűen továbbfejleszthetők, bővíthetők legyenek. A fejlesztési lépések szempontjából ez elengedhetetlen volt, hiszen például a tandem teljes mértékben magába foglalja a modern sakkot. Az egyes alfejezetekben, melyek az alkalmazás funkcionalitását írják le, külön kitérek a kliens és a szerver feladataira.



5.1. ábra. Az alkalmazás Use Case diagramja

## 5.1. Authentikáció

A kliens indításakor az autentikációs felület fogadja a felhasználót. Itt lehetősége van regisztrálni, vagy amennyiben rendelkezik játékos profillal bejelentkezhet. A regisztráció során meg kell adnia egy felhasználónevet, és egy jelszót megerősítéssel. Mind a login, mind a regisztrációs felületen a beviteli mezők validálva vannak egy reguláris kifejezéssel, ami biztonsági szempontból kiszűri a nem megengedett karaktereket.



5.2. ábra. A regisztrációs felület képernyőterve

Amennyiben a megadott adatok eleget tesznek a reguláris kifejezés megszorításainak, a kliens elküldi az első üzenetét a szervernek. Az elküldésre szánt HTTP kérés POST metódusú, és a body részében tartalmazza az input mezőbe bevitt adatokat JSON formátumban. A szerver rendelkezik egy-egy autentikációs végponttal a regisztráció és a bejelentkezés számára.

Regisztráció esetében ellenőrzésre kerül, hogy létezik-e már az új felhasználó által bevitt névvel játékos, amennyiben igen 409-es státuszkódú válasz kerül neki elküldésre, ami a konfliktust jelenti a HTTP protokollban. Ha nem létezik ilyen játékos még, tehát a kérés helyes a felhasználó által megadott jelszó titkosításra kerül a BCrypt titkosító algoritmussal, majd így kerül elmentésre az adatbázisba a játékosokat tartalmazó táblába. Az algoritmus Java megvalósítását a Maven Central Repository-ből adtam hozzá a projekt függőségeihez. Az adatbázishoz való sikeres hozzáadás függvényében a szerver beállítja a válasz státuszkódját.

Bejelentkezéskor a szerver lekérdezi az adatbázisból a megadott felhasználónévvel rendelkező játékost. A lekérdezés eredményének egy felhasználót megengedett tartalmaznia, amelynek a BCrypt-el visszafejtett jelszava megegyezik a megadottal. Minden feltételnek teljesülnie kell a sikeres bejelentkezéshez. A státuszkód ennek függvényében itt is beállításra kerül a válaszban és visszaküldésre kerül a klienshez.

A kliens csak 200-as státuszkód visszaérkezésekor engedi tovább a folyamatot. Bármilyen hiba felmerülését jelzi a felhasználónak. Teljesüléskor a kliens megnyitja a web-socket kapcsolatot a szerver felé.

```
// Client - SocketController.java
public void openSocketConnection() {
    ClientManager clientManager = ClientManager.createClient();

    try {
        String url = SystemUtils.getInstance().appProps
            .getProperty("websocket_url") + "/" +
            SystemUtils.getInstance().getUser().getId();
        clientManager.connectToServer(ClientEndpoint.class, new
            URI(url));
    } catch (DeploymentException | URISyntaxException e) {
        e.printStackTrace();
    }
}

// Client - ClientEndpoint.java
@OnOpen
public void onOpen(Session session) {
    logger.info("Connected: " + session.getId());
    this.session = session;
    SystemUtils.getInstance().session = session;
    try {
        SocketMessage message = new SocketMessage(
            session.getId(),
            SystemUtils.getInstance().getUser(),
            "start",
            null
        );
        String messageString = gson.toJson(message);
        session.getBasicRemote().sendText(messageString);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

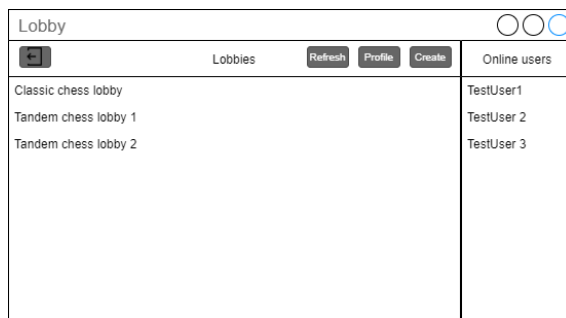
5.3. ábra. Websocket kapcsolódás autentikáció után

Az 5.3-as ábrán a kliens oldali websocket csatlakozás látható. A *SocketController* osztályban találhatóak azok a metódusok amelyek a csatornán elküldésre szánt üzeneteket konstruálják meg, valamint ez az osztály tartalmazza magát a kliens végpontot. A *ClientEndpoint* definiálja az egyes eseményekre, való reagálást, valamint a kapcsolat teljes fennállása alatt rendelkezik egy, az aktív *Session*-re való referenciával minden interakciója során. Az websocket kapcsolat eseményeit annotációkkal ellátott metódusok valósítják meg, példaképpen az itt kiemelt *OnOpen* függvénnnyel, ami a websocket kapcsolat létrejöttkor hívódik meg automatikusan.

Amennyiben a fentebb említett kapcsolódás is sikeresen végbemegy, az alkalmazás átnavigálja a felhasználót a lobbí grafikus felületre.

