**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Programozási nyelvek és Fordítóprogramok Tanszék**

Bang! társasjáték megvalósítása Javaban

Szerző: Külső témavezető Belső konzulens:

Ábrahám Erik Kovács Norbert Zsolt Kozsik Tamás

Programtervező informatikus BSc. Java fejlesztő Docens, PhD

**Budapest, 2025**



Absztrakt

A dolgozat a Bang! nevű kártyajáték számítógépes, Java nyelven készült asztali változatának elkészítését mutatja be. A projekt célja az volt, hogy a játék mechanikáit – mint például a kártyahúzás, kártyák kijátszása, speciális karakterképességek, illetve a játékosok közötti interakció – digitális formában, felhasználóbarát grafikus felületen valósítsam meg. A program háromrétegű architektúrára épül, külön kezelve a felhasználói felületet, az üzleti logikát és az adatkezelést. A fejlesztés során objektumorientált megközelítést alkalmaztam, a kártyák és karakterek külön osztályokban kerültek implementálásra, figyelembe véve a kiterjeszthetőséget és új funkciók hozzáadásának lehetőségét. Az elkészült alkalmazás lehetőséget biztosít új játék indítására, karakterek és szerepek kiosztására, valamint a körök során zajló akciók (támadás, védekezés, kártyakezelés) lebonyolítására. Az eredmény egy működő, tesztelhető és jövőben továbbfejleszthető játék, amely akár egy mélyebb mesterséges intelligencia, akár hálózati többjátékos mód integrálására is alkalmas alapot nyújt.

Tartalomjegyzék

[1. Bevezetés 1](#_Toc197017570)

[2. Felhasználói dokumentáció 2](#_Toc197017571)

[2.1. A játék rövid leírása 2](#_Toc197017572)

[2.2. Játékszabályok bemutatása 2](#_Toc197017573)

[2.2.1. A lehetséges szerepkörök, céljaik és győzelmi feltételeik: 2](#_Toc197017574)

[2.2.2. A kör menete 3](#_Toc197017575)

[2.2.3. Kártyatípusok 3](#_Toc197017576)

[2.2.4. Karakterek 5](#_Toc197017577)

[2.2.5. Játékosok közötti távolság 6](#_Toc197017578)

[2.2.6. Speciális szabályok 6](#_Toc197017579)

[2.3. A Bang! letöltése, telepítése 6](#_Toc197017580)

[2.3.1. Telepített Java verzió ellenőrzése 6](#_Toc197017581)

[2.3.2. Java JDK 21 telepítése 7](#_Toc197017582)

[2.3.3. A játék letöltése és indítása 7](#_Toc197017583)

[2.3.4. A program fordítása és futtatása 7](#_Toc197017584)

[2.4. Hardver- és szoftverigény 7](#_Toc197017585)

[2.4.1. Ajánlott hardverkonfiguráció 7](#_Toc197017586)

[2.4.2. Szoftverkövetelmények 8](#_Toc197017587)

[2.4.3. Egyéb megjegyzések 8](#_Toc197017588)

[2.5. Játékmenet 8](#_Toc197017589)

[2.5.1. A játék áttekintése 8](#_Toc197017590)

[2.5.2. Új játék indítása 8](#_Toc197017591)

[2.5.3. A játéktér felépítése 10](#_Toc197017592)

[2.5.4. A körök működése és fázisai 12](#_Toc197017593)

[2.5.5. Győzelmi feltételek és a játék vége 13](#_Toc197017594)

[3. Fejlesztői dokumentáció 15](#_Toc197017595)

[3.1. A szoftver rövid leírása, specifikáció 15](#_Toc197017596)

[3.1.1. A szoftver működésének rövid összefoglalása: 15](#_Toc197017597)

[3.2. Követelményelemzés 16](#_Toc197017598)

[3.2.1. Funkcionális követelmények 16](#_Toc197017599)

[3.2.2. Nem funkcionális követelmények 16](#_Toc197017600)

[3.3. Tervezés 17](#_Toc197017601)

[3.3.1. A szoftver architektúrája 17](#_Toc197017602)

[3.3.2. Use-case diagram 18](#_Toc197017603)

[3.3.3. Osztály- és objektumdiagram 19](#_Toc197017604)

[3.3.4. Sequence diagram 30](#_Toc197017605)

[3.4. Megvalósítás 31](#_Toc197017606)

[3.4.1. Használt technológiák 31](#_Toc197017607)

[3.5. Projekt szerkezete, CI/CD 32](#_Toc197017608)

[3.5.1. Projekt szerkezete 32](#_Toc197017609)

[3.5.2. CI/CD folyamat 32](#_Toc197017610)

[3.6. Kódbázis felépítése, könyvtárszerkezet, forrásfájlok 33](#_Toc197017611)

[3.6.1. Főkategóriák 33](#_Toc197017612)

[3.6.2. Általános jellemzők 35](#_Toc197017613)

[3.6.3. Kód egyes elemeinek bemutatása 35](#_Toc197017614)

[3.6.4. Bonyolultabb kódrészletek elmagyarázása 44](#_Toc197017615)

[3.7. Tesztelés 49](#_Toc197017616)

[3.7.1. Használt tesztelési módszerek, tesztautomatizálás 49](#_Toc197017617)

[3.7.2. Unit- tesztek összefoglalása 50](#_Toc197017618)

[3.7.3. Integrációs tesztek 52](#_Toc197017619)

[3.7.4. Acceptance tesztek 53](#_Toc197017620)

[4. Összefoglaló 56](#_Toc197017621)

[4.1. Konklúzió 56](#_Toc197017622)

[4.2. Továbbfejlesztési lehetőségek 56](#_Toc197017623)

[5. Irodalomjegyzék 58](#_Toc197017624)

[6. Melléklet 59](#_Toc197017625)

# Bevezetés

A számítógépes játékfejlesztés a szoftverfejlesztés egyik legsokoldalúbb és legizgalmasabb területe, amely egyesíti az algoritmikus gondolkodást, a vizuális megjelenítést, valamint a felhasználói élmény optimalizálásának kihívásait. A szakdolgozatom célja egy népszerű társasjáték, a *Bang!* digitális, asztali változatának megvalósítása Java nyelven. A projekt során egy önállóan futtatható, grafikus felhasználói felülettel rendelkező alkalmazást készítettem, amely követi az eredeti játék szabályait[1], és lehetőséget biztosít egyjátékos (AI elleni) játékra. A *Bang!* játék különlegessége, hogy a játékosok rejtett szerepek mentén, eltérő győzelmi feltételekkel versenyeznek, így a játékmenet változatos, és taktikai döntéseket igényel. A program célja ennek a játékmechanikának a pontos, de ugyanakkor élvezetes számítógépes adaptációja. A felhasználó egy grafikus felületen keresztül irányíthatja a karakterét, kártyákat húzhat, kijátszhat, támadhat vagy védekezhet. A játék során pedig a mesterséges intelligencia vezérelte ellenfelek ellen küzdhet. A fejlesztés során a Java programozási nyelvet és a Swing[2] grafikus felületkezelő könyvtárat választottam, mert ezek jól illeszkednek az asztali alkalmazásokhoz, illetve a Java objektumorientált szemlélete ideálissá teszi az összetettebb játéklogika kezelésére is. A játék fejlesztése során nagy hangsúlyt fektettem a rétegzett architektúra kialakítására, a kód újra felhasználhatóságára és a tesztelhetőségre. A dolgozat célja, hogy bemutassa a játék működését a felhasználó szemszögéből, majd részletesen feltárja a szoftverfejlesztési folyamatokat, az architektúrát, a program szerkezetét és a megvalósítás során használt technológiákat. Az utolsó fejezetben a játék teszteléséről és a továbbfejlesztési lehetőségekről esik szó.

# Felhasználói dokumentáció

## A játék rövid leírása

A Bang! egy vadnyugati tematikájú, szerepalapú társasjáték, amelyet digitális formában valósítottam meg Java nyelven. A játék során a játékosok rejtett szerepek szerint versenyeznek egymással: Sheriff, Renegát, Banditák vagy Sheriff-helyettes szerepben. Minden játékos célja eltérő, ami izgalmas és taktikus játékmenetet eredményez. Az alkalmazás egy asztali (desktop) környezetben futó, grafikus felhasználói felülettel ellátott Java program, amely lehetővé teszi a játék egyjátékos módját. Az egyjátékos módban a felhasználó számítógép által irányított (bot) játékosok ellen játszik, akik egyszerű mesterséges intelligenciával rendelkeznek, és saját, előre meghatározott stratégiájuk mentén cselekednek. A program a valódi társasjáték szabályait követi, beleértve a kártyák húzását, kijátszását, a különféle támadó és védekező kártyák használatát, valamint az életerő (HP) kezelését. A játékos karaktereket választ, amelyek mindegyike egyedi képességgel rendelkezik, ez tovább fokozza a stratégiai lehetőségeket. A digitális verzió célja, hogy a társasjáték hangulatát és mechanikáját a számítógépen keresztül is elérhetővé tegye, lehetőséget adva gyakorlásra, játékelemzésre, vagy egyszerű szórakozásra akkor is, ha nincs kéznél elég ember egy fizikai játékpartihoz.

## Játékszabályok bemutatása

A Bang! egyjátékos változatában a felhasználó egy választott, vagy véletlenszerű szerepű karaktert irányít, és mesterséges intelligencia által vezérelt ellenfelekkel (botokkal) mérkőzik meg. A játék célja az, hogy a játékos túlélje az összes ellenség támadását, és győzelemre vezesse karakterét azáltal, hogy minden ellenfelet legyőz.

### A lehetséges szerepkörök, céljaik és győzelmi feltételeik:

A játék elején a játékos kiválaszthat egy tetszőleges szerepet, vagy rábízhatja a programra, hogy osszon ki neki egy véletlenszerű szerepet (Sheriff, Bandita, Renegát, Sheriffhelyettes). Az összes többi szerep botokra van kiosztva, és a játékos a játék kezdetén nem tudja, ki milyen szerepben van, kivéve a Sheriffet, mert az ő szerepe nyilvános. A szerepkörök céljai:

* **Sheriff**  
  Célja: Minden Bandita és a Renegát likvidálása.  
  Ha a Sheriff túléli a játékot, és már csak ő és a Sheriff-helyettesek maradnak, akkor a Sheriff győz.
* **Bandita**  
  Célja: Megölni a Sheriffet.

Ha a Sheriff meghal, és a Renegáton kívül van még élő játkos, akkor a Banditák nyernek.

* **Renegát**  
  Célja: Ő legyen az utolsó túlélő játékos.

Előbb el kell távolítania minden más játékost, legvégül a Sheriffet. Ha a Sheriff előtt bárki más túléli, a Renegát automatikusan veszít.

* **Sheriffhelyettes**  
  Célja: Megvédeni a Sheriffet, és segíteni a Banditák és a Renegát kiiktatásában.  
  Ha a Sheriff győz, a Sheriff-helyettesek is nyernek, de ha a Sheriff meghal, ők is elveszítik a játékot.

### A kör menete

A játék körökre osztott. Egy körben minden játékos (bot is) végrehajtja a következő lépéseket:

1. **Kártyahúzás** – A kör elején minden[[1]](#footnote-2) játékos két kártyát húz a pakliból.
2. **Kártyák kijátszása** – A játékos tetszőleges számú kártyát kijátszhat, a szabályoknak megfelelően (pl. egyszerre csak egy Bang! kártya játszható ki).
3. **Kártyadobás** – A kör végén a játékos annyi kártyát tarthat a kezében, ahány életereje (HP) van. A fölösleget el kell dobni.

### Kártyatípusok

A játékban különböző típusú kártyák találhatók, melyek a harc és védekezés alapját képezik.

* **Bang!** – Elsősorban más játékosokra lehet vele lőni.
* **Nem talált!** -Elsősorban a Bang! hatását semlegesíti.
* **Sör** – Gyógyít egy életerőpontot, de csak ha a játékos nem maximális életerőn van. És kettőnél több játékos van életben.
* **Kocsma – Mindenkit gyógyít egy életpontot.**
* **Gatling** – Minden ellenfél egy Nem talált! kártyával védekezhet, vagy veszít egy életpontot.
* **Indiánok!** – Minden ellenfél egy Bang! kártyával védekezhet, vagy veszít egy életpontot.
* **Párbaj – A célpontnak Bang! kártyát kell kijátszania, majd a Párbaj lapot kijátszó játékosnak. Aki először nem tud Bang! lapot kijátszani, az veszít egy életpontot.**
* **Cat Balou – Eldobat egy kártyát a célponttal.**
* **Pánik** – Elvesz egy kártyát a hatótávolságon belüli célponttól.
* **Postakocsi – Húzhat 2 kártyát a kijátszó játékos.**
* **Wells Fargo – Húzhat 3 kártyát a kijátszó játékos.**
* **Szatócsbolt – Annyi lapból, ahány játékos életben van, mindenki választ egyet magának a kör haladása szerint.**
* **Musztáng** - A többi játékos eggyel messzebbről látja, míg aktív ez a lap.
* **Távcső** - A többi játékost eggyel közelebbről látja, míg aktív ez a lap.
* **Hordó** – Ha aktív ez a lap és meglövik egy Bang! kártyával, akkor húznia kell a pakli tetejéről és eldobni. Ha ez a lap kőr, akkor sikeresen blokkolta a Bang! lapot.
* **Dinamit** – Ha aktív a dinamit a játékos köre kezdetén, akkor húznia kell és eldobnia a lapot. Ha a lap pikk 2-9 közti magassággal, akkor a dinamit robban és 3 életpontot veszít a játékos.
* **Börtön** – Ha aktív a börtön a játékos köre kezdetén, akkor húznia kell és eldobnia a lapot. Ha a lap kőr, akkor kiszabadul és lejátssza a körét. Ellenkező esetben eldobja a börtön kártyát és kimarad ebből a körből. Ha a játékos előtt egyszerre van börtön és dinamit, a dinamitra húz először. A Sheriff nem célozható vele.
* **Fegyver – Megnöveli azt a távolságot, ahol Bang!-et játszhat ki a játékos.**
  + **Gyorstüzelő – Ha aktív, bármennyi Bang! kártyát kijátszhat a játékos az adott körben.**

### Karakterek

A játékban minden karakter egyedi képességgel rendelkezik. A játék kezdetén a játékos kiválaszthat egy karaktert (vagy választhatja a véletlenszerű kiosztást), és a botokhoz is véletlenszerű karakterek kerülnek. Karakterek és képességeik leolvashatóak az 1. táblázatról:

|  |  |
| --- | --- |
| **Bart Cassidy** | **Minden elvesztett életpont után húz egy lapot a pakli tetejéről.** |
| **Black Jack** | **A köre elején, húzáskor megmutatja a második lapot, ha az kőr vagy káró, húz még egyet.** |
| **Calamity Janet** | **Bang! lapokat Nem talált! lapként is kijátszhat és fordítva. Támadó szándékkal csak egyet használhat körönként ugyanúgy.** |
| **El Gringo** | **Minden sebződés után húz egy lapot, ha van, az őt megsebző játékos kezéből.** |
| **Jesse Jones** | **A köre elején választhatja, hogy az első lapot egy másik játékos kezéből húzza a pakli teteje helyett.** |
| **Jourdonnais** | **Van egy beépített Hordó-ja. Kijátszhat még egy hordót, mint a többi játékos és ilyenkor kétszer húzhat, ha meglövik egy Bang!-gel.** |
| **Kit Carlson** | **A köre elején megnézi pakli felső 3 lapját, választ kettőt és a maradékot visszateszi.** |
| **Lucky Duke** | **Ha Hordó, Dinamit vagy Börtön miatt húznia kell, akkor kettőt húz, kiválaszt egyet, majd mindkettőt eldobja.** |
| **Paul Regret** | **A többi játékos eggyel messzebbről látja, kijátszhat Musztángot is, ekkor kettővel messzebbről látják.** |
| **Pedro Ramirez** | **A köre elején az első lapot húzhatja a dobó pakli tetejéről.** |
| **Rose Doolan** | **A többi játékost eggyel közelebbről látja. Kijátszhat Távcsövet is, ekkor kettővel közelebbről látja a többi játékost.** |
| **Sid Ketchum** | **Ha eldob két lapot, akkor visszanyer egy életpontot, akár egy körön belül többször is.** |
| **Slab the Killer** | **A Bang! kártyái elől csak kettő Nem talált! lappal térhet ki a célpont. A Hordó ugyan úgy le tudja védeni a lövést.** |
| **Suzy Lafayette** | **Ha üres a keze, húz egy lapot.** |
| **Vulture Sam** | **Ha kiesik valaki, megkapja a kiesett játékos lapjait.** |
| **Willy the Kid** | **Egy körben bármennyi Bang! kártyát kijátszhat.** |

1. táblázat A különböző karakterek és speciális képességeik

Minden karakterképesség más és más, így érdemes ezek köré építeni a taktikát.

### Játékosok közötti távolság

A játékosok közötti távolságot úgy érdemes elképzelni, mintha egy körben ülnének egymás mellett. Adott játékos melletti játékosok egy távolságra vannak, azok szomszédjai, pedig kettőre és így tovább. Mivel két játékos között általában két különböző távolság van, egy az óramutató járásával megegyezően, másik az óramutató járásával ellentétesen. Ilyenkor a kisebb távolságot kell figyelembe venni. Ez a távolság vonatkozik a Pánik kártyára, amit olyan célpontot használhatunk, aki egy távolságon belül van. Ezt a távolságot módosítja a Musztáng és a Távcső kártya. Ha van aktív Musztáng lapja a játékosnak, akkor őt eggyel távolabbról látják. Ha van aktív Távcső kártyája, akkor a játékos a többieket eggyel közelebbről látja. Tehát ő és a szomszédjai között 0 a távolság és így tovább. A fegyverek csak a Bang! kártya hatótávolságát befolyásolják. Tehát például: Ha van egy Remingtonunk, ami három távolságot ad, akkor azokra az ellenfelekre lehet lőni, akik három távolságon belül vannak.

### Speciális szabályok

* A játékban nincs emberi játékostárs – mindenki más bot.
* Csak a Sheriff kiléte köztudott.
* A botok mesterséges intelligenciája egyszerű szabályok szerint működik, és elsősorban a kártyák típusára és lehetőségeikre reagálnak.

## A Bang! letöltése, telepítése

A Bang! játék futtatásához Java Development Kit (JDK) 21-es verzióra van szükség. A program Java 21-ben készült, így a megfelelő működés érdekében ajánlott, hogy ez a verzió legyen telepítve a felhasználó gépén.

### Telepített Java verzió ellenőrzése

Ellenőrizhetjük, hogy milyen Java-verzió van telepítve, ha megnyitunk egy parancssort vagy terminált, és beírjuk az alábbi parancsot: *java -version*

Ha a megjelenő verzió kisebb, mint 21, vagy a parancs nem ismert, akkor telepíteni kell a megfelelő verziót.