## 5.2. Várószobák

A websocket kapcsolat kiépülése után a kliens újabb HTTP kérést indít a szerver felé, melyben lekérdezi az aktuálisan online játékosokat és az elérhető várószobákat amelyek nincsenek még megtelve. A válaszok megérkezése után betölti a felhasználónak a grafikus felületet, ahol már láthatóak az említett adatok egy-egy listában. Mondhatni ez az alkalmazás központi része, mert a játékosok innen érik el az összes funkciót. Lehetőségük van frissíteni a listákat, ami során azonos lekérdezések futnak le, mint a felület betöltése előtt. Át tudnak navigálni a profil oldalra és ami a legfontosabb, várószobákat képesek létrehozni.



5.4. ábra. A lobb felület képernyőterve

Várószoba létrehozásakor a játékosnak választania kell a két lehetséges meccs típus közül, hogy milyen szobát kíván létrehozni. Klasszikus, modern sakk esetén a várószoba két fős, míg tandem során négy. Ilyenkor a szerver felé újabb HTTP kérés irányul, ami tartalmazza a játékos adatait és a lobb létrehozási szándékát. Ellenőrzések futnak le, amik vizsgálják, hogy a felhasználó megtalálható-e az online játékosok között, valamint, hogy esetleg nem tagja-e már egy várószobának. A feltételek vizsgálata után, ha a felhasználó kérése helyes, a lobb létrehozásra kerül, amit a szerver eltárol az elérhető várószobák listájában, a válasz pedig tartalmazza a létrehozott szoba azonosítóját. Az alkalmazás ebben az esetben átnavigálja a klienst egy másik grafikus felületre ahol várakozhat az ellenfelére. Itt lehetősége van lokális játékot indítani, ahol a partnerével egy számítógép használatával játszanak. Ilyenkor természetesen törlésre kerül az általa létrehozott várószoba.

A játékos másik lehetősége, hogy csatlakozik egy, már létrehozott várószobához. Ilyenkor azt ki kell választania a listából kattintással. A szerver felé indított HTTP kérés kapcsán hasonló ellenőrzések futnak le mint létrehozáskor. Amennyiben sikeresen hozzá tudta adni a szerver a lobbhoz, a kliens átnavigálja a játékost a várószoba felületére, ahol már láthatja ellenfelét. Az oldal frissítésével a host számára is megjelenik ellenfele, amit a kliens tárol.

Ezen a ponton megkülönböztethetünk tulajdonos(továbbiakban host) és vendég játékost. Modern sakk esetében a host színe világos az ellenfél sötét. Tandem mérkőzéskor úgy mond érkezői sorrend van érvényben, a host az egyes számú csapat világos játékosa, a második csatlakozó felhasználó a sötét csapatársa és a másik csapatban hasonlóan. Ha ebben a sorrendben megszámozzuk a játékosokat egytől négyig a párok az 1-3, 2-4 felállásban fognak alakulni. Az alkalmazás során mindig a host határozza meg játékindításkor a játékidőt, amit egy lenyíló listából tehet meg. Az idő kiválasztása után websocket csatornán keresztül a host kliense elküldi a szerver felé, hogy készen áll a játékindításra. Az

üzenet tartalmazza játékos adatait és a játékidőt. A szerver az üzenet típusát felismerve, broadcast-olja azt a többi kliens számára, köztük a vendég játékosnak. Mivel a játékosok kliense rendelkezik információval a partnerükről, amennyiben az üzenet feladója az ellenfél, feljegyzi magának, hogy ő készen áll az indításra, és beállítja a mérkőzés idejét. Hasonló módon kell eljárnia a vendég játékosnak is. Amennyiben mindkét játékos elküldte indítási szándékát az alkalmazás átnavigálja őket a sakkhöz. Tandem esetében ez annyiban módosul, hogy a szerver várja össze játékosok beérkező indítási szándékait, és ő küldi ki a start jelet.

```
// LobbyController.java
public void onClickedLobby(MouseEvent mouseEvent) throws
    IOException {
    int selectedLobby =
        lobbies.getSelectionModel().getSelectedIndex();
    Lobby lobbyToJoin = lobbiesInList.get(selectedLobby);
    int statusCode = 0;
    if (lobbyToJoin != null) {
        if (lobbyToJoin instanceof TLobby) {
            // joining tandem match
            statusCode = RESTTandemController.joinTandem((TLobby)
                lobbyToJoin);
            if (statusCode == 200) {
                WindowUtils.getInstance()
                    .switchScreen("tandem_match_making", logoutBtn, null,
                        500, 500);
            } else {
                AlertUtils.showAlert("Something went wrong!");
            }
        } else {
            // joining normal match
            statusCode = RESTLobbyController.joinLobby(lobbyToJoin);
            if (statusCode == 200) {
                SystemUtils.getInstance()
                    .setOpponent(lobbyToJoin.getHost());
                WindowUtils.getInstance()
                    .switchScreen("match_making", logoutBtn, null, 500,
                        500);
            } else {
                AlertUtils.showAlert("Something went wrong!");
            }
        }
    }
}
```

5.5. ábra. Lobbi kiválasztását kezelő függvény

Az 5.5-ös ábra az fx-controller részét képzi, a feladata lobbik kiválasztásakor, a felhasználó átnavigálása, a játékosok kezdési szándékát kezelő grafikus felületre. Ezt a folyamatot a *WindowUtils* osztály végzi minden esetben, amely dinamikusan képes fxml fájlokat betölteni a hozzájuk tartozó stílusfájllal, amennyiben az szükséges. A modern és tandem lobbikhoz való HTTP kérés indítások külön végpont felé irányulnak, viszont a szervertől visszaérkező státuszkódok jelentése azonos mind a két esetben. Hiba esetén a felhasználó felé hibaüzenet kerül megjelenítésre.

### 5.3. Modern sakk megvalósítása

A sakkjátszmák lehetősége képzi az alkalmazás fő funkcionalitását és az ehhez kapcsolódó részek komplexebbek a többinél. A grafikus felületen a játékos láthatja a saját, valamint ellenfelének nevét, a sakkórát, gombokat a különböző játékkimeneteli akciókhoz, és természetesen a táblát, amin a bábuk helyezkednek el.

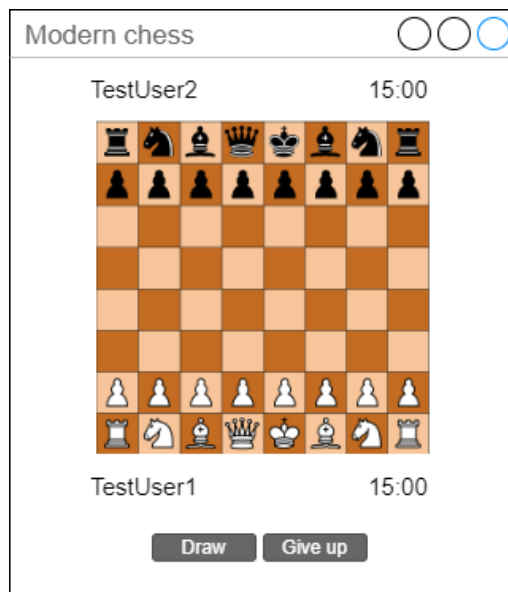
#### 5.3.1. Játék előfeltételeinek megteremtése

Ezen grafikus felület üzleti logikája két részre bontható. Az egyik rész az fx-controller, aminek a feladata, hogy aktualizálja a táblát és reagáljon a felhasználó különböző interakcióra, mint például a lépés. A másik rész sakk logikájáért felelős *BoardController*, biztosítja, hogy csak szabályos lépéseket végezheszenek a játékosok. A grafikus felület megnyitásakor az fx-controller a játékosadatok alapján beállítja a tábla mezőinek a színét, felhasználóneveket és elhelyezi a táblán a bábukat. Ezt követően jön a felhasználó számára nem látható rész, amikor a *BoardController* létrehozza a tábla objektumot. A sakktábla mezőin végighaladva felcímezi azokat a 2.3-as fejezetben leírt módon a könnyebb kommunikáció elősegítéséhez, és elhelyezi rajtuk a közös ősből öröklődő bábu objektumokat. A szeparáció célja, hogy elkülöníthető legyen egymástól a logika és a megjelenítés, követve az MVC tervezési mintát. Ezen a ponton kiemelném, hogy amennyiben lokális játékról van szó, tehát egy számítógépen játszanak a felek, kettő tábla kerül létrehozásra. A mezők címezése miatt a honnan-hova kérdésre a válasz egyértelműen meghatározható, és ebből kifolyólag optimálisabb két táblán végezni a lépéseket, majd kör cseréje esetén az egyiket elrejtteni, mint egy táblán lépésenként megváltoztatni a bábuk helyzetét, ezzel is javítva a felhasználói élményt. Az előfeltételek megteremtésének utolsó lépése a sakkóra beállítása és elindítása. Értelemszerűen a világos játékos órája indul el a grafikus felület betöltésekor.

#### 5.3.2. Lépések

A játékosok az egyes bábukra, majd a kívánt pozícióra való kattintással tudnak lépni a tábla mezőin. Lépéskor több kritériumnak is teljesülnie kell. A sakk szabályait megszegő interakciókkor a felhasználónak hibaüzenet jelenik meg. A lépések helyességének ellenőrzését a *BoardController* végzi. A sakk és matt helyzetek ellenőrzése minden lépés után lefut, hiszen előbbi a lépés helyességében játszik kulcsszerepet, utóbbi pedig a játék végét jelenti. A teljesség igénye nélkül kiragadnék pár feltételt, aminek eleget kell tегyen egy lépés.





5.6. ábra. A modern sakk játékelületének képernyőterve

- A játékos csak a saját bábujaival léphet.
- Saját bábuját nem ütheti le.
- Nem végezhet olyan lépést, amellyel sakkhelyzetbe kerülne a királya.
- Amennyiben sakkban van a királya, a lépésével el kell háritania.
- A figurák típusától függően meghatározott módon válthat velük pozíciót.

A 5.7-es ábrán a bástya *canMove* metódusának részlete látható. A függvény a közös absztrakt ősből származik, és minden leszármazottja felüldefiniálja a saját lépési lehetőségeit ellenőrizve. Ez a metódus minden esetben egy logikai értékkel tér vissza, ami jelzi, hogy a bábu elmozdulhat-e a paraméterben érkező pozícióra. A bábuk pozícióját a lépések végén a *BoardController* aktualizálja, hogy azok csak a lépésekhez szükséges logikát tartalmazzák. A metódus a kezdetekor alap ellenőrzéseket végez, mint például saját bábu ütése, vagy a sakkhelyzet megléte, esetleges kialakulása. Utóbbit egy temporális bábuáthelyezéssel teszi meg. Amennyiben a lépés következtében sakkhelyzet lesz érvényben, a lépés nem megfelelő. Ezután következik a bástya esetén a kizárólag horizontális, vagy vertikális mozgás vizsgálata. Az irányok ellenőrzésének metodikája felettébb azonos. Minden bábu esetén hasonló módon mennek végbe ezek a folyamatok a saját lépési logikájukkal. A figurák lépéseinek különbözősége miatt, a *canMove* metódus erőforrásigénye is más, különösen a matt ellenőrzésekor. Néhány helyen előzetes vizsgálatokat kellett alkalmaznom, hogy elkerüljem a memória megtelését, azaz a *StackOverflowException*-t.