### Java JDK 21 telepítése

A JDK 21 telepítő elérhető az Oracle hivatalos weboldalán[[2]](#footnote-3):

<https://www.oracle.com/java/technologies/javase/jdk21-archive-downloads.html>

Töltsük le az operációs rendszerünknek megfelelő verziót, és kövessük a telepítő utasításait. Telepítés után érdemes újraindítani a számítógépet, hogy a Java parancsok elérhetővé váljanak a rendszerben.

### A játék letöltése és indítása

A projekt elérhető a GitHubon[[3]](#footnote-4): https://github.com/ftenge/bang

Letölthetjük ZIP fájlban, vagy klónozhatjuk Git segítségével:

*git clone* [*https://github.com/ftenge/bang.git*](https://github.com/ftenge/bang.git)

Majd, nyissunk egy parancssort, és navigáljunk a src mappába, ahol a Main.java található.

### A program fordítása és futtatása

A játék elindítása parancssorból a következő lépésekkel történik:

javac Main.java

java Main

Ezekkel a parancsokkal először lefordítjuk a Main.java fájlt, majd elindítjuk a játékot.

A játék futtatásához nincs szükség külön fejlesztői környezetre (pl. IntelliJ), elég egy működő JDK 21 és a forráskód. Az alkalmazás egy ablakot nyit meg, ahol a Bang! játék grafikus felülete válik elérhetővé.

## Hardver- és szoftverigény

Az alkalmazás célja, hogy egy modern, könnyűsúlyú, egyjátékos asztali játékélményt nyújtson. A játék nem igényel jelentős számítási kapacitást vagy speciális eszközöket, így a legtöbb, Java-t futtatni képes számítógépen zökkenőmentesen használható.

### Ajánlott hardverkonfiguráció

* **Processzor**: 2 GHz-es, négymagos processzor vagy gyorsabb
* **Memória (RAM)**: 4 GB vagy több
* **Tárhely**: ~180 MB szabad lemezterület
* **Képernyőfelbontás**: 1920×1080 vagy magasabb, a kényelmes megjelenítéshez
* **Egér**: szükséges a játék felületén való navigáláshoz

### Szoftverkövetelmények

* **Operációs rendszer**: Windows 10
* **Java futtatókörnyezet (JRE)**: Java SE 21 vagy újabb verzió

### Egyéb megjegyzések

* Az alkalmazás nem igényel internetkapcsolatot a használathoz.
* Nem szükséges adminisztrátori jogosultság a futtatásához.
* Mivel a játék csak egyjátékos módban működik, nincs hálózati vagy multiplayer komponens, így különösen kis erőforrásigényű.
* Mivel a lapok „logolt” neve és különböző tulajdonságai angolul vannak, ezért minimális angol tudás szükséges.

## Játékmenet

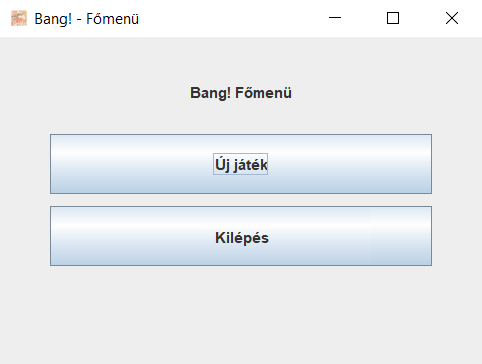
### A játék áttekintése

A játék egy digitális adaptációja a klasszikus Bang! társasjátéknak, kifejezetten egyjátékos módra fejlesztve. A játékos a Sheriff szerepét veszi fel, és célja, hogy a mesterséges intelligencia által irányított ellenfeleket legyőzze a játék szabályainak megfelelően. A játék teljes mértékben egérrel vezérelhető, és a felhasználói felület intuitívan vezeti végig a játékost a körökön. A játék körökre oszlik, és minden körben a játékos vagy a mesterséges intelligencia által vezérelt ellenfelek lépnek. Minden játékos körében kártyákat húz, kijátszik, támad vagy védekezik, és végül eldobja a felesleges kártyákat.

### Új játék indítása

A program elindításakor az 1. ábrán látható főmenü jelenik meg, ahol kettő lehetőség közül lehet választani:

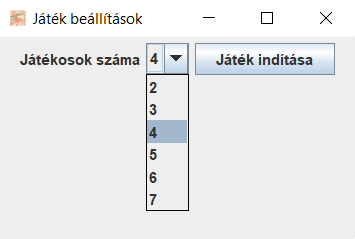
* **Új játék indítása**
* **Kilépés**



1. ábra A játék indítása

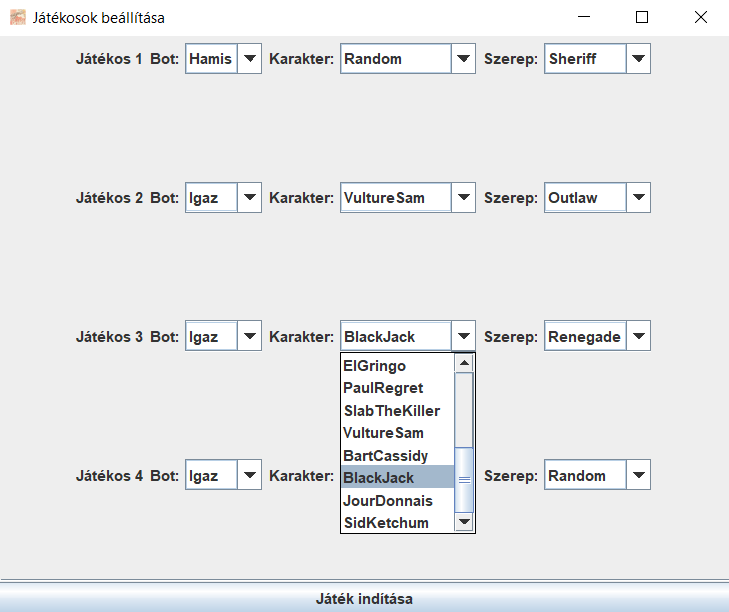
Az *Új játék* gomb megnyomásával megnyílik a 2. ábrán látható a *Játék beállítások* ablak, ahol a játékos kiválasztja, hogy mennyi játékos legyen (2-7).

2. ábra A játékosok számának kiválasztása



A *Játék indítása* gomb megnyomásával megjelenik 3. ábrán látható *a Játékos beállítása* ablak. Ebben a felhasználó kiválaszthatja:

* melyik játékos szeretne lenni
* a saját, és a botok karakterét (vagy választhat véletlenszerű karaktert)
* a saját, és a botok szerepeit (szintén lehetőség van véletlenszerű beállításokra)



3. ábra Botok, karakterek, és szerepek beállítása

Az ellenfelek minden esetben mesterséges intelligencia által vezérelt botok. A konfiguráció beállítása után a *Játék indítása* gombbal indítható el a játék.

### A játéktér felépítése

A játék elindulása után megjelenik a fő játéktér, amely több vizuális komponensből épül fel. A felület kialakítása Swing alapon történt, és minden elem világosan elkülönül, hogy segítse a játékos tájékozódását. A főbb elemek a következők:

#### **Játékos lapjai (alsó sáv)**

Ez a szekció a játékos kézben tartott lapjait jeleníti meg: mindegyik kártya egyedi képe jelenik meg, ha a játékos föle viszi az egeret, akkor megnő, és kijelölhető.

#### **Ellenfelek (felső sáv)**

Az ellenfelek, azaz a botok, külön panelen helyezkednek el. Mindegyik ellenfél esetén látható:

* A nevük, aktuális, és maximális HP értékük.
* Egy kártya borítóval felfelé, benne egy fehér kör és benne a kézben tartott kártyáinak száma.
* Az aktív, asztalra lerakott kártyáik képpel felfele (felszerelések, fegyverek).

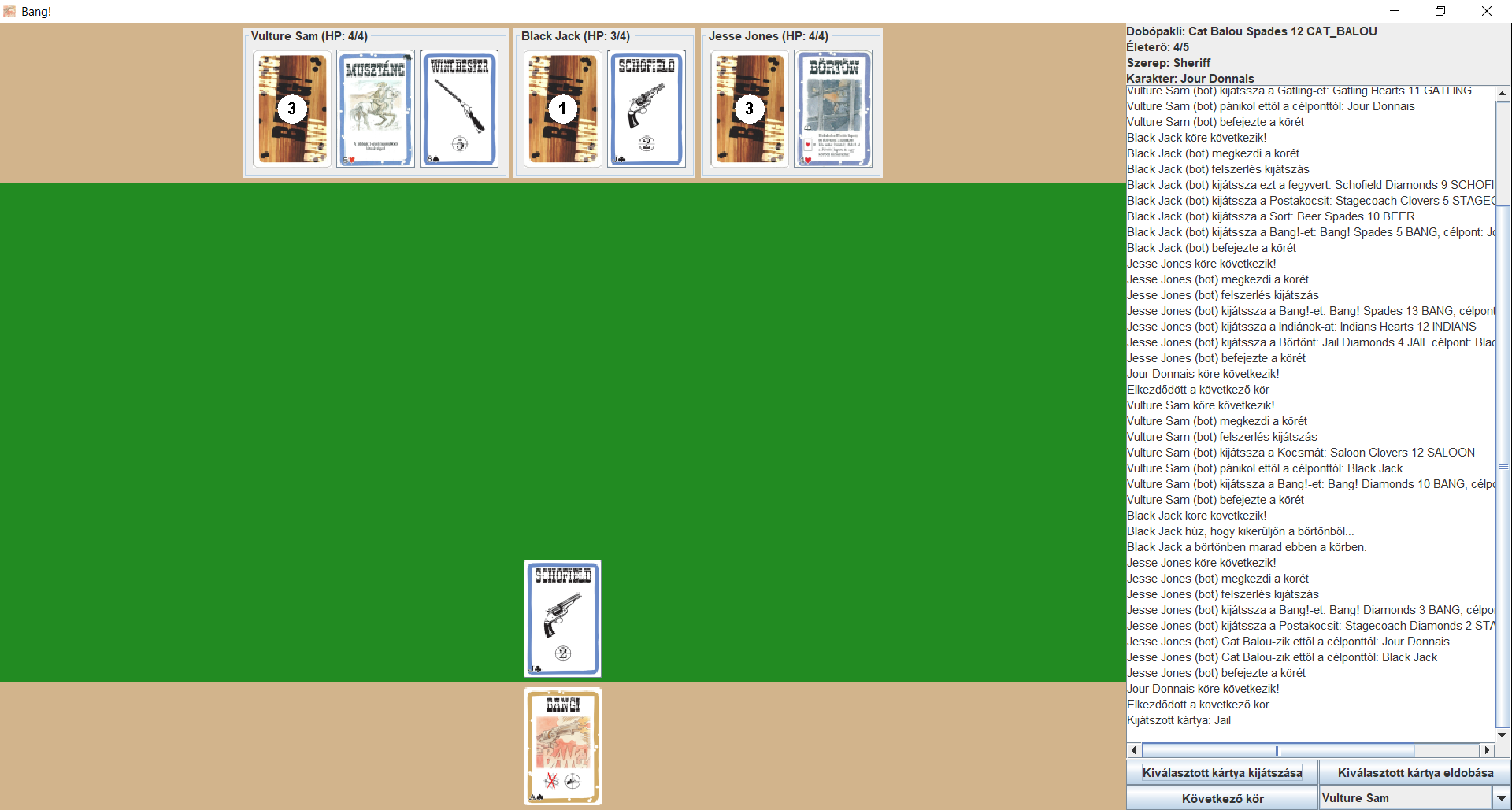
#### **Középső vezérlőpanel**

Ez a szekció a játékos asztalon lévő kijátszott, aktív lapjait jeleníti meg: mindegyik kártya egyedi képe jelenik meg, nem lehet velük interakcióba lépni közvetlenül, csupán azt az információt adják át, hogy a kártya aktív és befolyásolj a játékos képességeit.

#### **Jobb oldali log panel**

* **Eldobott lapok**: a Dobópakli jelzéssel jelenik meg az utolsó eldobott lap neve.
* **Aktuális/maximális életerő: a játékos aktuális és maximális életereje.**
* **Szerep: a játékos játékbeli szerepe.**
* **Karakter: a játékos karakterének a neve.**
* **Játéknapló**: itt látható az összes fontos esemény (pl. ki támadott kit, ki dobott lapot).
* **Kiválasztott kártya gomb: ezzel tudjuk kijátszani az aktuálisan kijelölt kártyát.**
* **Kiválasztottk ártya eldobása gomb: ezzel tudjuk eldobni az aktuálisan kijelölt kártyát.**
* **Következő kör** gomb: ezzel adható át a kör a következő játékosnak.
* **Célpontválasztó legördülő menü**: amelyben kijelölhető, hogy melyik ellenfelet kívánjuk megtámadni.

A felület célja, hogy világos, egyszerűen értelmezhető legyen, és megkönnyítse a játékos számára a döntéshozatalt. Ezt a felületet ábrázolja a 4. ábra.



4. ábra A játék futás közben

### A körök működése és fázisai

A játék körökre osztva zajlik. Minden kör egy játékos (jelenleg mindig az emberi játékos) teljes körét jelenti, amely során végrehajthat különböző műveleteket. A botok automatikusan, a megírt mesterséges intelligencia logikája alapján hajtanak végre akciókat a saját köreikben. Ezek a fázisok automatikusan követik egymást a játék során. Az emberi játékos minden akcióját vizuálisan is támogatja az alkalmazás, míg a botok automatikusan és minimális késleltetéssel lépnek a saját logikájuk alapján.

Egy kör három fő fázisból áll:

#### Kártyahúzási fázis

A játékos a kör elején automatikusan húz két lapot a húzópakliból. Ha a húzópakli kifogy, a dobópakli újra megkeverésre kerül és húzópaklivá válik, a legutolsó eldobott lap nem keveredik bele a húzópakliba, ott marad a dobópakli tetején.

#### Akciófázis

Ez a fázis a leghosszabb, itt történik a játék fő része. A játékos a kezében tartott kártyákat különböző célokra használhatja:

* **Felszerelés kijátszása**:

A játékos kijátszhat olyan kártyákat, mint a Hordó, Musztáng, Távcső vagy fegyverek. Ezek az asztalra kerülnek és onnantól aktívak.

* **Támadás**:

A Bang! típusú kártyával a játékos egy ellenfelet támadhat meg, ha az lőtávolságban van. A célpont kiválasztása a célpontválasztó segítségével történik.  
Egy körben alapesetben csak egy Bang! kártya játszható ki, kivéve, ha karakterképesség vagy más kártya ezt felülírja.

* **Különleges kártyák**:

Bizonyos kártyák minden ellenfelet érintenek („Gatling”, „Indiánok”), vagy más játékos lapját célozzák („Pánik”, „Cat Balou”). A játék automatikusan értelmezi ezek hatását.

* **Segéd kártyák**:

Néhány kártya csak az aktuális játékosra van hatással, például gyógyítja („Sör”), vagy új lapokat huzat a játékossal („Postakocsi”, „Wells Fargo”).

* **Nem talált!** kártya:

Nem akcióként, hanem reakcióként használható[[4]](#footnote-5). Amikor egy ellenfél megtámad minket, ezzel hárítható a támadás.

#### Dobás fázis

A kör végén a játékosnak el kell dobnia annyi lapot, hogy a kezében lévő kártyák száma ne haladja meg az aktuális életerőpontjainak számát. A felesleges kártyák a dobópakli tetejére kerülnek.

### Győzelmi feltételek és a játék vége

A játékos és a botok közötti küzdelem célja az egyéni **győzelmi feltétel** teljesítése, amelyet a játékos szerepköre határoz meg[[5]](#footnote-6). A játék indulásakor minden résztvevő véletlenszerűen (vagy felhasználói választással) kap egy szerepet. A szerepek titkosak, kivéve a Sheriff szerepét, amely mindig ismert. A játék vége automatikusan bekövetkezik, amikor valamelyik győzelmi feltétel teljesül. A játék végi képernyő kiírja a győztesek szerepét, valamint lezárja az aktív játékmenetet.

# Fejlesztői dokumentáció

## A szoftver rövid leírása, specifikáció

A szoftver egy Java nyelven készült, asztali (desktop) környezetben futtatható egyjátékos kártyajáték, amely a Bang! nevű társasjáték digitális adaptációja. A játék célja a klasszikus szerepalapú játékmenet és a kártyahasználat élményének átültetése egy számítógépes környezetbe, a lehető leghitelesebb módon, miközben a felhasználó botok ellen játszik. A projekt célja egy oktatási célú demonstrációs alkalmazás megvalósítása, amely bemutatja a Java objektumorientált programozási eszköztárát, a Swing[2] felhasználói felületépítést, valamint az egyszerű AI logika integrálását.

Főbb funkciók:

* Egyjátékos mód botok ellen.
* Véletlenszerű vagy felhasználó által választott karakterek és szerepek.
* Kártyák kezelése: kézben tartott, asztalra helyezett, és eldobott kártyák.
* Játékosinterakciók: támadás, védekezés, különleges akciók.
* Botlogika: minden bot szerepkörtől és kártyáktól függően dönt.
* Felhasználói felület: grafikus ablakos megjelenítés Java Swing segítségével.

### A szoftver működésének rövid összefoglalása:

A játék elindítása után a felhasználó a főmenüből új játékot indíthat, ahol kiválaszthatja a játékosok számát (önmagán kívül csak botokat), valamint beállíthatja a karaktereket és szerepeket (vagy véletlenszerűen oszthatja ki). Ezután a játékosok egymás után hajtanak végre köröket, kártyákat húznak, kijátszanak, támadnak, védekeznek. A játék automatikusan véget ér, ha valaki eléri a saját szerepkörének győzelmi feltételét.

A szoftver fejlesztése során elsődleges szempont volt a modularitás, a karbantarthatóság, valamint az objektumorientált szemlélet, különösen a karakterek, kártyák és botok kezelésében.

## Követelményelemzés

### Funkcionális követelmények

A funkcionális követelmények leírják azokat az alapvető műveleteket és képességeket, amelyeket a szoftvernek biztosítania kell a Bang! játék megfelelő implementálásához. A felhasználónak lehetősége kell legyen új játékot indítani a főmenüből, ahol kiválaszthatja a karaktereket és a szerepeket, vagy választhatja a véletlenszerű kiosztást is. A játék támogatja az egyjátékos módot, amelyben a felhasználó gépi ellenfelek (botok) ellen játszik. Ezek a botok önállóan, saját belső logikájuk alapján hajtják végre lépéseiket minden körben, beleértve a kártyahúzást, felszerelések kijátszását, támadást, védekezést és egyéb szükséges műveleteket.