```
// Rook.java
@Override
public boolean canMove(Position to) {
    // ...
    if (pos.col == to.col) {
        if (pos.row > to.row) {
            for (int i = pos.row - 1; i > to.row; i--) {
                if (board.getPiece(i, pos.col) != null) {
                    return false;
                }
            }
        } else {
            for (int i = pos.row + 1; i < to.row; i++) {
                if (board.getPiece(i, pos.col) != null) {
                    return false;
                }
            }
        }
    } else if (pos.row == to.row) {
        if (pos.col > to.col) {
            for (int i = pos.col - 1; i > to.col; i--) {
                if (board.getPiece(pos.row, i) != null) {
                    return false;
                }
            }
        } else {
            for (int i = pos.col + 1; i < to.col; i++) {
                if (board.getPiece(pos.row, i) != null) {
                    return false;
                }
            }
        }
    } else if (pos.row != to.row || pos.col != to.col) {
        return false;
    }
    return true;
}
```

5.7. ábra. Bástyá szabályos lépésének ellenőrzése

Amennyiben a lépés megfelel és sikeresen keresztül halad az ellenőrzéseken, a *Board-Controller* elvégzi a bábu áthelyezését, ütés esetén az ütött bábu levételét a tábla objektumon. Ilyenkor az fx-controller jelet kap, hogy elvégezheti a bábu mozgatását a felhasználó számára is látható módon, azaz a grafikus felületen lévő sakktáblán, ellenkező esetben hibüzenet jelenik meg. Ezt követően a kliensnek a feladata, hogy továbbítsa a lépést a partner felé. Minden, a játékmenettel kapcsolatos kommunikáció a websocket csatornán kerül elküldésre. Az socket üzenet ebben az esetben kiegészül egy lépés objektummal,

mely tartalmazza a kezdeti mezőt és az új pozíciót. Észrevehető, hogy a csatornán csak olyan üzenet haladhat át klienstől-kliensig, amivel a fogadónak annyi a dolga, hogy elvégezze a grafikus felületen és a tábla objektumon. A szerver fogadja az üzenetet, majd broadcast-olja a csatornán. Üzenet érkezésekor az egyes kliensek képesek eldönteni, hogy az adott játékosnak szólnak-e, és csak akkor végzik el.

Az alkalmazás működése ezen a ponton eltér a szekvenciális egy szálas megvalósítástól. A websocket implementáció külön szálon fut, ahonnan a felhasználói felület frissítése nem megengedett. A kliens alkalmazás az üzeneteket a *SocketController* segítségével küldi el. A vezérlő az általa tárolt *ClientEndpoint* használatával küldi ki az üzenetet a websocket csatornán. A csatorna implementációja tartalmazza a párhuzamosítást, amely nem befolyásolja az alkalmazást ezen a ponton. Üzenet érkezésekor viszont elkerülhetetlen a fő szál, más néven *Application Thread* frissítése, mivel mellékszálon fut a kezelő kódrészlet. Ezen a ponton a kliens alkalmazásnak a feladata, hogy frissítse a grafikus felületet. Mivel az üzenetre nem azonnali válaszadás kerül végrehajtásra a kommunikációt egyoldalúnak mondhatjuk. A JavaFX szerencsére megoldást ad az ilyen helyzetekre a Platform objektumon keresztül, a *runLater* metódussal. Ez egy *Runnable*-t vár paraméterként, és segítségével frissíthetjük a grafikus felületet, amennyiben az előzetes ellenőrzéseknek megfelelt az üzenet. A *WindowUtils* menedzser osztály tárolja a sakkhoz tartozó fx-controller referenciáját és ezáltal meghívásra kerülhet a *makeMove* metódus, amely a lépéseket kezeli.

### 5.3.3. A sakkóra

Az alkalmazás másik pontja ahol párhuzamosítást használtam a sakkóra. A megvalósításra több lehetőség is adott, én a JavaFX által biztosított *Timeline*-t használtam, amely az *Animation* osztályból öröklődik.

Paraméterként egy *KeyFrame* objektumot vár, amelyben 1 másodperces intervallumot beállítva a valós idő múlást szimulálhatjuk. Az *ChessClock* osztályból példány létrehozásakor paraméterben átadásra kerül a host játékos által beállított mérkőzés idő percben, és inicializálásra kerül egy-egy, a másodperceket nyilvántartó változó is. A *Timeline handle* metódusában az idő múlása és a változók aktualizálása van megvalósítva. A másodpercek 60-ról indulnak majd 1 másodperces intervallumonként dekrementálva lesznek és amennyiben eléri a nullát, visszaállításra kerülnek 60-ra, a perc pedig dekrementálódik egyel. Amennyiben a perc és a másodperc számlálók is lenullázódtak, a játékosnak lejárt az ideje.

Az osztály rendelkezik még két *StringProperty* adattaggal, amelyben a felhasználók hátralévő percei és másodpercei vannak konkatenálva olvasható formában. Ez a típus a JavaFX Beans osztályából származik, és az egyszerű String objektumokkal szemben rendelkezik a kötés, angolul Binding képességével. Segítségével, különböző kötések(egyirányú, többirányú) alkalmazhatunk például az egyes grafikus interfészen megjelenő elemekhez. Ezt alkalmazva a felhasználói felületen elhelyezkedő Text objektumokhoz hozzákötöttem a *ChessClock* osztály *StringProperty* adattagjait, hogy a felhasználók láthassák a hátralévő idejüket.

### 5.3.4. A mérkőzések vége és mentése

Most, hogy a lépések megtételének metodikája ismertetett, szükséges szót ejtenem a mérkőzések kimeneteleiről, és azok körülményeiről. Lépések elküldésekor, valamint megtételekor is lefut a sakk és matt helyzetek ellenőrzése. A sakk helyzet vizsgálata a tábla mátrixának egy bejárása. Ellenőrzésekor azt vizsgáljuk minden ellenséges bábura, hogy szabályosan *tud-e oda lépni ahol a királyunk tartózkodik*, amennyiben igen fennáll a sakk-helyzet. Ha ez teljesül, vizsgálatra kerül a matthelyzet is, amely során az összes bábu, minden lehetséges lépéslehetősége szimulálva van. Ha a helyzetet nem oldja meg semmilyen manőver, matt van érvényben, ami jelezve van a felhasználó felé is.

Végződhet még a mérkőzés feladással, vagy döntetlennel. A feladás értelemszerűen egyszerűbb a működés szempontjából: websocket csatornán elküldésre kerül egy üzenet, mely tartalmazza a játékos döntését. Döntetlen esetén először hasonló módon egy socket üzenet formájában megkapja az érintett kliens az üzenetet, annak típusából ismerve pedig felugró ablakban jeleníti meg a felhasználónak játékosát szándékát. Ezt tetszőlegesen elutasítja, vagy elfogadhatja, mint hagyományos körülmények között. A pozitív vagy a negatív válasz gombjára kattintva visszaküldésre kerül a döntés. Ilyenkor a döntetlent ajánló felhasználónak is megjelenik egy üzenet, ami tájékoztatja ellenfele válaszáról.

A mérkőzés végét jelentheti még a játékosidő felélése. Az éppen aktuális körből következőtethető a győztes játékos, és a *Timeline* megállításra kerül. Az idő lejárt felugró ablakban közölve van a felhasználók felé.

A mérkőzések végét jelentő események bekövetkezésekor szükséges, hogy azokat elementsük egy konzisztens tárolóba, ami alkalmazásom esetében az SQLite adatbázist jelenti. A játék végének detektálása után, modern sakk esetében mindig a host játékos, azaz a világos fogja elküldeni a mérkőzés eredményét a szervernek. Ezt HTTP kéréssel teszi meg, hogy ne terhelje a websocket csatornát. A parti kezdetekor és az óra indításkor a központi menedzser objektum egy, az adatbázisban tárolt formátumban elmenti a start időpontját, a mérkőzés végén pedig feljegyzi hasonló módon a zárást. A HTTP kérés tartalmazza a két játékos azonosítóját, a győztest, és az időpontokat. Abból kifolyólag, hogy az objektumokat GSON segítségével konvertálja át a kliens, a szervernek a kérés megérkezése után csak vissza kell alakítania egy paranccsal, és még a dátumok esetében is helyes marad a formátum, míg ez string műveletekkel nehezebb feladatot jelentene. A felhasználók azonosítása után megtörténik a mentés az adatbázisba, és ennek függvényében a kliens felé beállításra kerül a státuszkód. A kliens nem megfelelő státusz érkezésekor hibáüzenet jelenít meg, a mentés sikertelenségéről.

## 5.4. Tandem sakk megvalósítása

Munkám eredményének talán minden szempontból érdekesebb része a tandem sakk. A variáns technikailag magába foglalja a modern sakk szabályait, viszont a plusz szabályok megjelenésével és a csapatban való játékkal egy új élményt hozhat a végfelhasználóknak. Az alfejezetben leginkább a modern sakktól való eltérésektől és az újabb funkciók megjelenéséről írnék.