Kulcsfontosságú a kártyakezelés megvalósítása: a játékosoknak képesnek kell lenniük kártyát húzni a pakliból a körük elején, kijátszani különböző típusú lapokat (mint a Bang!, Nem talált!, Sör stb.), felszereléskártyákat (fegyverek, védelmi tárgyak) az asztalukra helyezni, és szükség esetén lapokat eldobni a kezükből. A rendszernek automatikusan kell kezelnie a játék végállapotát, és meg kell állapítania a győztest, amint valamelyik szerepkör teljesíti a győzelmi feltételeit. Mindezekhez elengedhetetlen egy grafikus felhasználói felület biztosítása, amely Swing technológiával készül. Ezen a felületen a játékosnak tisztán kell látnia a saját kezében lévő kártyákat, az asztalon lévő lapokat (mind a sajátját, mind az ellenfelekét), valamint az ellenfelek életerejét. A felületnek lehetővé kell tennie a kártyák egyszerű, kattintással történő kiválasztását.

### Nem funkcionális követelmények

A nem funkcionális követelmények a rendszer működésének minőségi jellemzőire vonatkoznak. A telepíthetőségnek egyszerűnek kell lennie; a projekt futtatásához elegendő egy Java fejlesztői környezet (mint az IntelliJ IDEA) vagy a program parancssorból történő indítása a Main osztályon keresztül. Ehhez szükséges egy Java futtatókörnyezet (JRE 21 vagy újabb verzió) A Swing alapú grafikus felületnek felhasználóbarátnak kell lennie, átlátható és logikus elrendezéssel, hogy a játékosok könnyen kezelhessék a játékot, még több ellenfél esetén is. Fontos szempont a stabilitás és hibatűrés, vagyis a szoftvernek képesnek kell lennie kezelni a váratlan helyzeteket és hibákat, mint például egy üres húzópakli vagy egy olyan szituáció, ahol nincs érvényes célpont. Végül, az objektumorientált tervezésnek köszönhetően a rendszernek bővíthetőnek kell lennie, hogy a jövőben könnyen hozzá lehessen adni új kártyákat, karaktereket vagy akár szabályvariánsokat a meglévő struktúra jelentős átalakítása nélkül.

## Tervezés

### A szoftver architektúrája

A Bang! játék megvalósítása egy háromrétegű architektúra szerint történt:

* **Prezentációs réteg (UI):**

Swing alapú grafikus felhasználói felület (*BangGameUI*, *MainMenuUI* stb.), amely felelős a felhasználói interakciók kezeléséért.

* **Üzleti logika réteg (BL):**

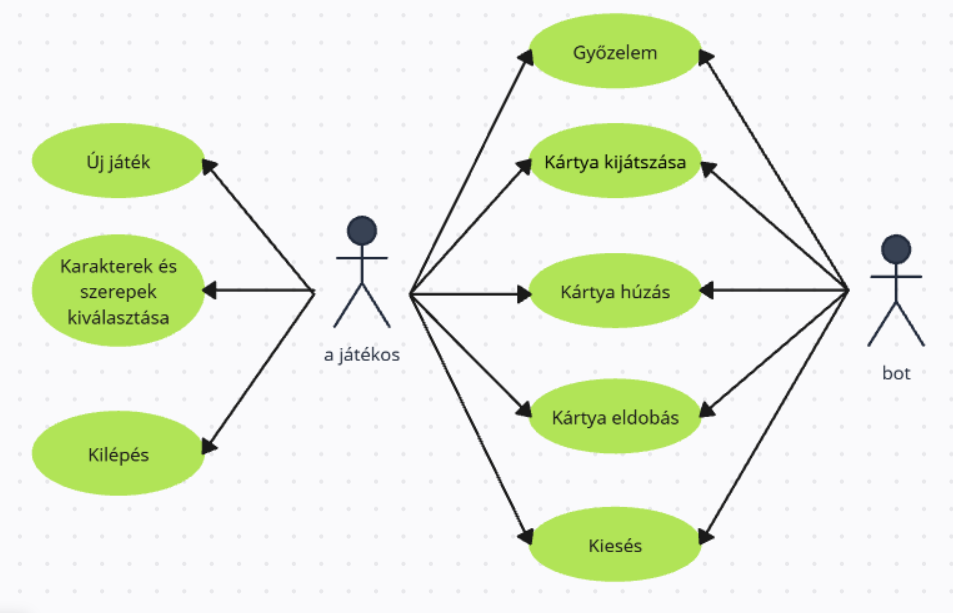
A játékmenet szabályait, a körök kezelését, győzelmi feltételek érvényesítését valósítja meg (*GameLogic*, *Bot*).

* **Adatkezelési réteg (Model):**

A játék modelljeit tartalmazza, mint például a játékosokat (*BaseModel* és leszármazottai), a kártyákat (*Card* és leszármazottai), a paklikat és az állapotokat.

Ez a rétegzett szerkezet biztosítja a fejlesztés átláthatóságát, az egyes rétegek független módosíthatóságát és a könnyebb karbantartást.

### Felhasználói eset diagram



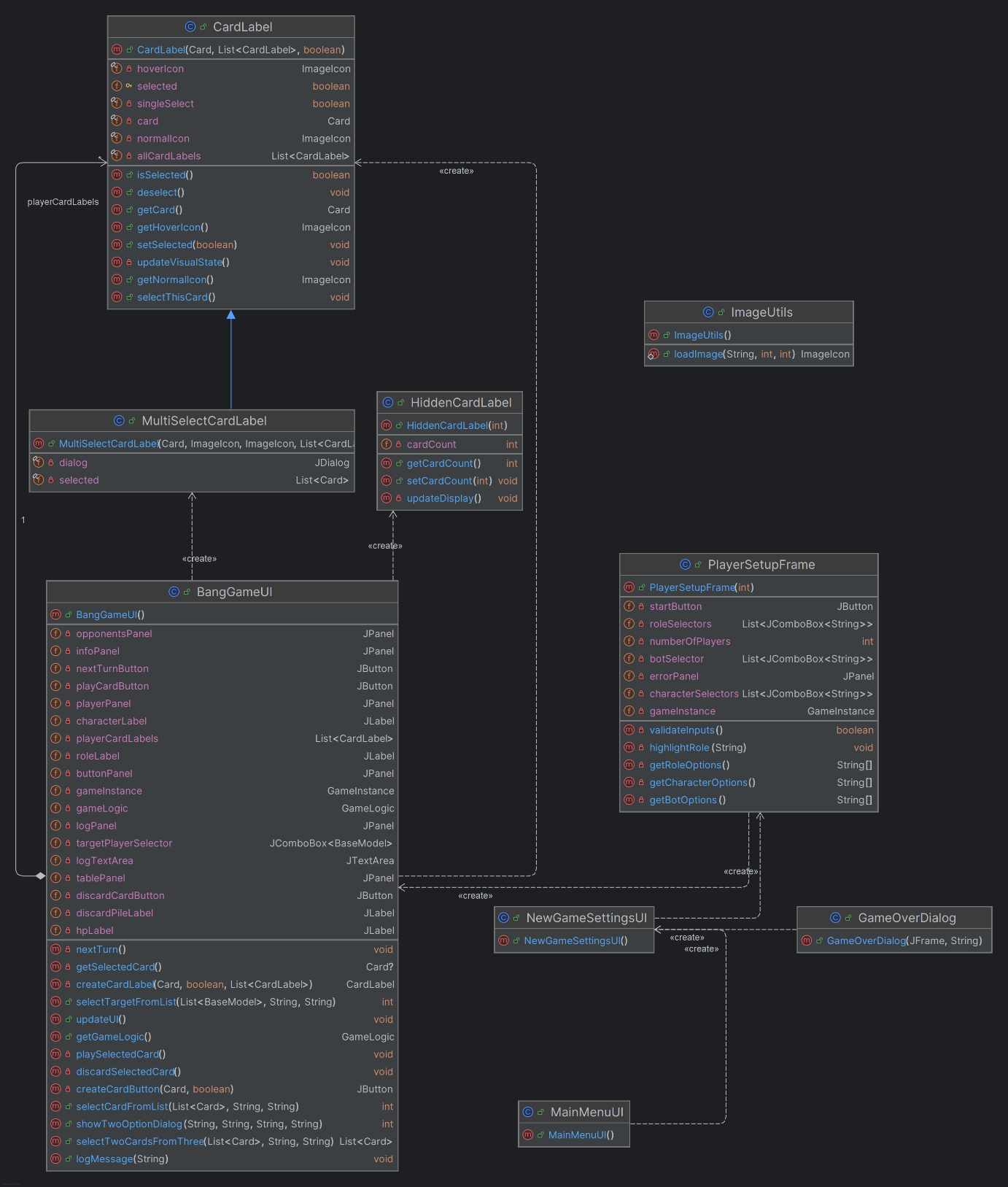
5. ábra Felhasználói eset diagram

Mint az 5. ábrán is látható, a Bang! játék egyjátékos változatában kétféle szereplő vesz részt: maga a játékos, akit a felhasználó irányít, és a bot, amely a gép által vezérelt ellenfél. Bár az irányításuk különbözik, a játék során nagyon hasonló tevékenységeket végeznek.

A játékos azzal kezdi, hogy új játékot indít, majd kiválaszt egy karaktert, amely meghatározza az életerejét és speciális képességeit. A szerepek kiosztása automatikusan történik, így nemcsak a karakter, hanem a győzelmi feltételek is egyértelműek lesznek. Miután a játék elindult, a játékos minden kör elején két kártyát húz a pakliból. Ezután dönthet arról, hogy milyen kártyákat játszik ki: támadhat, védekezhet, gyógyulhat, vagy speciális hatásokat idézhet elő. A kör végén figyelnie kell arra, hogy a kezében maradt kártyák száma ne legyen több, mint a megmaradt életereje — ha mégis több lenne, el kell dobnia a felesleges lapokat. A cél az, hogy teljesítse a szerepéhez tartozó győzelmi feltételeket, és ha sikerrel jár, megnyeri a játékot. Viszont ha az életereje nullára csökken, kiesik a játékból.

A bot, bár nem valódi játékos, szinte ugyanígy működik. A köre elején ő is húz két lapot, majd egy előre megírt logika szerint dönt arról, hogy melyik kártyát használja fel. A döntéseit egyszerű szabályok vezérlik: például ha van támadókártyája, és van támadható ellenfél, akkor azt játssza ki. A kör végén neki is el kell dobnia a felesleges kártyákat. A bot is megnyerheti a játékot a saját szerepének teljesítésével, vagy kieshet, ha az életereje elfogy.

### Osztálydiagramok



6. ábra ui csomag osztálydiagramja

A 6. ábrán látható osztálydiagram a Bang! játék implementációjának felhasználói felületét (UI) megvalósító Java Swing komponensek szerkezetét és kapcsolatait mutatja be. A diagram szemlélteti, hogyan épül fel a grafikus interfész az alkalmazás indításától kezdve a játékbeállításokon át egészen a tényleges játékmenet megjelenítéséig. Az itt látható osztályok mind a felhasználói interakciókért és a vizuális megjelenítésért felelősek.

A felhasználói interakció folyamata a *MainMenuUI* osztállyal kezdődik, amely a játék főmenüjét jeleníti meg, és innen indítható új játék. Új játék indításakor a *MainMenuUI* meghívja a *NewGameSettingUI* komponenst, amelynek feladata bekérni a játékosok számát. Ezt követően a *NewGameSettingUI* továbbítja a vezérlést a *PlayerSetupFrame* osztálynak. Ebben a keretben történik a játékosokhoz tartozó karakterek és szerepek részletes beállítása, lehetővé téve a játék testreszabását a kezdés előtt.

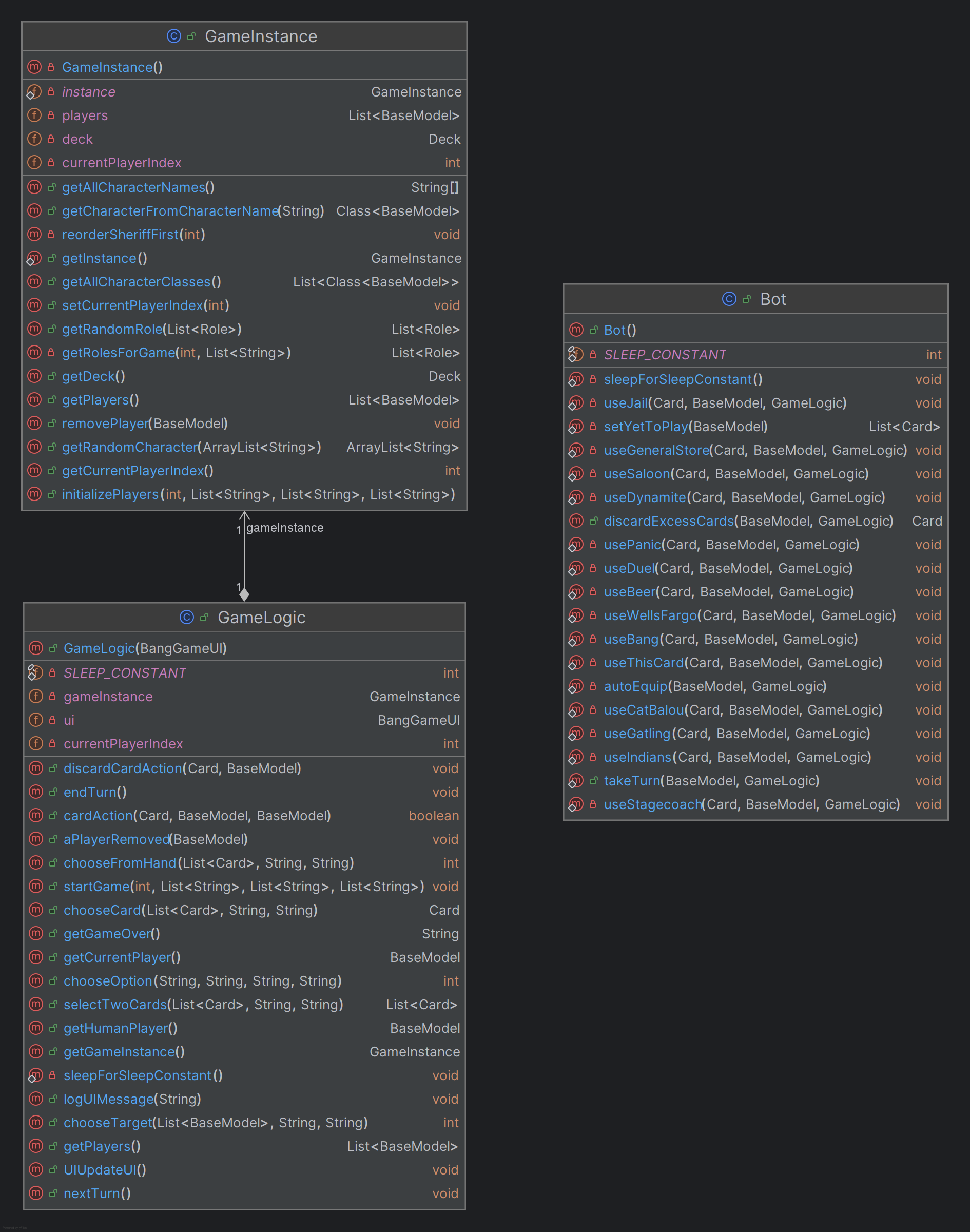
A beállítási fázis lezárultával aktiválódik a *BangGameUI* osztály, amely a felhasználói felület központi eleme és a játékmenet fő megjelenítője. Ez az osztály felelős a játék aktuális állapotának vizualizálásáért, beleértve a játékosok kezében lévő lapokat, az asztalon lévő felszereléseket, az ellenfelek adatait (pl. életerő, asztalon lévő lapok) és a játék közbeni eseményeket.

A *BangGameUI* működését több speciális segédosztály támogatja, amelyek szerves részét képezik a fő játékfelületnek. A *CardLabel* egy alapvető komponens, amelyet a *BangGameUI* kompozíció révén tartalmaz és kezel, felelőssége az egyes kártyalapok grafikus megjelenítése. Ebből az osztályból származik (öröklődés révén) a *MultiSelectCardLabel*, amely szintén a *BangGameUI* kompozíciójának része, és kibővíti az alap funkcionalitást azzal, hogy lehetővé teszi több kártya egyidejű kiválasztását, ami például lapok eldobásakor vagy bizonyos kártyaeffekteknél szükséges. Továbbá a *BangGameUI* kompozícióként tartalmazza a *HiddenCardLabel* komponenst is, amely olyan kártyák megjelenítésében nyújt segítséget, amelyek tartalmát a játékos nem láthatja (például az ellenfelek rejtett lapjai). Az *ImageUtils* hasznossági osztály bevezetése központosítja a képkezelési feladatokat (betöltés, átméretezés stb.). Ezáltal elkerülhető a kód duplikáció a különböző UI komponensekben, amelyek képeket használnak, valamint egységes felületet biztosít a képekkel kapcsolatos műveletekhez, elősegítve a karbantarthatóságot.

A UI struktúra kialakítása során több tudatos tervezési döntés született a jobb átláthatóság, karbantarthatóság és felhasználói élmény érdekében.

A *BangGameUI* és a különböző Label típusok (*CardLabel*, *MultiSelectCardLabel*, *HiddenCardLabel*) közötti kompozíciós kapcsolat biztosítja, hogy ezek a vizuális elemek szorosan a fő játékfelülethez tartozzanak. A *BangGameUI* felelős ezen komponensek létrehozásáért és életciklusuk kezeléséért, ami egyértelműsíti az objektumok közötti hierarchiát és megkönnyíti az erőforrás-kezelést, mivel a "rész" objektumok a "teljes" objektummal együtt jönnek létre és szűnnek meg.

A játékbeállítási folyamat (*MainMenuUI* -> *NewGameSettingUI* -> *PlayerSetupFrame*) lépésekre bontása külön ablakokba/panelekbe tudatos döntés volt. Egyrészt felhasználóbarát, vezetett élményt nyújt a játékosnak a konfiguráció során. Másrészt megakadályozza, hogy a felhasználó egyszerre túl sok beállítási opcióval legyen terhelve. Harmadrészt pedig logikailag elkülöníti a játék előkészítésének feladatait a tényleges játékmenet megjelenítésétől, ami tisztább kódszerkezetet eredményez.



7. ábra bl csomag osztálydiagramja

A 7. ábrán látható osztálydiagramon a bl csomag egyik kulcsfontosságú eleme a *GameInstance* osztály. Ennek az osztálynak a fő feladata, hogy a játék aktuális állapotának központi tárolójaként funkcionáljon. Itt tárolódnak a játékmenet szempontjából létfontosságú adatok, mint például a játékban részt vevő játékosok listája (*players*), a kártyapakli (*deck*) és az aktuálisan soron következő játékos indexe (*currentPlayerIndex*). Fontos megjegyezni, hogy a *GameInstance* Singleton tervezési mintát valósít meg, biztosítva ezzel, hogy az alkalmazásban mindig csak egyetlen példánya létezzen, így garantálva a játékállapot konzisztenciáját és egységes elérhetőségét.