5.8. ábra. A tandem sakk játékelülete

#### 5.4.1. A figurák bankja

A játék szabályaihoz igazítva a leütött figurákat a felhasználók el tudják helyezni a táblájukon a meghatározott szabályok szerint. Ehhez szükség van egy tárolóra, ami nyilván tartja a felhasználható figurákat. A *figura bank* kifejezést találtam a funkcionalitásnak a legjobban hozzáillőnek, és az alkalmazásnak is ezen a néven képzí részét. A hagyományos játékmenetből kifolyólag a bankban lévő bábuakat minden játékosnak látnia kell, beleértve az ellenfelét is. A megvalósítás során ezt elsődlegesnek tartottam, hogy a játékélmény és a szabályok ne térjenek el a valóstól. A várószobába való csatlakozáskor, valamint a játék közben a felhasználók láthatják, hogy melyik csapat részét képezik. Ezek az egyértelműség érdekében színekkel azonosíthatók, a zöld és a piros csapat méri össze erejét egymással. A bankok elhelyezése közvetlenül a tábla mellett található az egyszerűbb használatért és háttérük jelzi a felhasználók felé, hogy melyik csapathoz tartoznak. Az ütések analógiája egyezik a modern sakkéval. A kliensek ugyan úgy a saját adataikat és a lépést tartalmazva socket üzenetet küldenek a szerver felé amit az broadcast-ol a többi kliensnek. Tandem esetében itt a hálózati kommunikáció a lépéssel kapcsolatban nem áll meg viszont, a csapatok másik tagjának is szüksége van arra, hogy a bank aktualizálva legyen a táblán leütött bábuval. Ehhez a kliens, aki az ütet végezte kiküld websocket csatornán egy újabb üzenetet, amely tartalmazza, hogy melyik csapat bankjába és milyen típusú bábu került be. Ilyenkor egy plusz ellenőrzés szükséges az eddigi üzenetekhez képest, hogy a tandem tagja-e a játékos, aki az üzenetet küldte. Amennyiben igen, az üzenet további tartalmából aktualizálhatja a bankokat. Ilyenkor az adott bábutípus alatt megjelenik egy 1-es, ami tudatja a felhasználóval, hogy a másik táblán ütés történt. A küldő félénél a frissítés lokálisan történik meg.

### 5.4.2. A leütött figurák elhelyezése

A tandem esetén bankból való bábuk elhelyezése a táblára, ugyan úgy lépésnek számít, ebből kifolyólag számos megkötés vonatkozik rá. Az elhelyezést hasonló módon teheti meg a játékos, mint amikor modern sakkban egy bábuval lépne, a bábu majd a kiválasztott mezőre való kattintással. A különböző feltételek ellenőrzése a bábu kiválasztásakor veszik kezdetüket. Fontos megjegyezni, hogy ebben az esetben az egyes elhelyezési szabálysértések detektálásánál informatívabb hibaüzenet adható, mint egy hagyományos lépés esetében, mivel a sakk vagy a matt ellenőrzése egy zártabb, rekurzívabb ellenőrzési struktúrát alkot. Ettől elvonatkoztatva ez szükséges is, mivel a tandem sakkot kevesebben ismerik, ezáltal ha a felhasználók hibákba ütköznek, megtanulhatják a szabályokat. Az ellenőrzés szempontjai a következők:

- Csak saját bankból való bábu lerakása engedélyezett.
- A játékos, színétől ellentétes bábukat helyezhet csak el.
- Amennyiben sakkban van, úgy kell elhelyeznie a bábut, hogy a sakkhelyzetet elhárítsa.

Az 5.9-es ábrán a bankból való bábulerakás ellenőrzései láthatók. A kör meghatározása után a `javafx` felület által szolgáltatott esemény(`MouseEvent`) célpontjának(`getTarget()`) vizsgálata után, amennyiben egy `ImageView` lett kiválasztva, annak azonosítójából kiemelhető a csapatot jelző karaktersorozat. Ezt követi a színek, és a lerakási lehetőség vizsgálata. A metódus végén látható ellenőrzésben, a `ColoredPieceType` enumeráció tartalmazza a bábutípusokat színnel ellátva és ennek `valueOf` metódusa segítségével meghatározható a lerakni kívánt bábu típusa, olyan módon, ahogy a `BoardController` tárolja azt.

A `BoardController` osztály rendelkezik egy `TandemController` adattaggal, amely a tandem mérkőzések során kerül inicializálásra. A lényeges adattagjai az osztálynak két `Map`, amelyek az egyes bábutípusokhoz tárolják, hogy hány darab elérhető belőlük. Ezek az interakciók során átadásra kerülnek az `fx-controller`-nek. Kezdetben ezek természetesen nullák. Az osztály feladata a darabszámok aktualizálása, és annak eldöntése, hogy az adott játékos elhelyezheti-e a bábut. Ehhez a központi menedzser objektum, ellenfelekről tárolt információját is felhasználja. Amennyiben egy bankhoz adás, vagy elvétel sikeres volt, az osztály tovább küldi ezt a websocket csatornán a szerver felé.

```
// ChessController.java
public void bankPieceClicked(MouseEvent mouseEvent) {
    if (!boardController.isPlayerTurn()) {
        AlertUtils.showAlert("Not your turn!");
    }
    String team = null;
    if (mouseEvent.getTarget() instanceof ImageView) {
        team = ((ImageView)
            mouseEvent.getTarget()).getId().substring(0, 1);
    }
    if (team == null) {
        return;
    }
    boolean isHostTeam = false;
    Player user = SystemUtils.getInstance().getUser();
    TLobby current = SystemUtils.getInstance().getCurrentTandem();
    if (user.getId() == current.getHostTeam().get(0).getId() ||
        user.getId() == current.getHostTeam().get(1).getId()) {
        isHostTeam = true;
    }
    if ((isHostTeam && team.equals("O")) || (!isHostTeam &&
        team.equals("H"))) {
        AlertUtils.showAlert("That chess piece belongs to your
            opponent's bank!");
        return;
    }
    boolean isPieceWhite = ((ImageView)
        mouseEvent.getTarget()).getId().contains("WHITE");
    if (SystemUtils.getInstance().isWhite() != isPieceWhite) {
        AlertUtils.showAlert("Only your teammate can place that
            chess piece!");
        return;
    }
    ColoredPieceType pieceToPlace =
        ColoredPieceType.valueOf(((ImageView)
            mouseEvent.getTarget()).getId().substring(2));

    if
        (boardController.tandemController.getPieceCount(pieceToPlace,
            isHostTeam) == 0) {
        return;
    }
    tandemPick = pieceToPlace;
}
```

5.9. ábra. Tandem bábulerakás kezelése

## 5.5. Játékos statisztikák

A felhasználónak lehetősége van a várószoba felületről megnyitni a profilját, ahol láthatja az általa játszott mérkőzések számát, valamint azt, hogy melyik, milyen kimenetellel zárult. Innen tovább tud navigálni egy olyan felületre is, ahol táblázatos megjelenítéssel tanulmányozhatja a mérkőzések részletesebb adatait, mint például a partner neve, vagy a mérkőzés kezdő és záró időpontja. A felhasználó egy gomb segítségével válthat, hogy milyen típusú mérkőzéseket szeretne látni.