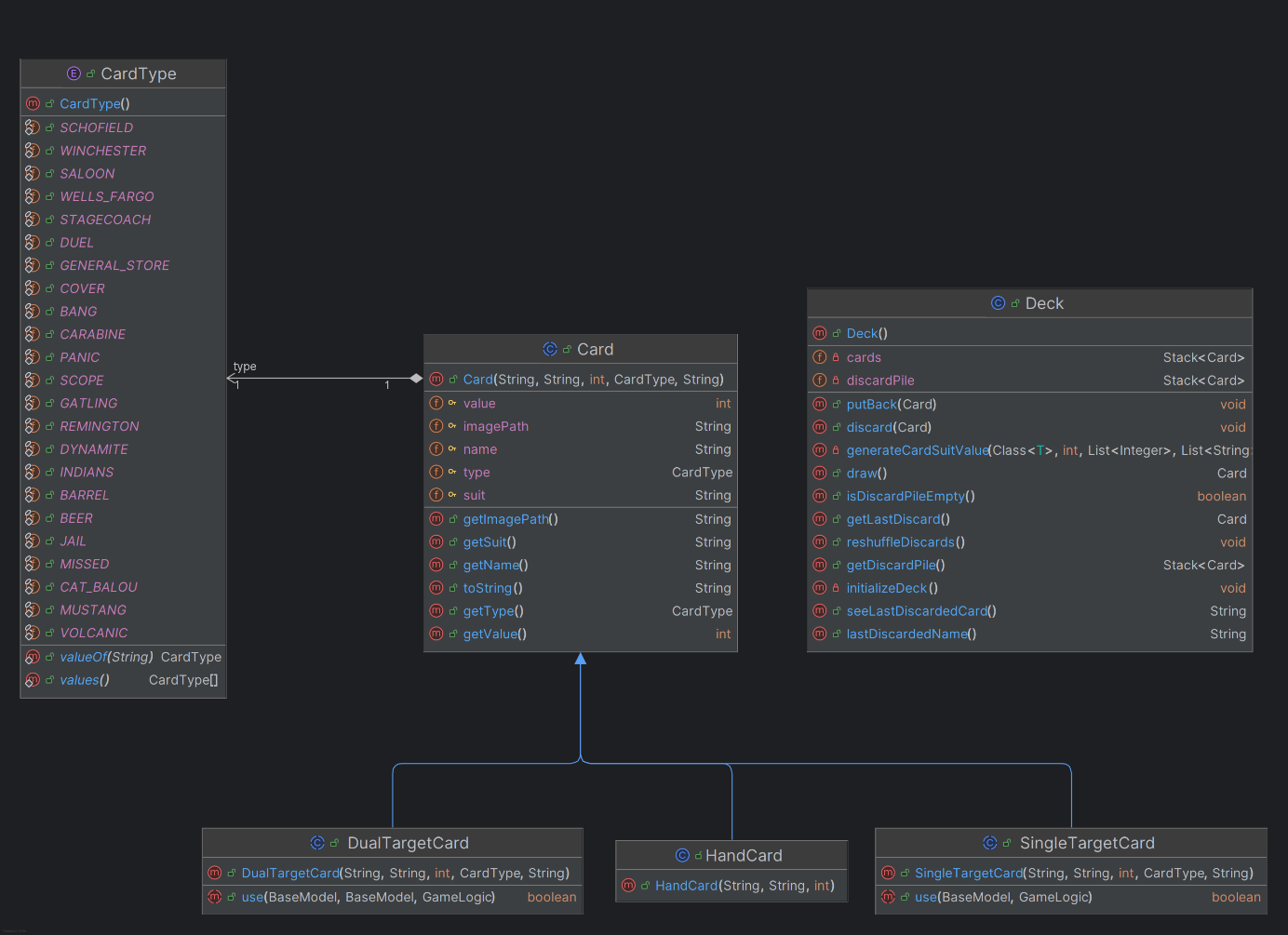
A csomag másik központi osztálya a *GameLogic*, amely az üzleti logika oroszlánrészét tartalmazza. Ez az osztály felelős a játékszabályok érvényesítéséért és a játékmenet dinamikájának irányításáért. Feladatai közé tartozik a játékosok akcióinak feldolgozása (pl. kártyahúzás, lap kijátszása), a körök léptetése, és a játék kimenetelének meghatározása (pl. győzelmi feltételek). A *GameLogic* szorosan együttműködik a *GameInstance* osztállyal az állapot lekérdezése és módosítása érdekében.

A harmadik osztály ebben a csomagban a *Bot*, amely a mesterséges intelligenciával vezérelt játékosok döntéshozatali logikáját foglalja magába. Amikor egy bot játékos kerül sorra, a *GameLogic* ennek az osztálynak a főmetódusait hívja meg, hogy meghatározza a bot lépéseit (milyen kártyát játsszon ki, kit célozzon meg stb.).

Az állapot (*GameInstance*) és a logika (*GameLogic*) szétválasztása egyértelműbbé teszi a rendszer felépítését. A *GameInstance* csak az adatokat tárolja, míg a *GameLogic* az ezeken az adatokon operáló szabályokat és folyamatokat tartalmazza. Ez a szétválasztás javítja a kód olvashatóságát, megkönnyíti a *GameLogic* tesztelését (akár különböző állapotokkal), és növeli a karbantarthatóságot.

A bot logikájának elkülönítése a *Bot* osztályba lehetővé teszi a mesterséges intelligencia viselkedésének egységbe zárását (enkapszuláció). Ez megkönnyíti a különböző nehézségű vagy stratégiájú botok implementálását (pl. további alosztályok létrehozásával vagy Strategy minta alkalmazásával) anélkül, hogy a *GameLogic* osztályt bonyolítanánk az AI specifikus részleteivel.

Az üzleti logika (bl) csomagba szervezés maga is egy tervezési döntés, amely a "Separation of Concerns" elvét követi, elkülönítve a játék mag logikáját a megjelenítési rétegtől (UI).

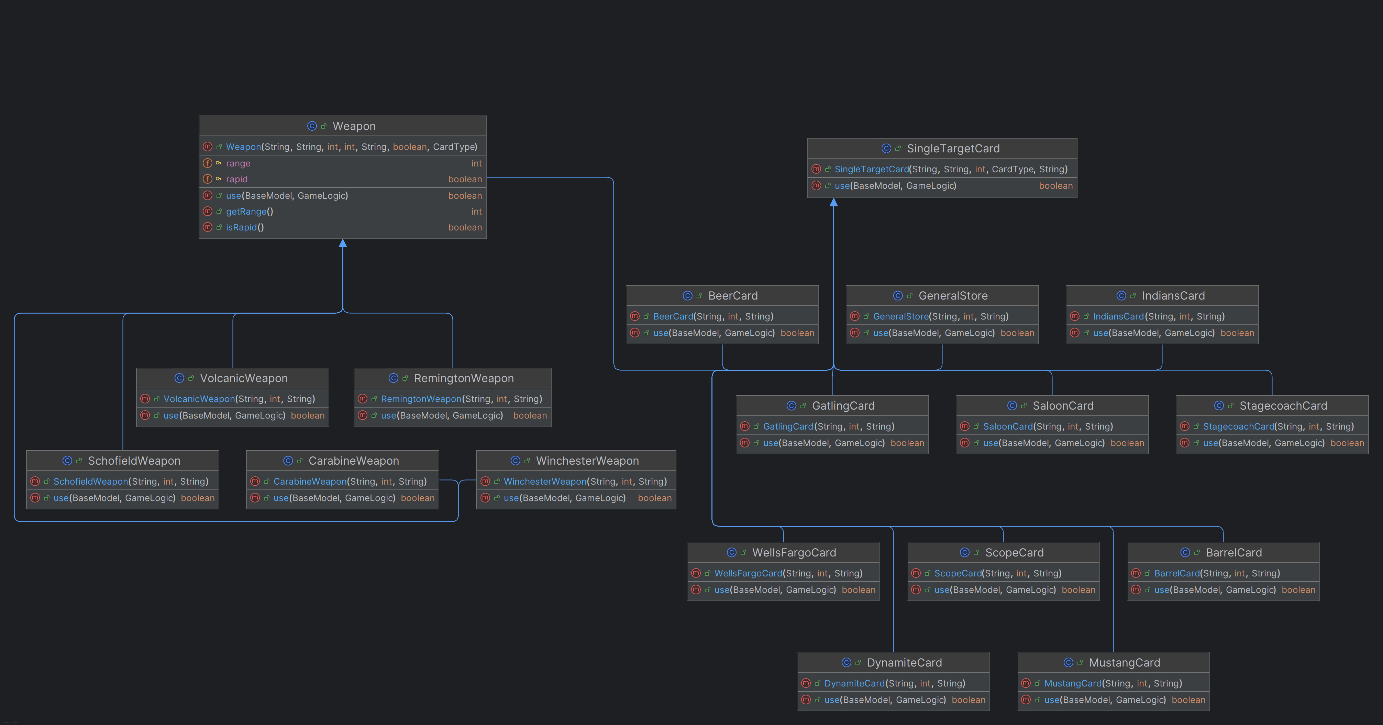


8. ábra model.card csomag osztálydiagramja

A 8. ábrán látható osztálydiagram a Bang! játék kártyáinak és a kártyapaklinak a modellezésére szolgáló osztályokat mutatja be, amelyek a model/cards csomagban helyezkednek el. Ez a struktúra definiálja a játékban használt kártyák alapvető tulajdonságait, típusait és a pakli kezelésének módját, alapot szolgáltatva ezzel a játék központi elemeinek.

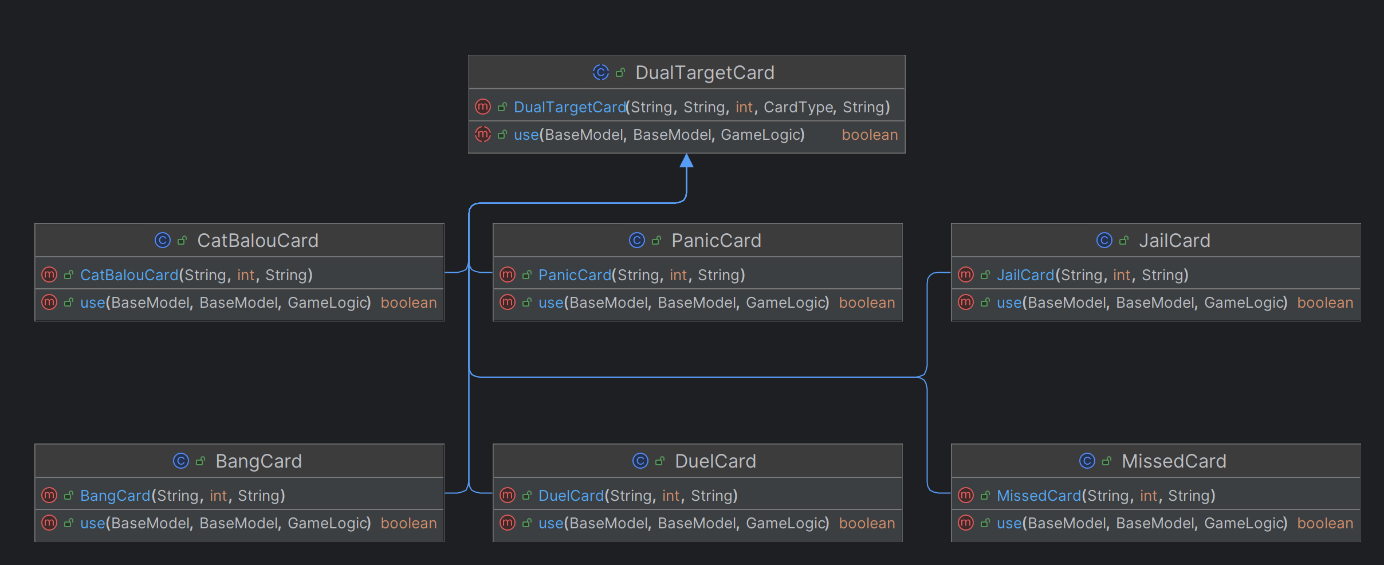
A diagram középpontjában a *Card* osztály áll, amely egy absztrakt ősosztályként funkcionál minden egyes kártyatípus számára. Ez az osztály tartalmazza a kártyák közös jellemzőit (például név, szín), és itt kerül felhasználásra a *CardType* enum is, amely a kártyák típusának (pl. BANG, MISSED, BEER) kategorizálására szolgál. Ez az enum segíti a kártyák egyértelmű azonosítását és a rájuk vonatkozó szabályok alkalmazását a játéklogikában. A csomag másik fontos eleme a *Deck* osztály, amely a húzó- és dobópaklit reprezentálja. Felelőssége a *Card* objektumok létrehozása és a húzó- és dobópakli tárolása, valamint ezekkel kapcsolatos műveletek biztosítása, mint a keverés és a lapok húzása.

A *Card* osztályból több specifikusabb kártyatípus is származik öröklődés révén, amelyek a kártyák eltérő viselkedését vagy célzási mechanizmusát modellezik. A *SingleTargetCard* olyan kártyákat képvisel, amelyek kijátszásához egyetlen célpont (a lapot kijátszó játékos) megadása szükséges. A *DualTargetCard* azokat a kártyákat modellezi, amelyek két célpontot (a lapot kijátszó játékos és egy valódi célpontot) igényelnek. A *HandCard* pedig azokat a kártyákat testesíti meg, amik betöltő szerepet töltenek be (pl. választani kell egy véletlenszerű kártyát az ellenfél kezéből). Ez az öröklődési hierarchia lehetővé teszi, hogy a közös kártyatulajdonságok az ősosztályban legyenek definiálva, míg a specifikus viselkedés vagy célzási logika a leszármazott osztályokban kapjon helyet. A *Deck* és a *Card* osztályok között aggregációskapcsolat áll fenn, mivel a *Deck* tartalmazza és kezeli a *Card* példányok egy csoportját. Az öröklődési hierarchia alkalmazása a *Card* osztályból kiindulva lehetővé teszi a polimorfizmus kihasználását (a különböző kártyatípusok egységesen *Card*-ként kezelhetők, miközben a saját specifikus *use()*. Emellett elősegíti a kód újrafelhasználását (közös attribútumok az ősosztályban) és a rendszer bővíthetőségét új kártyatípusok egyszerű hozzáadásával. A *CardType* Enum használata típustiszta és jól olvasható módot biztosít a kártyák kategorizálására, ellentétben például szöveges konstansok használatával. Ez megkönnyíti a kártyatípus-alapú logikai elágazások (pl. if elágazások) implementálását a *GameLogic*-ban. A *Deck* osztály elkülönítése egységbe zárja a paklival kapcsolatos összes műveletet (keverés, húzás, üres pakli kezelése). Ez a felelősségi körök tiszta szétválasztását eredményezi, mivel a pakli kezelésének logikája nem keveredik a kártyák egyedi tulajdonságaival vagy a játék általános menetével. A model/cards csomag létrehozása logikailag csoportosítja a kártyák és a pakli reprezentációjához kapcsolódó osztályokat, javítva ezzel a projekt általános struktúráját és átláthatóságát.



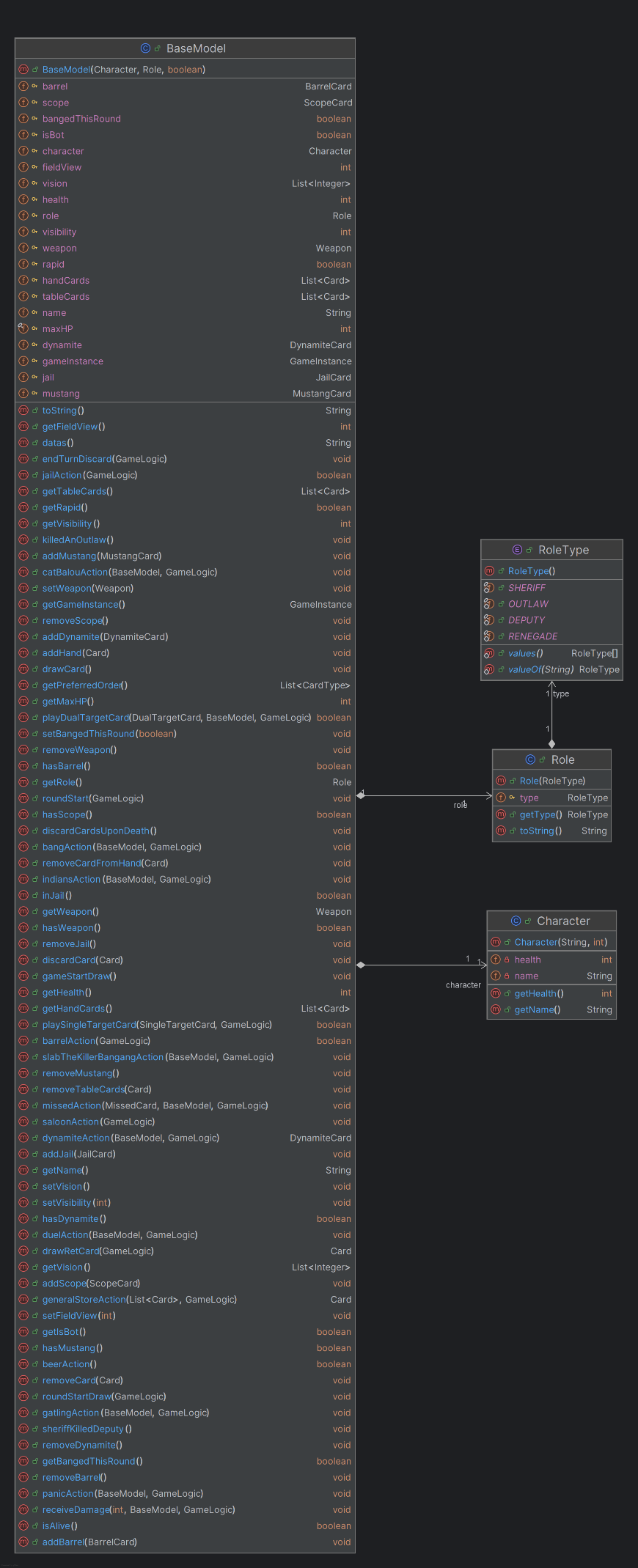
9. ábra SingleTargetCard osztály és leszármazottjainak osztálydiagramja

A 9. ábrán látható osztálydiagram a kártyamodell egy specifikus részletét mutatja be, kiemelve azokat a kártyatípusokat, amelyek a *SingleTargetCard* osztályból származnak, és amelyek kijátszásához vagy hatásának érvényesítéséhez egyetlen célpont szükséges. A diagramon látható, hogy a *Weapon* (Fegyver) osztály a *SingleTargetCard* egy specializációja. Ebből a Weapon osztályból további öt konkrét fegyverkártya-osztály öröklődik (pl. *Schofield*, *Volcanic*, stb.), amelyek a különböző lőtávolságú és képességű fegyvereket modellezik a játékban. A fegyvereken kívül további tizenegy különböző kártyaosztály (mint például a Sör vagy Indiánok!) is közvetlenül a *SingleTargetCard* osztályból származik. Ezek a kártyák testesítik meg azokat a különféle akciókat és hatásokat, amelyek egyetlen kiválasztott célpontra irányulnak vagy az összes játékosra.



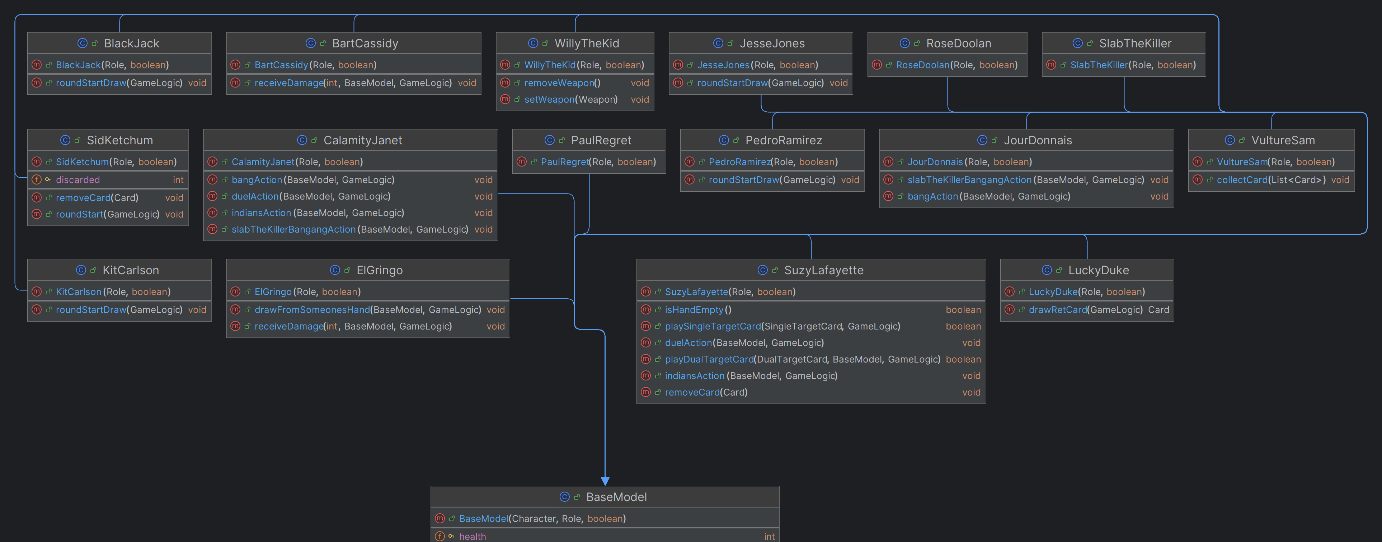
10. ábra DualTargetCard osztály és leszármazottjainak osztálydiagramja

A 10. ábrán látható osztálydiagram a kártyamodell azon részét ábrázolja, amely a *DualTargetCard* osztályból leszármazó kártyákat foglalja magában. Ezeket a kártyákat az jellemzi, hogy hatásuk érvényesítéséhez vagy kijátszásukhoz két célpont (ezek közül az egyik a lapot kijátszó játékos, másik a szó szerinti célpont) megjelölése szükséges. A diagram szerint hat különböző, konkrét kártyaosztály öröklődik közvetlenül a *DualTargetCard* ősosztályból. Mindegyik ilyen alosztály egyedi játékbeli akciót vagy hatást valósít meg, amelynek közös tulajdonsága a két célpontot igénylő mechanizmus. Az öröklődési struktúra itt is azt a célt szolgálja, hogy a két célpont kezelésével kapcsolatos közös logikát és adatokat a *DualTargetCard* bázisosztály tartalmazza.



11. ábra model.utilities csomag osztálydiagramja

A 11. ábrán látható osztálydiagram a model/utilities csomagban található alapvető osztályokat és típusokat mutatja be. Ezek az elemek kulcsfontosságúak a játékosok alapvető attribútumainak, mint a karakter és a szerepkör, egységes és strukturált reprezentációjához. A csomag központi eleme a *BaseModel* osztály, amely – ahogy a neve is sugallja – alapul szolgál a játék más, összetettebb entitásai számára, és kiemelt fontossággal bír a projekt egészében. A *BaseModel* egy alaposztály, amely az egyedi karakter osztályok ősosztályaként funkcionál. Elsődleges felelőssége, hogy közös struktúrát és alapvető funkcionalitást biztosítson a leszármazottak számára. A kompozíció alkalmazása a *Character* és *Role* objektumok *BaseModel*-en belüli tárolására (ahelyett, hogy például a karakter osztályok mindháromból öröklődne). Egy játékosnak van egy karaktere és van egy szerepköre. Ez a megközelítés rugalmasabb és tisztább, mint a többszörös öröklődés, és jobban elválasztja az egyes koncepciókat. A *Character* osztály segíti a játékos által megszemélyesített konkrét karakter ábrázolását. Ez az osztály tárolja a karakter nevét és maximális életerejét. A *Role* osztály a játékos szerepkörét képviseli, amely alapvetően meghatározza a játékos győzelmi feltételeit. A *Role* osztály a *RoleType* enumot használja a konkrét szerepkör típusának tárolására. A *RoleType* egy enumeráció (enum), amely a játékban előforduló lehetséges szerepkörök zárt halmazát definiálja (SHERIFF, DEPUTY, OUTLAW, RENEGADE). A *RoleType* Enum használata a szerepkörök definiálására növeli a típusbiztonságot, megakadályozza az elírásokat, és olvashatóbbá, könnyebben karbantarthatóvá teszi a kódot, összehasonlítva például sztringek vagy numerikus értékek használatával.

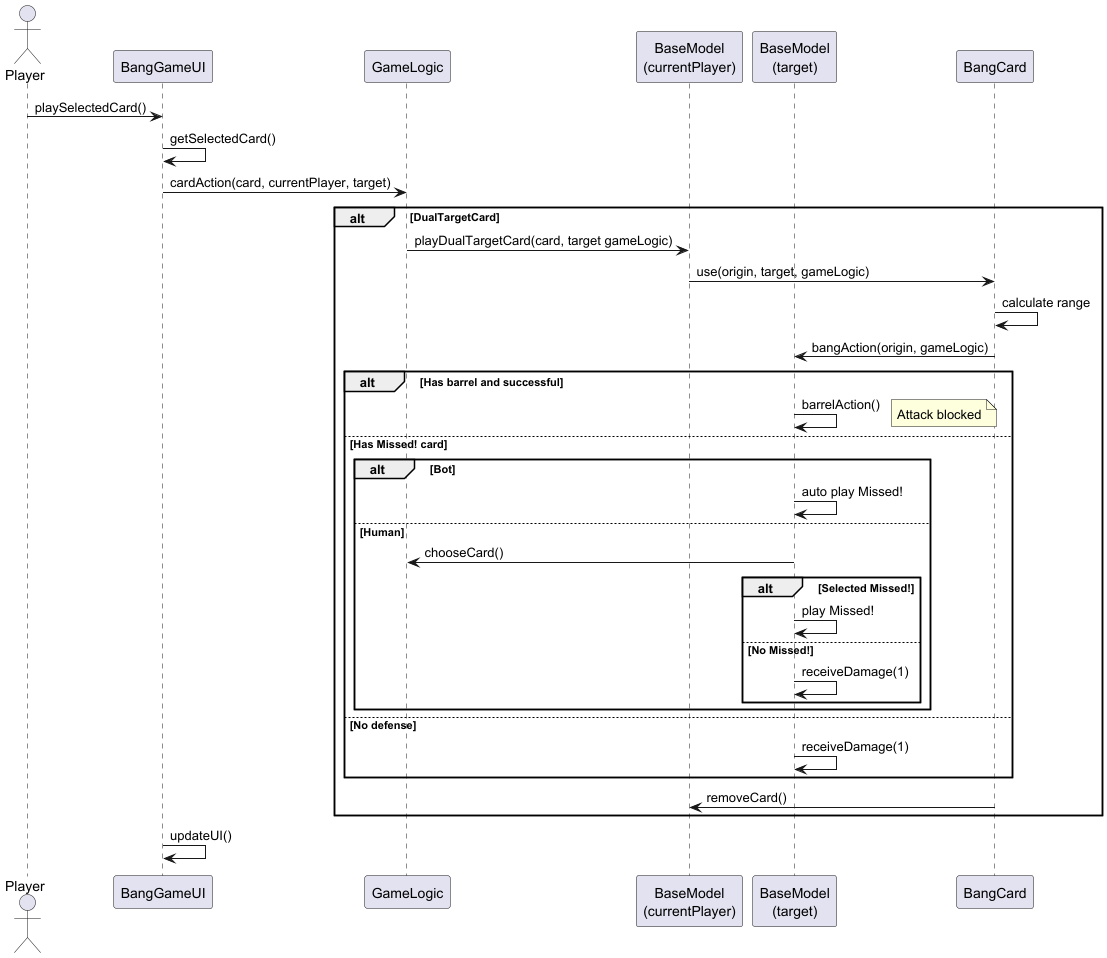


12. ábra BaseModel osztály leszármazottjainak az osztálydiagramja

A 12. ábrán látható osztálydiagram a *BaseModel* osztályt és annak közvetlen leszármazottait, a játékban szereplő 16 konkrét karaktert mutatja be. A *BaseModel* itt mint alapvető osztály jelenik meg, amely meghatározza a karakterek közös struktúráját és alapvető tulajdonságait. A diagramon látható tizenhat specifikus osztály (pl. *BartCassidy*, *SlabTheKiller*) mind a *BaseModel*-ből öröklődik. Mindegyik ilyen osztály egy-egy egyedi karaktert testesít meg, definiálva annak kezdő életerejét és nevét, illetve implementálva a karakterre jellemző különleges képességeket, a *BaseModel*-ben definiált metódusok felülírásával. Ez az öröklődési hierarchia biztosítja, hogy minden karakter rendelkezzen a szükséges alapvető attribútumokkal, miközben lehetővé teszi az egyedi viselkedések és tulajdonságok egységbe zárását (enkapszulációját) az adott karakterhez tartozó osztályban. A struktúra így hatékonyan modellezi a Bang! játék változatos karaktereit.

### Sequence diagram

13. ábra A Bang! kártya kijátszásának szekvencia diagramja



A 13. ábrán látató szekvenciadiagramon látszik, hogy amikor a játékos kijelöli a Bang! kártyát és a *Kiválasztott kártya kijátszása* gombot megnyomja, az elsőként meghívott metódus a *playSelectedCard()* a UI rétegben. Ez a metódus lekéri a kijelölt kártyát (*getSelectedCard()*), majd lekérdezi az aktuális játékost (*getCurrentPlayer()*), és megpróbálja végrehajtani a kártya hatását a *gameLogic.cardAction()* metóduson keresztül.

A *cardAction()* metódus eldönti, milyen típusú kártyáról van szó. A Bang! kártya DualTargetCard típusú, ezért meghívja a játékos *playDualTargetCard()* metódusát, amely delegálja a használatot a kártya *use()* metódusára.

A *use()* metódus belsejében történik a valódi játékszabályok ellenőrzése: először megnézi, hogy a célpont a játékos látótávolságán belül van-e, amit a látásmező (*fieldView*) és a fegyver hatótávolsága alapján számol ki. Ezután ellenőrzi, hogy a játékos még nem játszott ki Bang! kártyát az adott körben, hacsak nem rendelkezik a Rapid képességgel.

Ha a feltételek teljesülnek, akkor a célpont játékos *bangAction()* metódusa hívódik meg, amely a védekezés logikáját tartalmazza. Először ellenőrzi, van-e előtte egy hordó (*hasBarrel()*), ami automatikusan próbálja elhárítani a támadást. Ha ez sikertelen vagy nincs hordó, akkor megkísérel Nem talált! kártyát kijátszani – bot esetén automatikusan, emberi játékos esetén felhasználói választással (*chooseCard()*).

Ha a célpontnak nincs Nem talált! kártyája (vagy nem játszik ki egyet), akkor a támadás talál, és *receiveDamage()* metódus hívódik meg, ami csökkenti az életerejét.

A folyamat végén a kijátszott Bang! kártya eltávolításra kerül a támadó kezéből (*removeCard()*).

## Megvalósítás

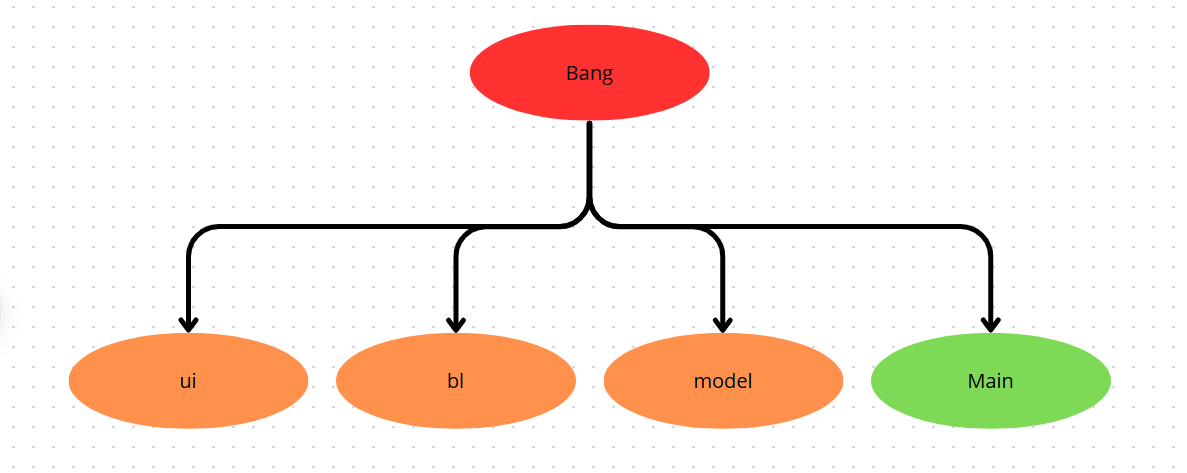
### Használt technológiák

A projekt megvalósítása során többféle technológia és eszköz került felhasználásra. Az alkalmazás alapját a Java 21 programozási nyelv adta, a fejlesztés pedig az IntelliJ IDEA Community Edition integrált fejlesztőkörnyezetben zajlott. A játék grafikus felhasználói felületének (GUI) kialakításához és a játékos interakcióinak kezeléséhez a beépített Java Swing[2] könyvtárat alkalmaztam. A fejlesztési folyamat követhetőségének biztosítása és a verziók hatékony kezelése érdekében a Git verziókezelő rendszert használtam, a kód távoli tárolására és megosztására pedig a GitHub platformot vettem igénybe. Ez a megközelítés lehetővé tette, hogy rendszeres mentéseket készítsek, így a fejlesztési szakaszok könnyen nyomon követhetővé váltak, és szükség esetén lehetőség nyílt korábbi állapotokra való visszatérésre. A projekt dokumentációjának elkészítése során a diagramok vizualizációjára a PlantUML eszközt, míg az írásos dokumentum összeállítására a Microsoft Word szövegszerkesztőt alkalmaztam. Fontos megjegyezni, hogy a projekt nem használ külön build-rendszert (mint például Maven vagy Gradle), hanem közvetlenül IntelliJ projektként jött létre és futtatható. A Java Swing keretrendszer biztosítja a teljes grafikus felhasználói felületet (GUI), amelyen keresztül a játékos a játékot vezérli. A Git verziókezelővel a fejlesztési folyamat során rendszeres mentéseket készítettem, hogy a fejlesztési szakaszok könnyen nyomon követhetők legyenek, és szükség esetén korábbi állapotokra visszatérhessek.

## Projekt szerkezete, CI/CD

### Projekt szerkezete

A projekt egy egyszerű, IntelliJ IDEA által kezelt, tiszta Java projekt, amely nem használ külön buildrendszert (pl. Maven, Gradle). A 14. ábráról leolvasható, hogy a projekt fájlszerkezete áttekinthető, a főbb csomagok logikailag jól elkülönítve tartalmazzák a különböző rétegekhez tartozó osztályokat:



14. ábra A projekt szerkezete vizuálisan

* A banggame.model csomag tartalmazza a játék alapvető adatszerkezeteit: játékosok, kártyák, szerepkörök és kezeli az adott adatszerkezethez vonatkozó szabályok zömét.
* A banggame.bl (business logic) csomag kezeli a játék inicializálását. Illetve, a játékmenet fő szabályait és folyamatát. Többek között a körök kezelését, kártyák húzását, és győzelmi feltétel esetleges teljesülését.
* A banggame.ui csomag tartalmazza a grafikus felhasználói felület összes elemét: panelek, ablakok, kezelőgombok.
* A Main.java osztályban történik a program inicializálása és a felhasználói felület elindítása.

### CI/CD folyamat

A projekt jelenlegi fejlesztési szakaszában nem került bevezetésre automatizált CI/CD (Continuous Integration/Continuous Deployment) rendszer. Ennek oka elsősorban a projekt jelenlegi mérete és az egyéni fejlesztői munkafolyamat, amely mellett az automatizálás előnyei még nem kerülnének teljes mértékben kihasználásra. A fejlesztési workflow ennek megfelelően manuális lépésekből állt. A forráskód módosításai helyileg, az IntelliJ IDEA fejlesztőkörnyezetben történtek. A kód integritásának és a változások követhetőségének biztosítása érdekében a módosításokat rendszeresen mentettem a Git verziókezelő rendszer segítségével, majd a változtatásokat feltöltöttem a projekt GitHubon található távoli repositoryjába. A szoftver buildelése és futtatása is közvetlenül az IntelliJ IDEA beépített eszközeivel történt. A funkcionalitás ellenőrzése, nagy mértékben manuálisan zajlott, elsősorban a grafikus felhasználói felületen keresztüli interakciókkal és a játékmenet kipróbálásával.

## Kódbázis felépítése, könyvtárszerkezet, forrásfájlok

A projekt forráskódja átlátható szerkezetben helyezkedik el, követve az MVC (Model-View-Controller) architektúra alapelveit. A csomagstruktúra célja, hogy elkülönítse a különböző funkcionális területeket: a megjelenítést, az üzleti logikát és az adatmodelleket.

### Főkategóriák

* banggame.model – Adatmodellek
* banggame.bl – Üzleti logika
* banggame.ui – Felhasználói felület
* Main.java – Belépési pont

#### banggame.model

Ez a csomag tartalmazza az összes osztályt, amelyek a játék adatainak tárolásáért felelősek. Itt találhatók a következők:

* BaseModel – Az összes játékos közös ősszülője. Tartalmazza az életerőt, kézben lévő kártyákat, asztali kártyákat stb.
* Character – Segíti a különböző karakterek leszármaztatását.
* BaseModel-ből származtatott osztályok – A 16 különféle karakter, mindegyik egyedi képességekkel.
* Card – Egy általános kártyát reprezentáló osztály.
* SingleTargetCard – A *Card*-ból származik, azon kártyák ősosztálya, melyeknek nem kell célpont a kijátszásukhoz.
* DualTargetCard – A *Card*-ból származik, azon kártyák ősosztálya, melyek egy bizonyos célpontra irányulnak.
* Card-ból származtatott osztályok – 22 egyedi *Card* osztály, különbözően felülírt függvényekkel.
* Role – Enum típusú osztály a szerepkörök kezelésére.
* RoleType – Enum típus a különböző szerepek definiálására.
* CardType – Enum típus a különböző kártyák definiálására (Bang!, Nem talált!, stb.).
* Deck – Ez az osztály generálja le a kártyapaklit és tartja számon a húzó- és dobópakli tartalmát.

Ezek az osztályok főként adatszerkezetek, getterekkel, setterekkel, és üzleti logikával.

#### banggame.bl

Az üzleti logika réteg, amely a játékmenet szabályait és folyamatait kezeli. A legfontosabb elemei:

* GameLogic – A játék fő vezérlője. Kezeli a köröket, a játékosok lépéseit, kártyák használatát, győzelem ellenőrzését.
* Bot – Az egyszerű mesterséges intelligencia, amely a botok körének automatikus lejátszását vezérli.
* GameInstance – Létrehozza a játékosokat és a kártypaklit. Illetve, a játék állapotát tárolja, aktív játékosok és a létrehozott *Deck* osztály.

Ez a réteg tartalmaz minden fontos szabályt és belső mechanizmust, amik a Bang! játékmenetét szimulálják.

#### banggame.ui

A felhasználói felületet megvalósító osztályok találhatóak itt, Swing alapokon. Főbb elemek:

* BangGameUI – A fő ablak, amely tartalmazza az összes vizuális elemet: játékos panelek, kártyák, logpanel, gombok.
* ImageUtils – Segít a felhasználói felület képeinek beállításában.
* MultiSelectCardLabel – Kezeli több kártya egyidejű kiválasztását.
* CardLabel, HiddenCardLabel – Különleges JLabel-ek, amelyek a kártyák megjelenítését (látható / rejtett módon) kezelik.
* MainMenuUI – A főmenü, ahol új játék indítható vagy a programból kiléphetünk.
* NewGameSettingsUI – Az új játék indításakor a játékosok szám választó ablak, ez hívja meg a *PlayerSetupUI*-t.
* PlayerSetupUI – Az új játék indításakor a karakter- és szerepkörválasztó ablak.
* GameOverDialog – A játék végén megadj a lehetőséget az új játék indítására.

Ezek az osztályok felelősek az eseménykezelésért is (pl. gombkattintásokra reagálás).

#### Main.java

A belépési pont, amely inicializálja az alkalmazást. Feladata:

* A program elindítása (*main()* metódus).
* A főmenü (*MainMenuUI*) megnyitása.

### Általános jellemzők

Kódolási stílus: A kód következetes névkonvenciókat követ (camelCase metódusok, PascalCase osztályok).

Egyszerűsített függőségek: A projekt kizárólag a Java SE beépített csomagjait használja (pl. javax.swing, java.util).

### Kód egyes elemeinek bemutatása

A projekt felépítése során végig törekedtem arra, hogy az egyes osztályok világosan elkülönüljenek egymástól, minden osztály jól körül határolható felelősségi körrel rendelkezzen. Ennek köszönhetően a kód átlátható marad, és bővítés, illetve karbantartás esetén is könnyen kezelhető.

#### bl csomag

A *bl.GameInstance* osztály az alkalmazás központi eleme, amely a Bang! játék egy adott futása során létrejövő egyetlen játékmenet teljes, aktuális állapotát tartalmazza és kezeli. Ez az osztály felelős azért, hogy egységes és konzisztens képet biztosítson a játék állapotáról, beleértve a résztvevő játékosokat (*BaseModel* leszármazott objektumok listájaként tárolva), a húzópaklit (*Deck*), valamint az információt arról, hogy éppen melyik játékos következik (*currentPlayerIndex*). Lényegében ez az osztály a játék "memóriája". Az osztály a Singleton tervezési mintát követi, amit a privát konstruktor és a statikus *getInstance()* metódus biztosít. Erre azért volt szükség, mert egy programfutás alatt csak egyetlen játékállapot létezhet, és a Singleton garantálja, hogy ebből az osztályból csupán egyetlen példány jöjjön létre. Ez biztosítja az állapot konzisztenciáját és könnyű, globális hozzáférést nyújt az üzleti logikai réteg (és közvetve más rétegek) számára anélkül, hogy az *GameInstance* objektumot explicit módon kellene továbbadni. Az osztály tartalmazza a játék inicializálásának logikáját is (*initializePlayers()*), amely felelős a játékosok számának megfelelő karakterek és szerepkörök kiosztásáért (akár véletlenszerűen, akár a felhasználói felületről kapott beállítások alapján), valamint a játékosok listájának átrendezéséért, hogy a Sheriff kezdjen. A *GameInstance* szorosan együttműködik a *GameLogic* osztállyal, amely innen olvassa ki az aktuális állapotot a döntéseihez és ide írja vissza a játékosok akcióiból vagy a játékmenetből következő állapotváltozásokat. A Bot osztály (gépi ellenfelek logikája) szintén felhasználja a *GameInstance*-ben tárolt adatokat a lépéseinek megtervezéséhez. Az osztály függ a model csomag több elemétől is, mint például a *BaseModel*, *Role*, *RoleType*, a konkrét karakterosztályok (*model.utilities.characters.\**) és a *Deck*. A felhasználói felület (UI) réteg közvetlenül nem lép vele interakcióba, de az általa megjelenített adatok forrása végső soron a *GameInstance*-ben tárolt állapot.

A *bl.GameLogic* osztály a Bang! alkalmazás központi vezérlője, amely az üzleti logika magját képezi. Felelős a játékmenet levezényléséért, a játékszabályok betartatásáért, a játékosok akcióinak feldolgozásáért, a körök kezeléséért és a játék végállapotának figyeléséért. Ez az osztály közvetít a játék állapotát tároló *GameInstance* és a felhasználói interakciókat kezelő *BangGameUI* között.

A *GameLogic* egy központi orchestrátor osztályként működik, amely logikailag elkülönül a játék állapotától (*GameInstance*) és a megjelenítéstől (*BangGameUI*). Létrehozásakor megkapja a *BangGameUI* példányát, amelyen keresztül kommunikál a felhasználóval és frissíti a felületet. Nem követ szigorúan egyetlen komplex tervezési mintát, de a felelősségi körök szétválasztása egyértelmű. A játékmenet egyes pontjain (pl. dinamit, börtön) szekvenciálisan kezeli az eseményeket, és beépített késleltetést (*sleepForSleepConstant()*) is alkalmaz a jobb követhetőség érdekében.

A *GameLogic* alapvetően meghatározza a játék működését, szorosan együttműködve több más komponenssel:

* GameInstance: Folyamatosan használja a Singleton *GameInstance*-t az aktuális játékállapot (játékosok, pakli, soron következő játékos) lekérdezésére és módosítására.
* BangGameUI: Üzeneteket küld a felhasználónak (*logUIMessage()*), frissíti a felületet (*UIUpdateUI()*), és bekéri a felhasználói döntéseket (pl. kártya-, célpont-, opcióválasztás a choose\* metódusokon keresztül). A játék végét is itt jelzi ki (*GameOverDialog()*).
* BaseModel (és leszármazottai): Meghívja a játékosokat reprezentáló *BaseModel* objektumok metódusait a körök különböző fázisaiban (pl. *gameStartDraw()*, *dynamiteAction()*, *endTurnDiscard()*).
* Bot: Amikor egy mesterséges intelligencia által vezérelt játékos kerül sorra, a *GameLogic* delegálja a vezérlést a statikus *Bot*.*takeTurn()* metódusnak.
* Card (és leszármazottai): Ismeri és kezeli a különböző kártyatípusokat (*SingleTargetCard*, *DualTargetCard*, stb.), például típusellenőrzéssel (instanceof) dönt a szükséges további interakciókról (pl. célpont kérése).

A *bl.Bot* osztály tartalmazza a mesterséges intelligencia (MI) logikáját, amely a számítógép által vezérelt játékosok (botok) viselkedését irányítja a Bang! játékban. Ez az osztály felelős azért, hogy a bot játékosok a körükben észszerű (bár egyszerű) döntéseket hozzanak a lapjaik felhasználásáról és a célpontok kiválasztásáról. Mindezt statikus metódusok segítségével valósítja meg.

Az osztály statikus metódusokat használó utility (segédprogram) osztályként lett implementálva, nem pedig példányosítható objektumként. Ez azt jelenti, hogy nincs külön *Bot* példány minden egyes MI játékoshoz; ehelyett a központi *GameLogic* hívja meg a *Bot.takeTurn()* statikus metódust, átadva neki az aktuálisan soron lévő bot (*BaseModel*) objektumot. A bot logikája egy előre meghatározott lépéssorozatot követ: kártyahúzás, automatikus felszerelés kijátszása (pl. jobb fegyver, védelmi lapok), majd a kézben maradt lapok végig tekintése és kijátszása egyszerűbb szabályok és heurisztikák alapján (pl. gyógyulás szükség esetén, támadás elérhető célpontra, egyéb akciók végrehajtása). A célpontválasztás többnyire véletlenszerű az érvényes lehetőségek közül. Az osztály késleltetést (*sleepForSleepConstant()*) is használ a bot lépései között, hogy a játék követhetőbb legyen a felhasználó számára.

A Bot osztály működése szorosan összefügg a rendszer többi részével:

* GameLogic: A *GameLogic* osztály hívja meg a *Bot.takeTurn()* metódust, amikor egy bot játékos kerül sorra. A Bot logikája visszahív a *GameLogic*-ba felhasználói üzenetek kiírására (*logUIMessage()*) és a UI frissítésére (*UIUpdateUI()*).
* BaseModel (és leszármazottai): A Bot metódusai paraméterként kapják meg az aktuális bot játékost (BaseModel bot), és ennek az objektumnak az állapotát olvassák (pl. *getHandCards()*, *getHealth()*, *hasWeapon()*), valamint ennek a metódusait hívják meg a tényleges akciók végrehajtására (pl. *roundStart(), playSingleTargetCard(), playDualTargetCard()*).
* Card (és leszármazottai): A bot logikája elemzi a *Card* objektumokat a játékos kezében, típusuk (instanceof, *getType()*) alapján döntve arról, hogyan és mikor játssza ki őket, figyelembe véve az olyan szabályokat, mint pl. hogy körönként csak egy Bang! játszható ki (kivéve speciális képességek esetén).
* GameInstance: Bár közvetlenül ritkán hivatkozik rá, a *Bot* a *BaseModel* objektumon keresztül hozzáfér a *GameInstance*-hez (pl. *bot.getGameInstance().getPlayers()*) a játék általános állapotának (pl. többi játékos) lekérdezéséhez a döntéseihez.

#### Model csomag

A model.cards.Card osztály a Bang! játékban szereplő összes kártya alapvető, absztrakt ősosztálya. Ennek az osztálynak a fő célja, hogy definiálja azokat a közös attribútumokat és alapvető információkat, amelyek minden egyes kártyalapra jellemzőek, függetlenül annak konkrét típusától vagy működésétől. Ilyen közös tulajdonságok a kártya neve, színe (*suit*), értéke (*value*), típusa (*CardType* enum segítségével) és a hozzá tartozó kép elérési útja (*imagePath*).

Az osztály abstract-ként lett definiálva, mivel önmagában nem képvisel egy konkrét, játszható kártyát, hanem csak egy közös interfészt és adattároló struktúrát biztosít a leszármazottak számára. Az alapvető kártya adatokat a konstruktorban kapja meg, és ezeket (a név, szín, érték stb.) jellemzően nem módosítható (immutable) módon tárolja. A konkrét kártyák viselkedését a belőle származtatott alosztályoknak kell implementálniuk. Az osztály egyszerű getter metódusokat biztosít az alapvető tulajdonságok lekérdezéséhez, valamint egy *toString()* metódust az alap adatok szöveges megjelenítéséhez.

A Card osztály a kártyahierarchia csúcsán helyezkedik el:

* Leszármazott osztályok: Minden konkrét kártyaosztály (pl. *BangCard*, *BarrelCard*) közvetlenül vagy közvetve ebből az osztályból öröklődik. Az olyan további absztrakt osztályok, mint a *SingleTargetCard* és a *DualTargetCard*, szintén a *Card*-ból származnak, és absztrakt *use()* metódusokat definiálnak, amelyek előírják a leszármazottaik számára, hogy hogyan implementálják az egy vagy több célpontot igénylő kártyák kijátszásának logikáját.
* Deck: A *Deck* osztály *Card* objektumok gyűjteményét tárolja és kezeli.
* BaseModel (Player): A játékosok (*BaseModel* leszármazottak) a kezükben és/vagy az asztalukon tárolnak *Card* objektumokat.
* GameLogic, Bot: Ezek az osztályok felhasználják a *Card* objektumokat, lekérdezve azok típusát (*getType()*) és egyéb tulajdonságait a játékszabályok alkalmazásához és a döntéshozatalhoz.

A *model.utilities.BaseModel* osztály a Bang! játék alkalmazás egyik legfontosabb és legösszetettebb osztálya. Ez testesíti meg a játékban részt vevő egyes játékosokat, legyen szó emberi vagy mesterséges intelligencia által irányított (bot) szereplőről. Az osztály felelőssége rendkívül szerteágazó: egyrészt ez az osztály őrzi egy adott játékos teljes, releváns állapotát, másrészt implementálja a játékossal kapcsolatos alapvető viselkedési formákat és interakciókat. Az állapot magában foglalja a játékos identitását (a hozzá rendelt *Character* és *Role* objektumokon keresztül), aktuális életerejét (*health*, *maxHP*), a kezében (*handCards*) és az asztalon (*tableCards*) lévő kártyákat, beleértve a specifikus felszereléseket (fegyver, hordó stb.), valamint a játékmenet szempontjából fontos jelzőket (pl. lőtávolságra és láthatóságra vonatkozó módosítók, *bangedThisRound* flag). Ezen túlmenően ez az osztály definiálja azokat a műveleteket, amelyeket egy játékos végrehajthat (pl. kártyahúzás, lap kijátszása) és azokat a reakciókat, amelyeket adhat bizonyos eseményekre (pl. Bang!-re válasz, indiánok támadása elleni védekezés).

A *BaseModel* egy központi osztály, amely egy játékos szinte minden aspektusát kezeli. A konstruktora megkapja a játékos *Character* és *Role* objektumait, amelyek meghatározzák annak alapvető tulajdonságait (név, maximális életerő, speciális képességek azon formája, mikor az adott karakter képessége, más karaktereken múlik [például Vulture Sam megkapja a kieső játékos lapjait]). A Sheriff szerepkörhöz automatikusan egy plusz maximális életerő jár. Az osztály egy *isBot* logikai flag segítségével különbözteti meg az emberi és gépi játékosokat, ami lehetővé teszi, hogy ugyanaz az osztály eltérően viselkedjen: az emberi játékosok esetében a döntésekhez (pl. kártya-, célpontválasztás) a *GameLogic*-on keresztül a felhasználói felülethez fordul, míg a botok esetében vagy belső logikát használ, vagy a Bot osztály metódusait hívja meg (pl. *Bot.discardExcessCards()*). Az osztály közvetlen referenciát tárol a *GameInstance* Singleton példányára, hogy hozzáférjen a közös játékállapothoz, elsősorban a paklihoz (*Deck*). Különleges tervezési döntés, hogy a gyakran ellenőrzött asztali lapok (fegyver, hordó, börtön stb.) külön attribútumként is szerepelnek a *tableCards* lista mellett, a gyorsabb hozzáférés és állapotellenőrzés (*hasBarrel(),* *inJail()* stb.) érdekében. A játékosok közötti távolság (*vision*) kiszámításának logikája *(setVision()*) szintén ebben az osztályban kapott helyet, figyelembe véve a saját és mások látótávolságot módosító lapjait (Mustang, Scope). A kártyák kijátszásának és a beérkező támadásokra/eseményekre adott reakcióknak a logikája (pl. *bangAction(), indiansAction(), duelAction()*) nagyrészt közvetlenül ebben az osztályban került implementálásra.

A BaseModel központi szerepe miatt számos más osztállyal áll kapcsolatban:

* Character, Role, RoleType: A konstruktorban kapja meg a *Character* és *Role* példányokat, amelyek meghatározzák az identitását és alapvető képességeit. A *Role* a *RoleType* enumot használja.
* GameInstance: Referenciát tárol rá, hogy elérje a közös *Deck*-et a kártyahúzáshoz (*drawCard()*).
* GameLogic: Nagyon szoros az együttműködés. A *GameLogic* vezérli a játékmenetet és hívja a *BaseModel* metódusait a körök fázisainak (pl. *roundStart()*, *endTurnDiscard()*) és a játékosok közötti interakciók (pl. *receiveDamage(), bangAction()*) kezelésére. A *BaseModel* pedig visszahív a *GameLogic*-ba, amikor emberi játékos esetén felhasználói döntésre van szükség (pl. *gameLogic.chooseCard()*). A játékos halálakor a *BaseModel* értesíti a *GameLogic*-ot (*gameLogic.aPlayerRemoved()*).
* Card (és alosztályai): Tárolja a *Card* objektumokat a *handCards* és *tableCards* listákban, valamint specifikus kártyatípusokat külön attribútumokban. Meghívja a *SingleTargetCard* és *DualTargetCard* *use()* metódusait a kártyák kijátszásakor. Implementálja a specifikus kártyákra (pl. *MissedCard*, *BangCard*, *BeerCard*) adott reakciók logikáját.
* Bot: Emberi játékosoktól eltérő viselkedés implementálásához használja (pl. a kötelező lapdobásnál).
* Saját alosztályai (Karakterek): Explicit metódushívások formájában (a karakter-specifikus logika felülírt metódusokban), a *BaseModel* példányai valójában konkrét karakterek (pl. *SuzyLafayette*, *VultureSam*), és bizonyos logikai pontokon (pl. *SuzyLafayette* laplopás utáni húzása, *VultureSam* kártyagyűjtése halálkor) figyelembe veszi ezeket a speciális eseteket (instanceof ellenőrzéssel).

#### ui csomag

A *ui.PlayerSetupFrame* egy Swing JFrame ablak, amely a játék indítása előtti konfigurációs lépést valósítja meg. Miután a felhasználó kiválasztotta a játékosok számát (ezt az információt a konstruktorban kapja meg), ez az ablak jelenik meg, ahol minden egyes játékoshoz külön-külön beállítható, hogy bot (Igaz/Hamis) legyen-e, melyik karaktert (Random vagy specifikus) kapja, és milyen szerepkört (Random vagy specifikus) töltsön be.