A felületekre navigáláskor itt is HTTP kérések futnak le, klasszikus, és tandem sakk-eredményeket szolgáltató végpontok felé. Kérés érkezésekor adatbázisból való lekérdezés fut le, amely utána konvertálásra kerül. A visszaküldésre szánt mérkőzés objektumokból ún. *viewmodel* készül, melynek előnye, hogy a kliens számára csak olyan információ érkezik, amely nem tartalmaz részéről felesleges, az alkalmazás szempontjából pedig szenzitív adatokat, például azonosítókat.



## 6. fejezet

# Tesztelés és minősbiztosítás

Alkalmazások fejlesztése során nagy figyelmet kell fordítanunk a produktum tesztelésre is, különben annak minősége nem lesz megfelelő és a felhasználók elégedetlenségét válthatjuk ki, továbbá biztonsági rések keletkezhetnek.

Munkám során folyamatosan teszteltem az alkalmazást, annak új funkcióit amikor elkészültem velük. A manuális tesztelést részesítettem előnyben, mind a sakkot és a hálózati funkciókat tekintve. Az ellenőrzések az alkalmazás előrehaladtával nehezebbé váltak, ezért egyes fázisoknál külön időt szántam arra, hogy egy másik számítógépre build-elve végighaladtam a funkciókon és amennyiben hibát találtam feljegyeztem azt. Ezt követően a hibákat development környezetben javítottam.

### 6.1. Egy számítógépen való tesztelés

Az fejlesztés során az egy számítógépen való manuális tesztelés volt a legszámottevőbb. Amíg az új funkciókat fejlesztettem, a szerver és a kliens természetesen egy számítógépen működtek, ahol a HTTP kérések átfutási ideje gyors, viszont amennyiben nem egy fizikai eszközön futnak, ez az időintervallum megnőhet, ami nagy hibaforrási lehetőség. A tesztelési fázisok során ezért a szervert minden esetben másik számítógépen működtettem. Az ilyen alkalmakat követően lehetőségem volt alakítani a kérések felépítésén, és a hálózati kommunikáción általában. Az általam használt webes megvalósításokban (webszerver és websocket) a kérésekre a válaszok bevárásra kerültek, ami nagy mértékben megkönnyítette a fejlesztési folyamatot.

Az egy számítógépen történő sakkozás volt az első elkészült sakk funkció, amelynél egy kliens és a szerver futtatása természetesen elegendő volt. Ezt követte a két játékos, majd a tandem funkciók megvalósítása, ahol már több klienst volt szükséges futtatnom egyszerre. Ezen a ponton vált lényegessé, hogy a felhasználókról egyértelműen beazonosítható legyen, hogy már csatlakozva vannak-e a szerverhez, más szóval egy felhasználó online-e. Erre olyan hiba hívta fel a figyelmemet, amikor figyelmetlenségből ugyan azzal a felhasználóval jelentkeztem be kétszer, és ezt követően a HTTP kérések nem a megfelelő helyre kerültek elküldésre.

További hibaforrást találtam a dialógusablakok *Cancel* gombjánál. Nem megfelelően használtam a dialógusablak visszatérési értékét, ami miatt különböző kivételek dobódtak, a választási lehetőségek funkcionalitásától függően. A JavaFX dokumentáció olvasása

után találtam rá az *Optional* típusra. Ennek *isPresent* metódusa képes egyértelműen meghatározni, hogy választás történt-e, vagy egyszerű gombbal történő bezárás.

## 6.2. Több számítógépen való tesztelés

Az egymással kommunikáló alkalmazások általában külön eszközökön futnak, mivel a felhasználók fizikai lokációja eltérő. Alkalmazásom fő funkcionalitása a *multiplayer* sakk, ezért a hasznosságát is több a eszközön való használat meríti ki. A tesztelések során külön eszközökön működött az alkalmazás egy hálózaton.

Először csak a két személyes sakkot teszteltem 2 külön gépen, még a tandem elkészülte előtt. Ezen a ponton már lényegesnek tartottam bevonni nem csak Unix alapú rendszert, hanem a szélesebb körben használt Windows-t is. A build-et követően az alkalmazás másik számítógépre való áthelyezésekor ütközött ki egy hiba a Windows-t futtató gépen. A websocket implementáció amit felhasználtam, hibát jelzett a bejelentkezéskor. A hibát az okozta, hogy míg a Java nyelv portabilis, ez a megvalósítás további fájlokat igényelt az operációs rendszer családokat tekintve. A két család a Windows és az Unix alapú rendszerek. A jar készítése során a függőségek közé külön fel kellettennem előbbire a dll, utóbbira pedig az *so* fájlokat. A hiba felkutatása után újrafuttattam a build-eket és ezután az alkalmazás már működött az egyes operációs rendszereken, a típusukhoz tartozó extra fájlokkal. MacOS-t futtató számítógépen nem volt lehetőségem tesztelni az alkalmazást, viszont mivel Unix alapú rendszerről van szó ugyanúgy az *so* fájlokat tartalmazó build-elt jar-t kell futtatnunk rajta.