Az ablak szabványos Swing komponenseket (JPanel, JLabel, JComboBox, JButton) használ a beállítások megjelenítésére és bekérésére, a játékosok számának megfelelően dinamikusan felépítve a felületet. Beépített validációs mechanizmust (*validateInputs()*) tartalmaz, amely a kiválasztások módosításakor automatikusan lefut. Ez a validáció ellenőrzi a játékszabályok betartását (pl. pontosan egy emberi játékos, karakterek egyedisége, szerepkörök megfelelő száma a játékosszámhoz képest) és vizuális visszajelzést ad a felhasználónak egy hibaüzeneteket megjelenítő panelen (*errorPanel*), valamint a hibás beállítások kiemelésével (pl. rózsaszín háttér). Az *Új játék indítása* gomb csak akkor aktív, ha minden beállítás érvényes.

* GameInstance: Felhasználja a *GameInstance* szolgáltatását az elérhető karakternevek lekérdezéséhez a JComboBox feltöltéséhez.
* BangGameUI & GameLogic: Sikeres validáció és az *Új játék indítás* gomb megnyomása után összegyűjti a felhasználói beállításokat (botok, karakterek, szerepkörök listáit). Létrehozza a fő játékablakot (*BangGameUI*), elkéri annak *GameLogic* példányát, majd meghívja a *gameLogic.startGame()* metódust a játék elindításához a megadott konfigurációval. Ezt követően bezárja saját magát és láthatóvá teszi a *BangGameUI*-t.
* ImageUtils: Segítségével tölti be az ablak ikonját.

*ui.BangGameUI* osztály a Bang! játék alkalmazás központi felhasználói felülete, amely egy JFrame ablakban valósul meg. Ez az osztály felelős a teljes játékállapot vizuális megjelenítéséért az emberi játékos számára, valamint az interakciók fogadásáért és feldolgozásáért. Megjeleníti a játékos saját lapjait, az asztalon lévő felszereléseket, az ellenfelek állapotát (életerő, lapok száma, kijátszott lapok), a játék eseményeit egy naplóterületen (*logTextArea*), és gombokat biztosít a főbb akciókhoz (lap kijátszása, eldobása, kör befejezése).

Az osztály egy tipikus Swing alkalmazásablakot valósít meg, szabványos komponensek (JPanel, JLabel, JButton, JTextArea, JComboBox stb.) és elrendezéskezelők (BorderLayout, GridLayout, FlowLayout) használatával. A felület logikailag több panelre van osztva, amelyek a játék különböző területeit reprezentálják (saját kéz, ellenfelek, asztal közepe, napló és vezérlők).

Lényeges tervezési döntés, hogy a *BangGameUI* példányosításakor létrehozza a saját *GameLogic* példányát, és átadja saját magát (*this*) a *GameLogic* konstruktorának. Ez egy szoros kapcsolatot alakít ki, ahol a UI "birtokolja" a fő játéklogikai vezérlőt, és lehetővé teszi a *GameLogic* számára, hogy visszahívjon a UI-ba (pl. üzenetek logolására vagy felhasználói input bekérésére).

A felhasználói felület frissítése egy központi *updateUI()* metóduson keresztül történik. Ezt a metódust kell meghívni minden olyan állapotváltozás után, amely vizuális megjelenítést igényel. Az *updateUI*() felelős azért, hogy a *GameInstance*-ben (a *GameLogic*-on keresztül elért) aktuális adatok alapján újrarajzolja a dinamikus felületi elemeket (pl. a játékos kezében lévő lapokat, az ellenfelek paneljeit).

A játék közben szükséges felhasználói döntések (pl. kártya kiválasztása a kézből, célpont kijelölése, két opció közüli választás) kezelésére a *BangGameUI* különálló metódusokat (*selectCardFromList(), selectTargetFromList(), showTwoOptionDialog(), selectTwoCardsFromThree()*) biztosít. Ezeket a metódusokat a *GameLogic* hívja meg, amikor a játékmenet emberi beavatkozást igényel. A metódusok modális dialógusablakokat (JDialog, JOptionPane) jelenítenek meg, amelyek blokkolják a további végrehajtást, amíg a felhasználó nem választ, biztosítva a szinkron interakciót. Ezek a dialógusok gyakran használnak egyedi *CardLabel* vagy *MultiSelectCardLabel* komponenseket a kártyák grafikus megjelenítéséhez és kiválasztásához.

Az osztály az *ImageUtils* segédosztályra támaszkodik a játékhoz szükséges képek (kártyalapok, ikonok) betöltéséhez.

A BangGameUI a felhasználói felület központjaként számos más osztállyal áll kapcsolatban:

* GameLogic: Létrehozza és tárolja a *GameLogic* példányát. A felhasználói műveletek (gombok megnyomása) hatására meghívja a *GameLogic* megfelelő metódusait (pl. *cardAction(), discardCardAction(), endTurn()*). Átadja magát a *GameLogic*-nak, hogy az visszahívhasson üzenetek kiírására (*logMessage()*) vagy modális dialógusok megjelenítésére (*select\** metódusok). A *GameLogic* hívja meg az *updateUI()* metódust a felület frissítésére.
* GameInstance: Közvetve, a *GameLogic*-on keresztül fér hozzá a *GameInstance*-ben tárolt adatokhoz, amelyek szükségesek az *updateUI(*) metódusban a játékállapot megjelenítéséhez (pl. játékoslista, pakli állapota, dobópakli).
* BaseModel: A *BaseModel* objektumok reprezentálják a játékosokat a felületen. Az *updateUI()* metódus lekéri a játékosokat (pl. *gameLogic.getHumanPlayer(), gameLogic.getPlayers()*) és megjeleníti azok adatait (név, HP, kézben lévő lapok száma, asztali lapok). A *targetPlayerSelector* JComboBox is BaseModel objektumokkal van feltöltve.
* Card: A *Card* objektumok vizuálisan jelennek meg a felületen, vagy *CardLabel* (esetleg *MultiSelectCardLabel*) komponensként a kézben és dialógusokban, vagy JButton-ként az asztalon lévő lapok esetében.
* CardLabel, HiddenCardLabel, MultiSelectCardLabel: Ezeket a specializált UI komponenseket használja a kártyák és a rejtett lapinformációk megjelenítésére, valamint a kártyaválasztási interakciók megvalósítására.
* ImageUtils: Ezt a segédosztályt használja a grafikus erőforrások betöltésére.
* Swing Komponensek: JFrame-ből származik, és széleskörűen használja a Swing könyvtár elemeit (panelek, gombok, címkék, szövegterület, legördülő listák, dialógusablakok) a felület felépítéséhez és működtetéséhez.

A Bang! kártyajáték implementációja egy rétegzett architektúrán valósult meg, amely sikeresen különválasztja az adatmodellt (pl. *Card*, *BaseModel*), a központi játéklogikát és állapotkezelést (pl. *GameLogic*, *GameInstance*, *Bot*), valamint a felhasználói felületet (pl. *BangGameUI*). Ez a strukturált, moduláris megközelítés egy jól szervezett, karbantartható és bővíthető rendszert eredményezett, amely hatékonyan kezeli a Bang! összetett szabályrendszerét és biztosítja a játék működőképes megvalósítását.

### Bonyolultabb kódrészletek elmagyarázása

#### GameInstance.initializePlayers Metódus Magyarázata

Ez a metódus felelős a játékmenet elindításához szükséges játékos-lista (*players*) létrehozásáért és feltöltéséért a *GameInstance* osztályon belül. A metódus a felhasználói felületről (konkrétan a *PlayerSetupFrame*-ből) kapott konfigurációs adatok alapján állítja be a játékosokat.

Metódus szignatúra:

**public void initializePlayers(int count, List<String> characterNames, List<String> roleNames, List<String> bots)**

* **count**: A játékosok teljes száma.
* **characterNames**: Lista a kiválasztott karakterek neveiről (lehet "Random" is).
* **roleNames**: Lista a kiválasztott szerepkörök neveiről (lehet "Random" is).
* **bots**: Lista, amely jelzi, hogy az adott játékos bot ("Igaz") vagy ember ("Hamis").

Működés Lépései:

1. Inicializálás: A metódus első lépésként kiüríti a meglévő *players* listát (*players.clear()*), hogy tiszta lappal induljon az új játék beállítása. Ezután meghívja a *getRolesForGame()* segédmetódust, amely a játékosok száma (*count*) alapján előállítja a kiosztandó szerepkörök (*Role* objektumok) alap listáját (figyelembe véve a Bang! szabályait a szerepek számára vonatkozóan, és esetleg kiszűrve azokat, amelyeket a felhasználó specifikusan választott). Lekéri az összes elérhető karakter nevét is (*getAllCharacterNames()*).
2. Játékosok Létrehozása (Ciklus): Egy for ciklus segítségével végigmegy minden egyes létrehozandó játékoson (0-tól count-1-ig). Minden ciklusiteráció egy játékos beállítását végzi:

* Szerepkör kiosztása: Megvizsgálja az adott játékoshoz tartozó *roleName*-et.
  + Ha a név "Random", akkor a *getRandomRole()* segédmetódus segítségével kiválaszt és eltávolít egyet a még elérhető *roles* listából, majd ezt a véletlenszerűen kapott *Role* objektumot rendeli a játékoshoz.
  + Ha a név specifikus (pl. "Sheriff", "Outlaw"), akkor egy switch szerkezet segítségével létrehozza a megfelelő *RoleType*-pal rendelkező *Role* objektumot.
  + Mindkét esetben eltárolja a Sheriff pozícióját (*sheriffIndex*) az i ciklusváltozó alapján.
* Karakter kiosztása: Hasonlóan a szerepkörhöz, megvizsgálja a *characterName*-et.
  + Ha "Random", a *getRandomCharacter()* segédmetódus véletlenszerűen kiválaszt egy nevet a még elérhető karakternevek listájából (*allCharacterNames()*), majd eltávolítja azt a listából.
  + Ha a név specifikus, akkor azt használja.
  + Ezután a *getCharacterFromCharacterName()* segédmetódus megkeresi a karakter nevéhez tartozó konkrét Class objektumot (pl. *LuckyDuke.class, CalamityJanet.class*). Fontos, hogy ezek az osztályok a *BaseModel* leszármazottai.
* Játékos példányosítása: Ez a metódus egyik legkomplexebb része. Miután megvan a konkrét karakter osztálya (*characterClass*), a kiosztott szerepkör (*role*) és a bot státusz (*bots.get(i)*), a program Java Reflection API-t használ a játékos objektum létrehozásához:  
  **player = characterClass.getDeclaredConstructor(Role.class, boolean.class).newInstance(role, isBot);**
* Ez a sor dinamikusan megkeresi a *characterClass*-ban azt a konstruktort, amely egy *Role* és egy boolean paramétert vár, majd meghívja azt a *role* és a *bot* státusz (igaz/hamis) értékekkel. Az így létrejött konkrét karakter-objektumot (amely *BaseModel* típusú) hozzáadja a *players* listához. A try-catch blokk a reflectionnel kapcsolatos esetleges hibákat (pl. ha nem található megfelelő konstruktor) kezeli.

1. Sheriff előre rendezése: A ciklus lefutása után, miután minden játékos létrejött a listában, meghívja a *reorderSheriffFirst(sheriffIndex)* segédmetódust. Ez a metódus átrendezi a *players* listát úgy, hogy a korábban eltárolt *sheriffIndex* pozícióban lévő játékos (a Sheriff) kerüljön a lista 0. indexére, biztosítva, hogy a játék a Sheriffel kezdődjön. A Sheriff előtti játékosok az eredeti sorrendben a sor végére kerülnek.

#### BaseModel.duelAction Metódus Magyarázata

Ez a metódus segít megérteni a kártyák *BaseModelre* kiváltott akcióját, hiszen mindegyik hasonló logika alapján épül fel. Ez a metódus aktiválódik akkor, amikor egy játékos (this objektum) egy Párbaj kártya célpontjává válik, vagy amikor egy folyamatban lévő párbajban őrá kerül a sor, hogy reagáljon. A metódus felelőssége annak eldöntése, hogy a játékos tud-e (vagy akar-e) egy Bang! kártyát kijátszani a párbaj folytatásaként, vagy elszenvedi a következményt (sebzést).

Metódus Szignatúra:

**public void duelAction(BaseModel target, GameLogic gameLogic)**

* **target**: Az a *BaseModel* objektum (játékos), aki a párbajt kezdeményezte, vagy aki utoljára játszott ki Bang! lapot a párbaj során. Ha this játékos nem tud/akar reagálni, akkor ettől a *target* játékostól szenved sebzést.
* **gameLogic**: Referencia a *GameLogic* objektumra, amely szükséges a felhasználói interakciókhoz (emberi játékos esetén) és potenciálisan más játékállapot-információk eléréséhez.

Működés Lépései:

1. Bot vagy emberi játékos megkülönböztetése: A metódus először ellenőrzi a *this.isBot()* flag értékét, hogy eldöntse, az aktuálisan reagáló játékos gép vagy ember.

* Bot Logika: Ha a játékos bot (*isBot* == true):
  + Végigiterál a kezében lévő lapokon (*this.getHandCards()*).
  + Ha talál egy *BangCard* típusú lapot (instanceof *BangCard*), akkor azt a lapot "kijátssza": meghívja a *removeCard(bangCard)* metódust (ami eldobja a lapot és eltávolítja a kézből).
  + Ezután rekurzívan meghívja az ellenfél (*target*) *duelAction* metódusát (*target.duelAction(this, gameLogic)*), átadva saját magát (this) mint az új kihívót. Ezzel a párbaj "visszapattan" az ellenfélhez.
  + A return; utasítással a bot befejezi a reagálását erre a párbaj-körre.
  + Ha a ciklus végigfut anélkül, hogy a bot Bang! lapot talált volna, a vezérlés a sebződéshez ugrik. A bot az első Bang! lapot használja fel, amit talál.
* Emberi Játékos Logika: Ha a játékos ember (*isBot* == false):
  + Egy while(true) ciklusba lép, amely addig fut, amíg a játékos érvényes lépést nem tesz (Bang! kijátszása vagy passzolás).
  + A cikluson belül meghívja a *gameLogic.chooseCard()* metódust. Ez a metódus (a *BangGameUI*-n keresztül) megjelenít egy dialógusablakot a felhasználónak, ahol kiválaszthat egy lapot a kezéből, vagy dönthet a passzolás mellett (pl. a dialógus bezárásával).
  + Ha a játékos Bang! lapot választ: A program ellenőrzi (instanceof *BangCard*), hogy a választott lap valóban Bang! volt-e. Ha igen, akkor a lapot eldobja (*discardCard()*, *handCards.remove()*), majd a bothoz hasonlóan rekurzívan meghívja az ellenfél (*target*) *duelAction*() metódusát, és a return; utasítással kilép a metódusból.
  + Ha a játékos passzol: Amennyiben a *gameLogic.chooseCard*() null-t ad vissza (a felhasználó passzolt), a break; utasítással kilép a while ciklusból, és a vezérlés a sebződéshez ugrik.
  + Ha a játékos más lapot választ: Ha a felhasználó nem Bang! lapot és nem is null-t választ, a ciklus egyszerűen újra kezdődik, ismét bekérve a választást.

1. Sebződés (Ha nincs Bang!): Ha a játékos (akár bot, akár ember) nem játszott ki Bang! lapot (a bot nem talált, az ember passzolt), akkor a vezérlés eljut a *receiveDamage(1, target, gameLogic)* metódus hívásához. Ez azt jelenti, hogy az aktuális játékos (this) elszenved 1 pont sebzést attól a játékostól (*target*), aki a párbajt indította vagy utoljára sikeresen reagált.

#### Bot.setYetToPlay Metódus Magyarázata

Ezen metódus megismerésével betekintést nyerhetünk a *Bot* osztály működésének alapvető logikájába. Ez a private static segédmetódus a *Bot* osztályon belül azt a célt szolgálja, hogy az adott bot játékos (*BaseModel bot*) aktuális kézben tartott lapjai (*handCards*) közül kiválogassa azokat, amelyeket a körének fő akciófázisában potenciálisan ki tud (vagy érdemes) játszania. Lényegében egy szűrési logikát valósít meg, amely előkészíti a kijátszandó akciókártyák listáját a *takeTurn*() metódus számára, megvalósítva ezzel a bot egyszerűbb stratégiai döntéseinek egy részét.

Metódus Szignatúra:

**private static List<Card> setYetToPlay(BaseModel bot)**

bot: Az a *BaseModel* objektum (*bot játékos*), akinek a kezét elemezzük.

Visszatérési érték: Egy List<Card>, amely azokat a kártyákat tartalmazza a bot kezéből, amelyeket a metódus logikája alapján kijátszhatónak ítélt.

Működés Lépései:

1. Inicializálás: A metódus létrehoz egy másolatot a bot aktuális kézben tartott lapjairól (hand), egy üres listát az eredmények tárolására (*cardsYetToPlay*), egy *alreadyHasBang* logikai változót (kezdetben false), ami azt követi, hogy adtunk-e már hozzá Bang! lapot a listához (a körönkénti egy Bang! szabály miatt), és egy *beerInPocket* számlálót a hozzáadott gyógyító lapok nyomon követésére.
2. Lapok Szűrése (Ciklus): Egy for ciklussal végigmegy a kézben lévő lapok másolatán (hand):

* Felszerelés/Kék lapok kihagyása: Azonnal átugorja (continue) azokat a lapokat, amelyek felszerelések (*Weapon*) vagy tipikusan "passzív" kék lapok (*BarrelCard*, *MustangCard*, *ScopeCard*, *DynamiteCard*). A bot logikája feltételezi, hogy ezeket már az *autoEquip*() metódus (ha lehetséges volt) kijátszotta, vagy nem a fő akciófázisban kerülnek felhasználásra.
* Gyógyító lapok (Sör/Kocsma) kezelése: Ha a lap *BeerCard* vagy *SaloonCard*:
  + Csak akkor veszi fontolóra a kijátszását, ha kettőnél több játékos van még játékban (mivel a Sörnek nincs hatása, ha már csak ketten maradtak).
  + Továbbá csak akkor adja hozzá a *cardsYetToPlay* listához, ha a botnak hiányzik életereje (*bot.getMaxHP()* - *bot.getHealth()* > 0), ÉS a már hozzáadott gyógyító lapok száma (*beerInPocket*) még nem érné el a maximális gyógyulás mértékét. Ezzel elkerüli a felesleges gyógyítást. A *beerInPocket* számlálót növeli, ha hozzáad egy ilyen lapot.
* Bang! kezelése: Ha a lap *BangCard*:
  + Csak akkor veszi fontolóra, ha a botnak van "rapid fire" képessége (*bot.getRapid()* pl. *Volcanic* fegyver miatt), VAGY ha még nem adott hozzá Bang! lapot ebben a körben (!*alreadyHasBang*).
  + Mielőtt hozzáadná, ellenőrzi, hogy van-e egyáltalán lőtávolon belül (*bot.getVision(), bot.getWeapon(), bot.getFieldView()* alapján számolva) érvényes célpont. Ha nincs lőtávolon belül senki, a Bang! lapot nem adja hozzá a listához (nem érdemes kijátszani).
  + Ha a feltételek teljesülnek és van célpont, hozzáadja a Bang! lapot a *cardsYetToPlay* listához, és az *alreadyHasBang* flaget true-ra állítja.
* Speciális eset (Calamity Janet): Ha a lap *MissedCard* ÉS a bot karaktere *CalamityJanet* (instanceof *CalamityJanet*), akkor a Nem talált! lapot úgy kezeli, mintha Bang! lenne (azaz hozzáadja a listához, ha a "körönként egy Bang!" szabály engedi, és van célpont). Ez implementálja Calamity Janet különleges képességét. Más botok a Nem talált! lapot természetesen nem adják hozzá támadó szándékkal.
* Egyéb Akciókártyák: Minden más kártyatípust, amelyet a fenti logika nem szűrt ki (pl. *Panic, CatBalou, WellsFargo, Stagecoach, Indians, Gatling, Duel, Jail, GeneralStore*), alapértelmezetten hozzáad a *cardsYetToPlay* listához. A bot feltételezi, hogy ezeket a lapokat általában érdemes vagy lehetséges kijátszani, ha a kezében vannak.

1. Visszatérés: A ciklus lefutása után a metódus visszaadja a *cardsYetToPlay* listát, amely tartalmazza azokat a kártyákat, amelyeket a bot megkísérelhet kijátszani a körében.

## Tesztelés

### Használt tesztelési módszerek, tesztautomatizálás

A projekt tesztelési stratégiája – összhangban a fejlesztési erőforrásokkal és a projekt jelenlegi fázisával – teljes mértékben manuális tesztelési módszerekre épült. A fejlesztés során nem került sor automatizált tesztek (sem egység-, sem integrációs) implementálására, és nem használtam dedikált tesztautomatizálási keretrendszereket vagy CI/CD folyamatokat az automatikus ellenőrzéshez.

A tesztelés gerincét a manuális funkcionális tesztelés adta, amelyet közvetlenül a grafikus felhasználói felületen (*BangGameUI*) keresztül végeztem. Ennek során a főbb funkciókat ellenőriztem: új játék indítása és beállítása (*PlayerSetupFrame*), a játékmenet alapvető lépései (kártyahúzás, kijátszás, eldobás), a központi játékszabályok érvényesülése (pl. lőtávolság, kártyaeffektusok), a botok (*Bot*) viselkedése a körükben, valamint a játék végi feltételek helyes kezelése. Ez a megközelítés elsősorban fekete dobozos tesztelésnek felelt meg, ahol a rendszer működését a külső interfészen keresztül figyeltem meg, esetenként a belső állapot (pl. log üzenetek, debugger) ellenőrzésével kiegészítve.

Az integrációs tesztelés, vagyis a különböző komponensek (UI, Üzleti Logika, Modell) együttműködésének ellenőrzése szintén manuálisan, a funkcionális tesztelés részeként történt. Az end-to-end forgatókönyvek (pl. egy kártya kijátszása a UI-tól a játékállapot (*GameInstance*) frissüléséig és a UI visszajelzéséig) végigjátszásával győződtem meg arról, hogy a rétegek és osztályok megfelelően kommunikálnak egymással.

Automatizált unit tesztek nem készültek az egyes osztályok vagy metódusok izolált ellenőrzésére. Az egyes logikai egységek helyességét fejlesztés közben, célzott manuális próbákkal és a beépített fejlesztőkörnyezeti eszközökkel ellenőriztem.

### Unit- tesztek összefoglalása

A projekt fejlesztése során nem került sor dedikált, automatizált egységtesztek (unit tesztek) írására JUnit vagy hasonló keretrendszer használatával. Az egyes logikai komponensek és metódusok helyes működésének ellenőrzése elsősorban a manuális funkcionális tesztelés keretében, valamint a fejlesztés közbeni célzott hibakeresés (debugging) és kódátvizsgálás során történt meg.

#### Példák a manuális "unit" ellenőrzésekre:

Annak ellenére, hogy nem készültek automatizált tesztek, a fejlesztés során több kulcsfontosságú logikai egység működését manuálisan ellenőriztem, mintegy szimulálva egy unit teszt célját. Ezek a 2., 3., 4. táblázaton láthatóak:

|  |  |
| --- | --- |
| Távolságszámítás (*BaseModel.setVision()*) ellenőrzése: | |
| Cél: | Annak verifikálása, hogy a játékosok közötti távolságot helyesen számolja-e a rendszer, figyelembe véve a Musztáng és Távcső lapokat. |
| Manuális lépések: | * Játék indítása meghatározott számú játékossal. * Ideiglenes *System.out.println()* utasításokkal kiírattam a *setVision()* által kalkulált vision lista tartalmát egy adott játékos körének kezdetén. * Manuálisan hozzáadtam/eltávolítottam Távcső és Musztáng lapokat játékosokhoz. * Újra futtattam a *setVision()* logikáját (új kör indításával), és ellenőriztem, hogy a kiíratott távolságértékek megfelelnek-e a várakozásoknak. |
| Eredmény: | Az eredményeket összevetettem a szabályok alapján manuálisan kiszámított értékekkel és nem találtam különbséget. |

2. táblázat BaseModel.setVision() ellenőrzése

|  |  |
| --- | --- |
| Sebződés és Sör használat (*BaseModel.receiveDamage()*) ellenőrzése: | |
| Cél: | Ellenőrizni, hogy a játékos helyesen használhat-e Sör lapot a halál elkerülésére, amikor életereje 0-ra vagy az alá csökken. |
| Manuális lépések: | * Olyan játékhelyzetet idéztem elő, ahol az emberi játékosnak 1 életereje és legalább egy Sör lapja van a kezében. * Egy másik játékossal Bang! lapot játszattam ki erre a játékosra. * Figyeltem a UI felugró ablakát, amely a Sör lap kijátszását ajánlja fel (ezt a *receiveDamage()* metódus hívja meg a *GameLogic*-on keresztül). * Kiválasztottam a Sör lapot. * Ellenőriztem, hogy a játékos életereje 1-re állt vissza, és nem esett-e ki a játékból. A log üzeneteket is figyeltem. * Megismételtem a szcenáriót, de a dialógusban a passzolást választottam (nem játszottam ki Sört), és ellenőriztem, hogy a játékos kiesik-e a játékból, és a *GameLogic.aPlayerRemoved()* lefut-e. |
| Eredmény: | * Az első esetben a játékos életereje visszaállt 1-ra és játékban maradt. * A második esetben a meghívódott a *GameLogic.aPlayerRemoved()* és kiesett a játékos |

3. táblázat BaseModel.receiveDamage() ellenőrzése

|  |  |
| --- | --- |
| Bot Döntési Logika (*Bot.setYetToPlay()* - Bang! szűrés) ellenőrzése: | |
| Cél: | Verifikálni, hogy a bot csak akkor tervezi be a Bang! lap kijátszását, ha van lőtávolon belül célpontja, és még nem lőtt az adott körben (kivéve Volcanic esetén). |
| Manuális lépések: | * Olyan helyzetet teremtettem, ahol egy bot játékosnak Bang! lap van a kezében, de nincs Volcanic fegyvere. * Biztosítottam, hogy a többi játékos a bot alap lőtávolságán kívül legyen (pl. Musztánggal vagy nagy távolságra). * Elindítottam a bot körét, és console-ra iratással ellenőriztem a *setYetToPlay()* metódus végén a *cardsYetToPlay()* lista tartalmát. * Meggyőződtem róla, hogy a Bang! lap nem került be a listába a célpont hiánya miatt. * Módosítottam a helyzetet, hogy legyen célpont lőtávolon belül, és újra ellenőriztem, hogy a Bang! lap most már bekerül-e a listába. |
| Eredmény: | * Az első esetben a Bang! lap nem került be a listába. * A második esetben a Bang! lap bekerült a listába. |

4. táblázat Bot.setYetToPlay() ellenőrzése

### Integrációs tesztek

A projekt során nem készültek automatizált integrációs tesztek. Az egyes szoftverkomponensek – a Felhasználói Felület (UI), az Üzleti Logika (BL) és a Modell rétegek, illetve azokon belüli osztályok – közötti együttműködésének és adatcseréjének helyességét kizárólag manuális módszerekkel ellenőriztem, a funkcionális tesztelés szerves részeként.

A manuális integrációs tesztelés célja annak biztosítása volt, hogy az adatok és a vezérlés megfelelően áramoljon a rendszer különböző részei között. Ezt úgy értem el, hogy a teljes alkalmazást futtatva komplex, több komponenst érintő felhasználói forgatókönyveket játszottam végig. Egy-egy ilyen forgatókönyv során figyeltem az alábbiakat:

* A felhasználói felületen (UI) indított akció (pl. gombnyomás, kiválasztás) eljut-e a megfelelő üzleti logikai (BL) komponenshez (*GameLogic*).
* Az üzleti logika helyesen használja-e a model réteg objektumait (*BaseModel*, *Card*), és lekérdezi/módosítja-e a központi játékállapotot (*GameInstance*).
* Az állapotváltozások helyesen jelennek-e meg a felhasználói felületen (*BangGameUI*) a *updateUI()* metódus hívása után.
* A *GameLogic* által a UI felé indított visszahívások (pl. felhasználói input kérése dialógusablakon keresztül, üzenetek logolása) megfelelően működnek-e.

#### Példák a manuálisan tesztelt integrációs forgatókönyvekre:

1. Játék Indítási Folyamat: A *PlayerSetupFrame*-ben kiválasztott karakterek, szerepkörök és bot/ember beállítások hatását ellenőriztem. Végig követtem, hogy az Új játék indítása gomb megnyomása után a *GameLogic.startGame* helyesen hívja-e meg a *GameInstance.initializePlayers()*-t a megfelelő paraméterekkel, létrejönnek-e a *BaseModel* példányok a kívánt állapottal a *GameInstance*-ben, és végül a *BangGameUI* helyesen jeleníti-e meg a kezdőállapotot (megfelelő számú játékos, kiosztott lapok stb.).
2. Akciókártya Kijátszása (pl. Pánik): Szimuláltam egy Pánik kártya kijátszását. Ellenőriztem a teljes láncot: kártya kiválasztása a *BangGameUI*-n -> célpont kiválasztása -> *Kiválasztott kártya kijátszása* gomb -> *GameLogic.cardAction*() hívás -> *BaseModel.playDualTargetCard()* hívás a játékoson -> a Pánik kártya logikájának lefutása -> *BaseModel.panicAction()* hívás -> *GameLogic.chooseCard()* hívás a célpont lapjainak kiválasztására -> UI dialógus megjelenése -> felhasználói választás -> a kártya áthelyeződése a célponttól a játékoshoz (állapotváltozás a *BaseModel* objektumokban) -> *BangGameUI.updateUI*() hívás -> a változás vizuális megjelenítése a felületen.

Az integráció helyességét ezekben az esetekben a felhasználói felületen látható állapotváltozások vizuális ellenőrzésével, a log üzenetek (*logTextArea*) figyelésével és annak biztosításával értékeltem ki, hogy a megfigyelt viselkedés megfelelt a Bang! játékszabályainak és a várt működésnek.

### Acceptance tesztek

Az elfogadási tesztelés célja annak végső ellenőrzése volt, hogy a kifejlesztett Bang! játék alkalmazás megfelel-e a projekt célkitűzéseinek és a dokumentáció korábbi szakaszában (lásd 3.2. Követelményelemzés fejezet) részletezett funkcionális és nem-funkcionális követelményeknek. Ez a validációs fázis kizárólag manuális tesztelési módszerekkel történt.

A tesztelés elsődleges módszere teljes játékforgatókönyvek végig játszása volt. Több alkalommal indítottam új játékot különböző játékosszámokkal (pl. 4, 5, 7 fővel), ahol a saját (emberi) játékosommal különböző szerepköröket (Sheriff, Bandita, Renegát, Sheriffhelyettes) és karaktereket próbáltam ki a mesterséges intelligencia által irányított botok ellen. Ezek a teljes játékmenetet lefedő tesztek az alábbiakra fókuszáltak:

* A játék sikeres elindítása és a beállítások (*PlayerSetupFrame*) helyes alkalmazása.
* A körök logikus váltakozása, a kör eleji események (Dinamit, Börtön) kezelése.
* A kártyahúzás, a különböző típusú kártyák kijátszása (támadó, védekező, felszerelés, speciális akciók) és eldobása a szabályoknak megfelelően.
* A botok képesek-e önállóan végrehajtani a körüket (lapot húzni, felszerelést lerakni, akciókat végrehajtani).
* A játék végállapotának helyes felismerése: a különböző szerepkörökhöz tartozó győzelmi feltételek teljesülésének ellenőrzése, és annak verifikálása, hogy a játék a megfelelő győztest hirdeti ki a végén (*GameOverDialog*).
* A grafikus felhasználói felület (*BangGameUI*) folyamatos használata, az információk (életerő, kézben/asztalon lévő lapok, ellenfelek adatai, játéknapló) helyes és érthető megjelenítésének, valamint az interakciók (kártya-, célpontválasztás) működésének ellenőrzése.

Ezeken a forgatókönyveken keresztül expliciten ellenőriztem a funkcionális követelmények teljesülését. A nem-funkcionális követelményeket is manuálisan értékeltem:

* Platformfüggetlenség: Az alkalmazás Java 21+ alapú; a tesztelés elsősorban Windows környezetben történt.
* Egyszerű telepíthetőség: Az alkalmazás közvetlenül futtatható volt az IntelliJ IDEA fejlesztőkörnyezetből.
* Felhasználóbarát felület: A játék során szubjektíven értékeltem a GUI átláthatóságát és kezelhetőségét, és felhasználóbaráttá nyílvánítottam.
* Stabilitás: A tesztelés során figyeltem az esetleges fagyásokat vagy váratlan hibákat, különösen a komplexebb interakciók vagy potenciális szélsőséges esetek (pl. pakli kiürülése) során.
* Bővíthetőség: Az objektumorientált struktúra alapján koncepcionálisan értékeltem, hogy az új kártyák vagy karakterek hozzáadása mennyire tűnik egyszerűnek.

A manuális elfogadási teszteket én, a fejlesztő végeztem. Eredmény: A végigjátszott forgatókönyvek és a követelményekkel való összevetés alapján az alkalmazás a tesztelés időpontjában megfelelt a projekt keretében definiált alapvető célkitűzéseknek és a specifikált követelményeknek, biztosítva a Bang! játék egyjátékos, botok elleni verziójának működőképességét.

# Összefoglaló

## Konklúzió

A szakdolgozat célja a népszerű Bang! kártyajáték egyjátékos, asztali számítógépes változatának megvalósítása volt Java programozási nyelven, a Swing grafikus könyvtár felhasználásával. A projekt sikeresen elérte ezt a célt: egy önállóan futtatható alkalmazás készült, amely hűen követi az eredeti játék alapvető szabályait, karakterképességeit és szerepköreit, lehetővé téve a felhasználó számára, hogy mesterséges intelligencia által vezérelt botok ellen játsszon.

A fejlesztés során központi szerepet kapott a jól strukturált, háromrétegű architektúra (Modell-BL-UI) kialakítása, amely elválasztja az adatkezelést, az üzleti logikát és a felhasználói felületet. Az objektumorientált tervezési elvek (pl. öröklődés a kártya- és karakterhierarchiákban, kompozíció a játékosok attribútumainál) és tervezési minták (pl. Singleton a *GameInstance* esetében) alkalmazása elősegítette a kód modularitását és karbantarthatóságát. A megvalósítás Java 21 nyelven, IntelliJ IDEA fejlesztőkörnyezetben történt, külső függőségek vagy build eszközök (Maven/Gradle) nélkül. A szoftver működőképességét és a követelményeknek való megfelelését alapos, manuális funkcionális, integrációs és elfogadási teszteléssel ellenőriztem, amely során a játék különböző forgatókönyveit és funkcióit vizsgáltam. Az eredmény egy stabil, az alapvető Bang! élményt nyújtó egyjátékos alkalmazás lett.

## Továbbfejlesztési lehetőségek

Bár a projekt elérte a kitűzött célt, számos irányban továbbfejleszthető a jövőben:

* Többjátékos Mód (Multiplayer): A legnyilvánvalóbb és legnagyobb bővítési lehetőség a hálózati többjátékos mód implementálása, amely lehetővé tenné, hogy emberi játékosok játszhassanak egymás ellen interneten keresztül.
* Fejlettebb Mesterséges Intelligencia (AI): A jelenlegi Bot logika egyszerű szabályokon alapul. Jelentős javulást lehetne elérni komplexebb AI fejlesztésével, amely figyelembe veszi a szerepköröket (pl. Sheriffhelyettes védi a Sheriffet), jobban modellezi az ellenfeleket, taktikusan használja a lapokat, és esetleg különböző nehézségi szinteket kínál.
* Mentés és Betöltés Befejezése: A dokumentáció említi, hogy ez előkészített, de nem teljes. A játékállapot mentésének és későbbi betöltésének teljes körű megvalósítása növelné a felhasználói kényelmet.
* Felhasználói Felület (UI) Fejlesztése:
  + Vizuális javítások: Modernebb grafikai elemek, animációk bevezetése a játékélmény fokozására.
  + Reszponzivitás: Nagyobb felbontások és ablak átméretezés jobb kezelése.
  + Esetleg áttérés modernebb UI keretrendszerre (pl. JavaFX), bár ez jelentős átalakítást igényelne.
* Játéktartalom Bővítése: Az alapjáték kártyáin és karakterein túl további, hivatalos Bang! kiegészítők (pl. Dodge City, High Noon) lapjainak, karaktereinek és mechanikáinak implementálása. Az objektumorientált, bővíthető tervezés ezt támogatja.

Ezek a fejlesztési irányok tovább növelhetnék az alkalmazás értékét, funkcionalitását és felhasználói bázisát.

# Irodalomjegyzék

1. <https://tarsasjatekok.com/files/common/b/ba/ba1/ba1b4f22bae5e6ef1e07d0354f040dbe/bang-jatekszabaly.pdf> / Utolsó megtekintés dátuma: 2025. 05. 01.

1. [https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/package-summary.html /](https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/package-summary.html%20/) Utolsó megtekintés dátuma: 2025. 05. 01.

1. Néhány karakter speciális képessége ezt felülírja. Lásd: 2.2.4 Karakterek [↑](#footnote-ref-2)
2. Utolsó megtekintés dátuma: 2025. 05. 01. [↑](#footnote-ref-3)
3. Utolsó megtekintés dátuma: 2025. 05. 01. [↑](#footnote-ref-4)
4. Kivéve, ha a karakter speciális képessége ezt felülírja. Lásd: 2.2.4 Karakterek [↑](#footnote-ref-5)
5. Lásd: 2.2.1 A lehetséges szerepkörök, céljaik és győzelmi feltételeik: [↑](#footnote-ref-6)