A tandem funkció elkészültekor, felkerestem az ismerőseimet, hogy valós szituációban tudjuk manuálisan tesztelni a programot. A tesztgépek között voltak Windows-t és Linux-ot futtató gépek is. A szerver indítása után, létrehoztak maguknak felhasználókat, majd lejátszottunk egy mérkőzést 2 fős csapatokban. Console-os módban kértem, hogy indítsák az alkalmazást, az esetleges hibák(például *NullPointerException*) könnyebb felderítése érdekében. Az első ilyen alkalomkor a figura bankkal kapcsolatban jött elő egy hiba, amely során a bábu elhelyezést végző játékosnál a grafikus felület nem frissült megfelelően. A problémát a hálózati kommunikációban találtam meg, majd javítottam. A tesztelés során megkértem őket arra, hogy próbálják ki az alkalmazás további funkcióit is és adjanak visszajelzést a kezelhetőségről, valamint, hogy tapasztaltak-e egyéb problémákat a program használata közben. A találkozót követően a szerencsére kevés, felmerülő hibákat javítottam, majd egy pár nap kihagyással ismét felkértem őket egy második manuális tesztelésre. Ezen az alkalmon már nem találtak hibákat és az alkalmazást is egyszerűen kezelték.

## 7. fejezet

# Összefoglalás

Szakdolgozatom kezdetekor célként tűztem ki magam elé egy olyan alkalmazás megírását, melynek mérete, robosztussága és funkcionalitása túlmutat az általam megismerteken, amellett, hogy a későbbiekben hasznos eszközként szolgáljon a virtuális sakkpartik lebonyolításában. Az általam tervezett funkciókat sikerült megvalósítanom, miközben számos pozitív és tanulságos élménnyel gazdagodtam. Munkám során törekedtem arra, hogy a készített kód átlátható és könnyen értelmezhető legyen. A fejlesztés során számos új technológiát és technikát volt szerencsém elsajátítani. Végeredményben egy olyan alkalmazást sikerült készítenem, amely lehetőséget ad a sakkot kedvelőknek, hogy a hagyományos, fizikai partikhoz nagyon hasonló módon tudják összemérni tudásukat egy közismert, valamint kisebb réteg számára ismerős variánsban.

Igyekeztem a fejlesztést olyan módon végezni, hogy az elkészült alkalmazás a későbbiekben igény szerint bővíthető legyen új funkciókkal. Az első ilyen lehetőség lehetne a szerver rész kihelyezése egy virtuális privát szerverre, melynek segítségével internet kapcsolat mellett bárholnan lehetne mérkőzéseket indítani.

Ismerőseim, barátaim érdeklődését felkeltette az alkalmazásom, miután felkértem őket a tesztelésre. Számomra ez egy felettébb pozitív élmény volt, és a program befejeztével több alkalommal is tartottunk összejöveteleket, hogy modern, valamint tandem sakkozzunk. Ezen a ponton éreztem azt, hogy sikerült elérnem a célokat amit a szakdolgozat kezdetekor kitűztem magam elé.

# Nyilatkozat

Alulírott Programtervező Informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, Programtervező Informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2021. május 21.

.....  
aláírás

# Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Dr. Siket Istvánnak, aki szakmai értelmével, hasznos magyarázataival és a konzultációk során biztosított tanácsaival segítséget nyújtott szakdolgozatom elkészüléséhez.

Köszönettel tartozom Kovács Ádámnak aki segítségemre volt a szakdolgozat hálózati részének a megtervezésében.

Hálával tartozom továbbá szüleimnek, akik a tanulmányaim során végig támogattak és mellettem álltak.

# Irodalomjegyzék

- [1] Bobby Fischer, *My 60 Memorable Games*, Batsford, London, 2008.
- [2] Garry Kasparov, *My Great Predecessors, Part I.*, Everyman Chess, London, 2003.
- [3] Harold Murray, *A History of Chess*, Oxford University Press, London, 1913.
- [4] Bruce Eckel, *Thinking in Java*, Prentice Hall, New Jersey, 2006
- [5] Sharan, Kishori, *Learn JavaFX 8*, Apress, New York, 2015
- [6] Sakk történelme:  
<http://www.globalchessfestival.com/hu/sakk/sakktortenet>
- [7] Java. <https://docs.oracle.com/javase/7/docs/api/>
- [8] JavaFX. <https://openjfx.io/javadoc/16/>
- [9] Maven. <https://maven.apache.org/guides/>
- [10] JDBC.  
<https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>
- [11] SQLite. <https://www.sqlite.org/docs.html>
- [12] GSON. <https://sites.google.com/site/gson/gson-user-guide>
- [13] Websocket.  
<https://www.oracle.com/technical-resources/articles/java/jsr356.html>
- [14] Sakktábla képe.  
[https://commons.wikimedia.org/wiki/File:AAA\\_SVG\\_Chessboard\\_and\\_chess\\_pieces\\_02.svg](https://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg)