Eötvös Loránd Tudományegyetem

Informatikai Kar

Média- és Oktatásinformatika Tanszék

Labirintus verseny mesterséges intelligenciával

Témavezető:

Bende Imre

tanársegéd, MA

Szerző:

Lukács Dávid István

Programtervező informatikus BSc.

Budapest, 2022

Tartalomjegyzék

1. Bevezetés

A szakdolgozatom témájának kiválasztása, ezáltal alapvető koncepciója az volt, hogy nem tartottam túl valóságos, gyakran előforduló esetnek azt, hogy labirintusba kerüljek valaha, de mindig úgy éreztem, hogy kicsit tartok, félek a megoldandó problémától, hogy nem lennék képes csak a józan eszemre és szerencsémre hivatkozva kijutni. Ez mellé jött az a tulajdonságom, hogy sokkal könnyebben szokott egy új dolog megtanulása, elsajátítása menni akkor, ha valaki azt a cselekvést éppen előttem végzi, és mindig érdeklődéssel töltött el az, ahogy próbáltam rájönni a cselekvés közben hozott döntések miértjére. Ezt a gondolatot és fejlődési vágyat ragadtam meg. Tehát a programom szeretné úgy szórakoztatni a felhasználóját, hogy gondolkodásra, gyorsaságra, döntéshozatalra készteti, illetve ad is neki egy példát, akitől akár tanulhat is, hiszen kijutása során egy ellenfélt állít vele szembe, aki közben mesterséges intelligenciát, gráfkereső és labirintus generáló algoritmusokat használ ellene. Azonban ezt még kevésnek tartottam, szerettem volna azt, hogyha a felhasználók egymással is interakcióba léphetnek valamilyen módon, de azt éreztem, hogy túl sok terhelést jelentene minden éppen online felhasználót megjeleníteni a labirintusban. A felhasználói élmény, és a szociális jelleg növelése érdekében egy közös adatbázisban láttam a megoldást, ahova a felhasználók kedvük szerint készíthetnek labirintusokat, ezáltal adva egy végtelenségig szórakoztató, el nem laposodó jelleget a játéknak, hisz bármely pillanatban bővülhet az adatbázis néhány új labirintussal, amivel aztán a többi felhasználó, vagy akár saját maga is küzdhet. A szociális jelleg egyfajta kompetitív tulajdonságot is hoz magával, ugyanis a játék bár elméletileg a végtelenségig mehet, a közben érkező folyamatos nehezítések, amely leginkább a mesterséges intelligencia fokozatos gyorsítását jelenti, egy idő után lehetetlenné teszik a túl jó pontszám elérését. Kicsit elvesz a kompetitív jellegből viszont a sok véletlenszerűség, hogy véletlenszerű pályát kapunk, ami abból kiindulva, hogy a labirintus kezdő és végpontja között nincs meghatározva sem minimális, sem átlagos távolság, végül változó nehézséget jelent a kijutást illetően. A nyelvek, környezetek választásánál az volt a szempontom, hogy az itteni tanulmányaim során megismert technológiákat, algoritmusokat vegyítsem olyan újdonságokkal, amik a témából kifolyólag érdekelnek engem.

2. Felhasználói dokumentáció

Az alapvető probléma, hogy nem volt még szerencsém olyan játékszoftverhez, amelyben volt lehetőség labirintusok létrehozására, illetve a kijutáson kívül valamilyen extra nehézség is került a játékmenetbe. Jelen esetben, a program futása során egyfajta mesterséges intelligenciát személyesít meg, ami az egyik hatékonyabb labirintus útkereső algoritmust használja, ezzel próbálkozik az emberi versenytársa szerencséje, gyorsasága és tudása ellen. A probléma másik fele, hogy bár mindenképp szerettem volna a játékot még ennél is izgalmasabbá tenni azzal, hogy a felhasználókat az egymással történő versengésre, akadályoztatásra buzdítom, nem szerettem volna kiszúrni azon játékosokkal, akik infrastruktúrájuk, vagy pillanatnyi helyzetük miatt nem rendelkeznek aktív internetkapcsolattal. Így a játék rendelkezik egy offline móddal is, ami az eddigi tapasztalataim miatt kicsit nehezebbre is sikerült, mint az online mód, tekintettel arra, hogy az online módban a játékosok hajlamosak egyszerűbb labirintust csinálni, mint egy, a bonyolultságra összpontosító algoritmus.

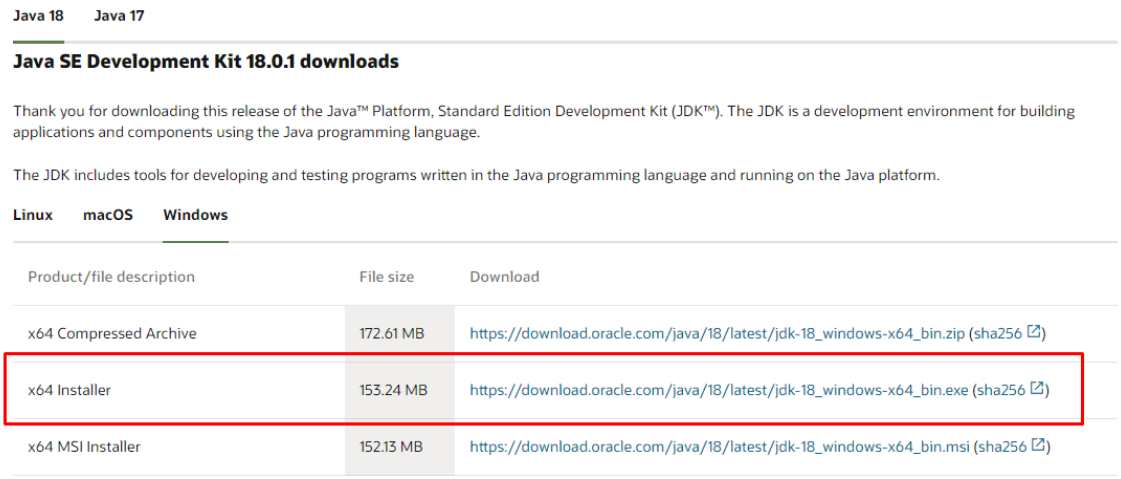
2.1 A program követelményei hardveres és szoftveres téren

A program futtatásához egy Windows 10 rendszerű, 64-bites processzoros számítógép szükséges. A pontos, minimum követelmények [1] :

- Processzor: Legalább 1 gigahertz (GHz) órajelű processzor.
- RAM: 2 gigabyte (GB).
- Tárhely: Az operációs rendszeren, és a futtatókörnyezeten felül 2 GB szabad tárhely szükséges.
- Grafikus kártya: DirectX 9 kompatibilis grafikus kártya, legalább WDDM 1.0 illesztőprogrammal.
- Kijelző: Legalább 1280 x 720 pixel felbontású
- Operációs rendszer: A játék Windows és Ubuntu alatt is tud futni, a megfelelő játékélményhez a Windows 10/11 operációs rendszer ajánlott.
- Szoftver: A szoftver futtatásához egy legalább Java 17 Runtime verziót futtatni képes Java verzió szükséges.

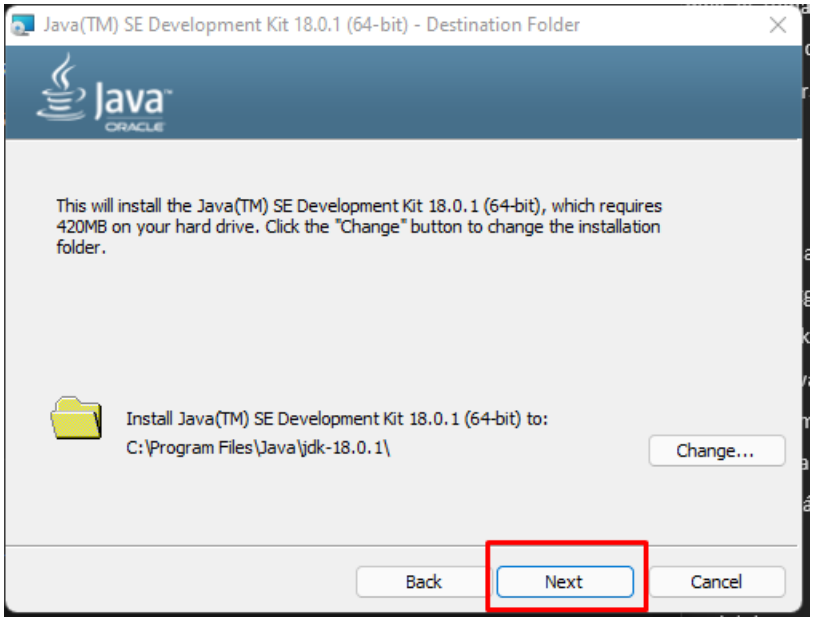
2.2 A Java telepítése

A játékhoz szükséges java verzió (Windows és Linux esetén) a következő weboldalról érhető el: [2]. Windows esetén a weblapot megnyitva a Windows fülön az „X64 Installer” sorban szereplő linken lehet letölteni a telepítő fájlt.



1. ábra : a letöltési link lokációja a weblapon. [2]

Példaképpen, a Java 18 telepítője 420 megabyte (MB) helyet igényel a merevlemezünkön. A telepítő megnyitásánál, ha szeretnénk eltérni az alapvető telepítési helytől, válasszuk ki a számunkra szimpatikus útvonalat, majd a tovább gombbal fel is települ a futtatási környezet.

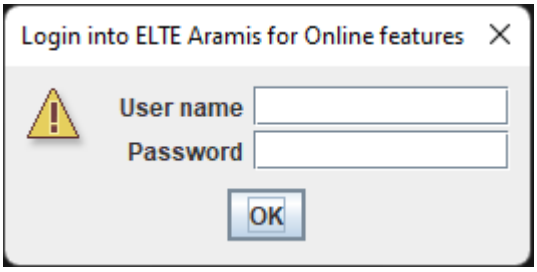


2. ábra : a Java futtatási környezet telepítő ablaka.

2.3 A program futtatása

A Java futtatási környezet feltelepülése után már csak a játékszoftver futtatása maradt hátra. Ezt a csatolmányban mellékelt tömörített állomány kicsomagolásával, majd a kicsomagolt verzióban, a főkönyvtárban elhelyezett „Labyrinth Adventure.jar” fájl megnyitásával érhetjük el.

A futtatás során (több képernyős rendszerek esetén az operációs rendszerünkben megadott alapértelmezett képernyőn) megnyílik egy kis ablak:



3. ábra: Bejelentkező képernyő a program megnyitásakor

Ebben az ablakban szükséges az ELTE Aramis adatbázisához rendelkezésünkre álló felhasználónév-jelszó párt beírnunk, amivel később elérjük az online funkciókat. Amennyiben ilyennel nem rendelkezünk, a következő felhasználók valamelyikével szintén be tudunk lépni:

- Adminisztrátori felhasználó:
    - felhasználónév: IDU27K
    - jelszó: almafa
- Felhasználói jogokkal rendelkező felhasználók:
    - felhasználónév: HF3LBG
      jelszó: HF3LBG
    - felhasználónév: RNYR2F
      jelszó: RNYR2F
    - felhasználónév: GF4SDH
      jelszó: GF4SDH

Ha nem rendelkezünk, vagy nem szeretnénk az imént megadottfelhasználók valamelyikét használni, a bemeneti mezőket hagyhatjuk üresen is, ezzel viszont csak az offline funkciók maradnak használhatóak. Ha ezt a döntést meghoztuk, az „OK” feliratú gombra kattintva a főmenübe jutunk.

2.4 Új felhasználó regisztrálása

Ha szeretnénk egy új felhasználót regisztrálni, azt úgy tehetjük meg, ha egy e-mailt írunk a szerver üzemeltetőjének és adminisztrátorának, a nikovits@inf.elte.hu címre. A levélben szerepeljen, hogy ezzel a játékkal szeretnénk játszani, és az is, hogy milyen felhasználónevet szeretnénk. Jelszót nem szükséges az e-mailbe írni, egy generált jelszót fog adni az adminisztrátor a válaszlevélben. A kért felhasználó létrehozása akár több ideig is eltarthat, addig is használhatjuk a feljebb megadottakat.

2.5 A játékszoftver funkciói

4. ábra: A főmenü kinézete, ha nem elérhetőek az online funkciók

Sikertelen bejelentkezés, vagy szándékosan üresen hagyott bejelentkezési adatok esetén azt tapasztalhatjuk, hogy csak a „Play Offline” és „Exit Game” feliratú gombok elérhetőek. Amennyiben ez nem az elvárt eredmény, ellenőrizzük internetkapcsolatunkat és a program újra futtatásával, majd gondosan ellenőrzött

bejelentkezési adatok megadásával próbálkozzunk ismét. Ha viszont a bejelentkezés sikeres, a következő főmenü jelenik meg:

5. ábra: A főmenü kinézete, sikeres bejelentkezés, ezáltal elérhető online funkciók mellett

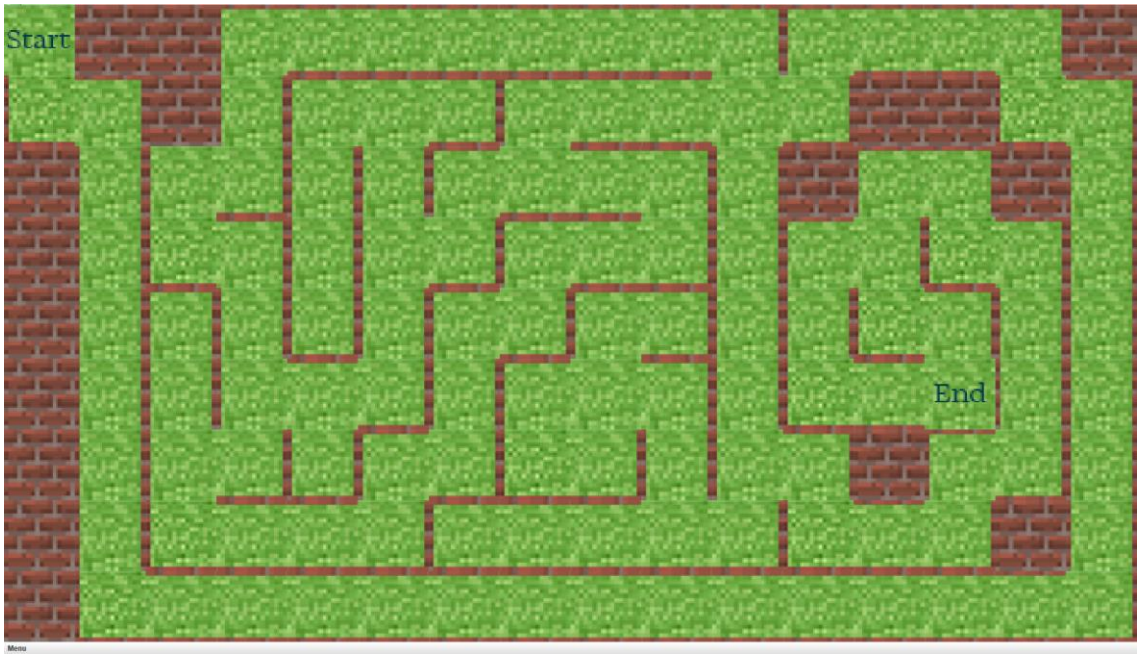
2.5.1 A pályakészítő funkció

A főmenüből az elérhető menüpontok, és funkciójuk balról jobbra: „Map Builder” (csak online elérhető): Ezen menüpont alatt van lehetőségünk a pályakészítő funkciót elérni.

A pályakészítőben egy labirintust úgy tudunk megrajzolni, először az egérmutatót az általunk kiválasztott kezdőcellára helyezzük, majd a bal egérgombot lenyomva, és a kurzort valamilyen irányba húzva elkezdjük kialakítani a labirintusunkat. A labirintusunk kezdőhelye egy „Start” felirattal lesz jelölve, a labirintusból kivezető utat pedig teljesen a saját elképzelésünkre bízhatjuk. A végpontot bármikor változtathatjuk úgy, hogy a jobb egérgombbal a megfelelő cellára kattintunk. Ha nem tettünk ilyet, és úgy próbáljuk elmenteni a labirintust, akkor automatikusan a legutolsó olyan cellát jelöli ki, amelynél utat csináltunk.

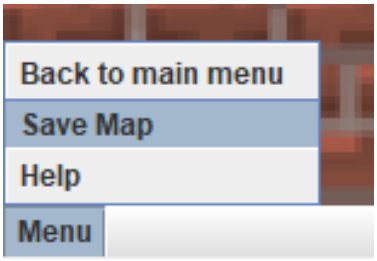
Arra viszont figyelnünk kell, hogy ha a labirintus kezdőpontjából a végpontjába nem vezet bejárható út (például úgy, hogy a labirintusunk végpontja és kezdőpontja közti részt minden esetben legalább egy fal elválasztja), akkor egy későbbi pontban az

elmentést, az adatbázisba feltöltést a program nem fogja engedni. Egy példa egy elkészített labirintusról:

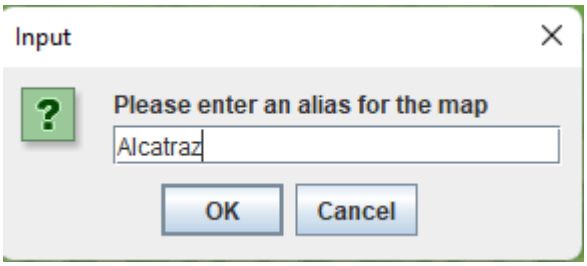


6. ábra: Egy mentésre kész labirintus

Ha elégedettek vagyunk az elkészített labirintussal, a képernyő bal alsó sarkában látható „Menu” feliratú gombra kattintva, majd a „Save Map” feliratú gombra kattintva a játék arra fog kérni bennünket, adjunk egy becenevet a labirintusunknak, amiről később majd felismerjük. Természetesen, ha mégsem szeretnénk elmenteni a labirintusunkat, a „Back to main menu” gombot megnyomva ismét a főmenüben találjuk magunkat.

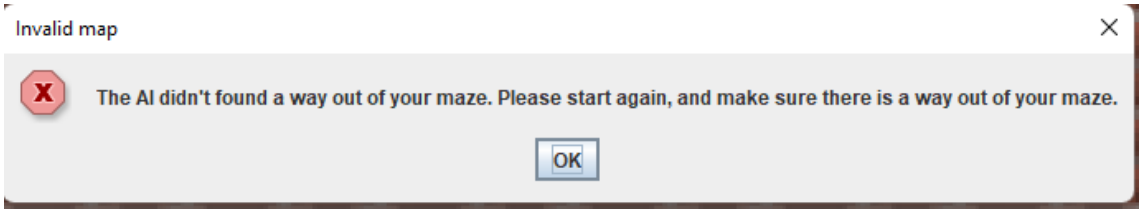


7. ábra: Képernyő bal alsó sarkában található "Menu" gomb, amiből aztán a "Save Map" menüpont lesz elérhető.



8. ábra: Becenév adása a labirintusnak

Ha az OK gombra kattintottunk, a szoftver a háttérben ellenőrzi, hogy ténylegesen van-e kiút a labirintusból, amit készítettünk, és ha talált, azt úgy jelzi, hogy ki lép az aktuális ablakból. Ha viszont olyan labirintust szeretnénk elmenteni, ami nem rendelkezik kiúttal, a következő hibaüzenet fogad minket:



9. ábra: Hibaüzenet, ha olyan labirintust készítenénk, amiből nincs kiút

Ha viszont nem kaptunk hibaüzenetet, és semmilyen más internetkapcsolati probléma nem avatkozott közbe, a „My Maps” menüpont alatt megtekinthető az új labirintusunk, de ezt a menüpontot még részletesen bemutatok.

2.5.2 Online játék

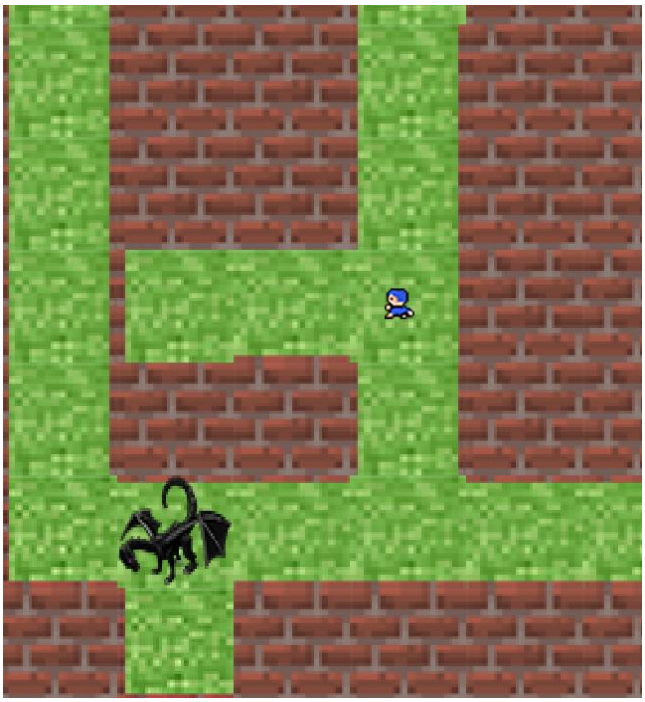
Ez a funkció csak aktív internetkapcsolat mellett elérhető. Ha a „Play Online” marad. Így folytatódik ez mindaddig, amíg vagy mi, vagy a sárkány nem talál ki a labirintusból.

Ez a játékmód annyiban jelent többet az offline, azaz internetkapcsolat nélkül is játszható módnál, hogy ebben az esetben csak olyan labirintusok jönnek elő, amelyeket egy valós felhasználó készített. Ha esetleg ismerős számunkra az egyik labirintus, érdemes megnézni a lenti sávban a készítő nevét, hisz könnyen előfordulhat, hogy pont mi készítettük.

menüpontot választjuk, egy nagyrészt felfedezetlen labirintusban találjuk magunkat egy sárkánnyal. A bal alsó sarokban látjuk a készítő felhasználó nevét, illetve kis várakozás esetén azt is megfigyelhetjük, hogy a sárkány tétovázás nélkül elkezdte a maga kis útját

járni. Ebben a játékban ő lesz az ellenfelünk, aki folyamatosan azon lesz, hogy az algoritmusból könnyebben kijusson, mint mi. Hogy ezt hogyan teszi pontosan, ha érdekel bennünket, bármikor követhetjük útján, és tanulhatjuk el tőle a taktikáját, viszont az egyáltalán nem biztos, hogy jó döntés azokat a lépéseket követni, amiket a sárkány meghoz.

De ha elindulunk például saját utunkon, amit a billentyűzeten található négy nyíllal jelzett billentyű segítségével tehetünk meg, elkezdhetjük bejárni a labirintust, és ahogy odaérünk egy eddig még felfedezetlen részhez, úgy minden irányban felfedeződnek az ezelőtt még számunkra ismeretlen folyosók. Ezek később sem felejtődnek el, tehát ha már tovább haladunk, és egy másik folyosóba megyünk, ahol a falak rendszere miatt egyébként nem látnánk át az első, eredeti folyosóra, attól még az ugyanúgy látható



10. ábra: Online játékmenet a sárkánnyal és a karakterrel

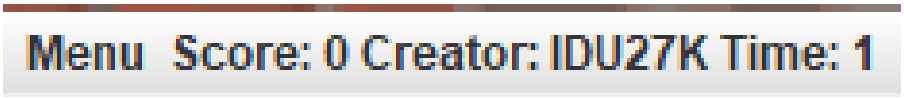
Abban az esetben, hogyha megtaláltuk az „End” feliratú mezőt a labirintusban, amely a kiutat szimbolizálja, érdemes minél hamarabb oda sietnünk, és várnunk a következő pálya betöltését.

Bizonyára feltűnhetett az első lejátszott alkalom után, hogy a sárkány jelentősen lassabban mozgott, mint amilyen gyorsan a kis kék karakterünk mozogni képes. Itt jön a képbe a pontszám jelentősége. A pontszám egyenlő a már, sárkánynál gyorsabban

megoldott labirintusok számával. Így kezdeti állapotban ez a szám 0, emiatt a sárkány az alapvető, leglassabb sebességgel fog a kiút felé haladni.

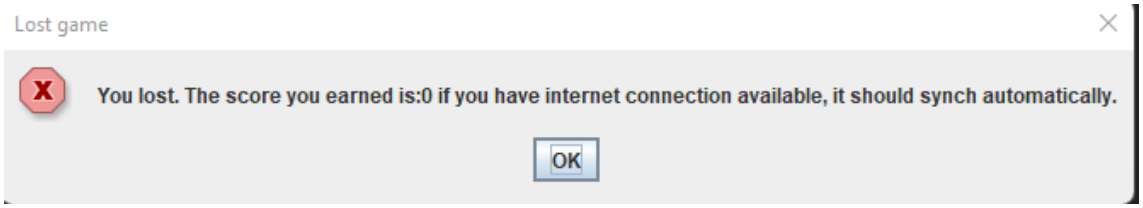
Így hogyha kijutottunk egy labirintusból, a feladat végül is nem változna, egy részt kivéve: A sárkány a pontszámunk növekedésével egyre jobban, és jobban fog próbálkozni, hogy emberi ellenfelén felülkerekedjen. Ezt szimplán úgy teszi, hogy labirintusonként növekszik egy picit a sebessége.

A pontszámunk, az adott labirintusban eltöltött másodpercek száma és a labirintus készítője mindig megjelenik az alsó menüsávban, így egy hosszabb, akár 10 labirintusos játékmenet után is mindig tisztában lehetünk azzal, hogy mennyi szükséges az éppen aktuális rekordunk megdöntéséhez.



11. ábra: Menüsor, ahol látható a labirintus készítője, a pontszám és az idő

Ha azonban sem a szerencsénk, sem a bevetett tudásunk nem hozta meg a tőle várt győzelmet, egy felugró ablak jelzi, hogy a sárkány nálunk gyorsabban kijutott. Ekkor, amennyiben sikerült a rekordunkat megdönteni, az új pontszámunk a felhővel szinkronizálódik, majd a játékmenet végezetéül ismét a főmenüben találjuk magunkat.



12. ábra: A játék végét jelző üzenet

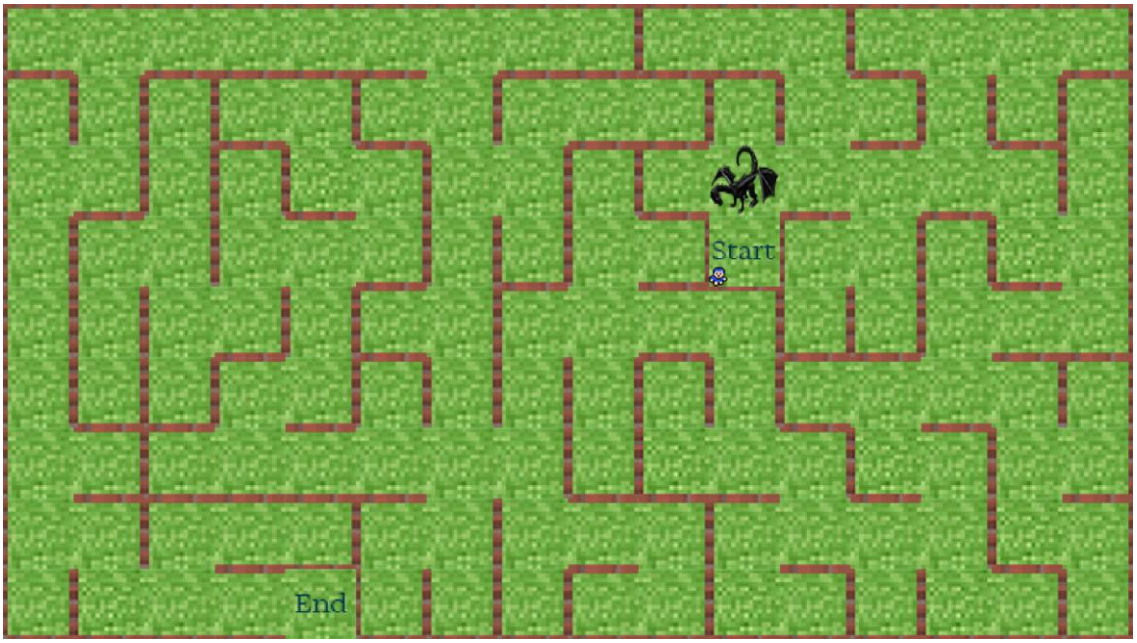
A játékot nehezíti, hogy tekintettel arra, hogy ez mégis csak egy kompetitív, szellemi jelenlétet igénylő játék, szünet tartására nincsen lehetőség. Ezzel elkerüljük az olyan „csalásokat”, ha például a szüneteltetett játék közben gondolkozna a játékos azon, hogy merre érdemes tovább indulnia, vagy tapasztalt játékosok esetében azt, hogy a lementett, lefényképezett labirintusok közül melyik az amelyikből aktuálisan ki kellene jutniuk. Ha viszont úgy érezzük, hogy a pályának csak nagyon lassan találnánk meg a kiútját, esetleg belefáradtunk az adott labirintus megoldásába, az alulról elérhető

menüből bármikor kérhetünk egy új labirintust, viszont ezzel feláldozzuk a jelenlegi pontszámunkat. Természetesen ugyanígy a „Back to main menu” gombbal a főmenübe is visszajuthatunk, a játék további funkcióinak megismeréséhez.

2.5.3 Offline játék

Ha azonban nem vagyunk abban a kényelmes helyzetben, hogy aktív internetkapcsolattal rendelkezünk, például vonaton, repülőn, hajón ülünk, akkor is tartogat számunkra ez a játékszoftver, még ha csak csökkentett funkcionalitással is.

A játékmenet nagyon hasonló lesz ebben az esetben is, mint hogyha internetkapcsolattal rendelkeznénk: Ebben az esetben is a sárkány lesz az ellenfelünk, aki most is gyorsul annak függvényében, hogy hány labirintusból jutottunk ki egyhuzamban gyorsabban, mint bináris ellenfelünk. A különbség a pályák nehézségében és jellegében lesz, mivel nem tudunk ebben az esetben az adatbázisból adatot beszerezni. A játékprogram egy beépített algoritmussal rendelkezik, amely képes komplex labirintusokat készíteni, és pontosan ez kerül használatba az offline játékmód esetén. Ez az algoritmus hajlamos nagyon véletlenszerű, emiatt akár nagyon könnyű vagy nagyon nehéz labirintusokat is generálni, így alapvetően offline üzemmódban átlagosan kicsit nehezebb a kijutás, ezáltal a pontszerzés is.

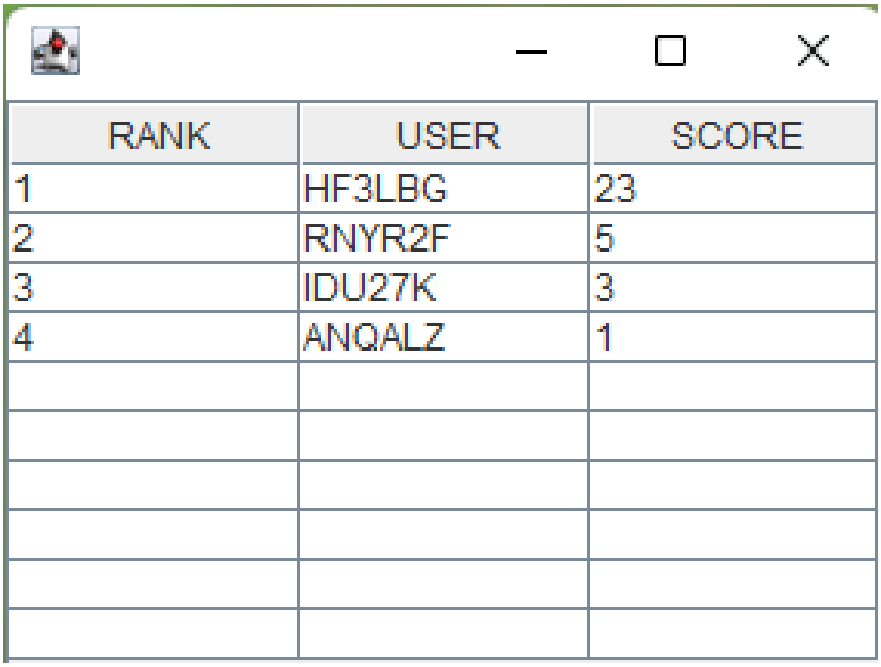


13. ábra: Egy algoritmus által generált labirintus

Természetesen a labirintus mérete a nehézség körülbelüli szabályozása érdekében mindig ugyanakkora, és az online, játékosok által pályák is lehetnek az elvártnál sokkal nagyobb erőfeszítést igénylőek. A játék viszont ebben a formájában ténylegesen gyakorlási célra szolgál, hiszen habár számolja az elért pontszámot a játék, ezt a játékmenet végén, azaz egy olyan esetben amikor a sárkány felülkerekedik rajtunk, nincs lehetőség a felhőbe menteni.

2.5.4 Toplista

Ahogy véget ér az első néhány lejátszott kör, egyre kíváncsibbak lehetünk arra, vajon hasonló érdeklődésű embertársaink hogyan teljesítettek az útvesztőkből való kijutásban. Ez a menüpont pontosan erre a kérdésre adja meg a választ. A menüben rákattintva egy kis ablakban nyílik meg a toplista, melyben az oszlopok szerint balról jobbra haladva egyenként a sorszámot, a hozzá tartozó felhasználót, és az általa elért legmagasabb pontszámát láthatjuk. Kilépni a jobb felső sarokban található „X” szimbólumra kattintással tudunk. Előfordulhat, hogy kis időbe telik, míg a frissen elért rekord eredményünk ide is bekerül, az is megeshet, hogy csak a következő játékindításnál lesz elérhető.



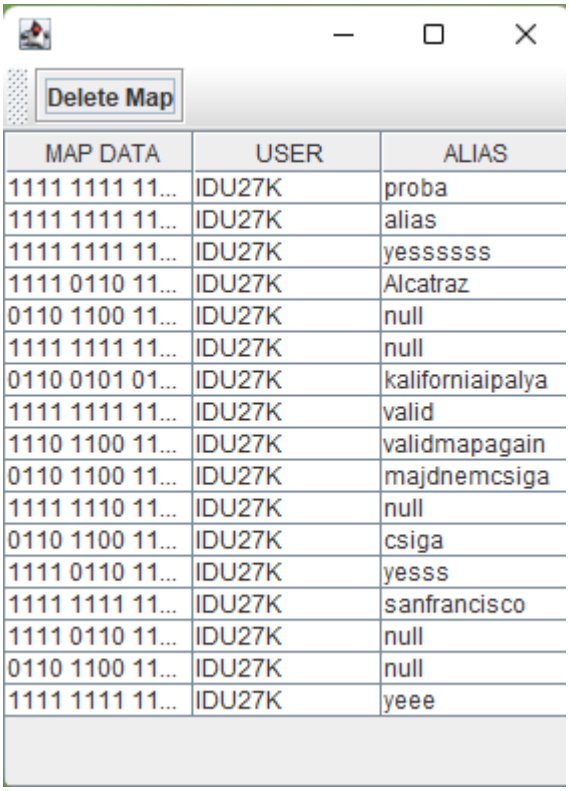
RANK	USER	SCORE
1	HF3LBG	23
2	RNYR2F	5
3	IDU27K	3
4	ANQALZ	1

14. ábra: A toplista

2.5.5 Saját labirintusok menüpontja

Ez a menüpont akkor lesz a felhasználó számára érdekes, hogyha a pályakészítő funkcióban már eljutott egy remekmű elkészítéséig. Azért, mert itt kilistázásra kerül az adott, éppen bejelentkező felhasználóhoz tartozó összes, az adatbázisban aktuálisan szereplő labirintus. Mivel itt is szükséges az adatbázissal adatot cserélni, hogy biztosan naprakész információt láthasson a játékos, elengedhetetlen az aktív internetkapcsolat.

Ha a menüpontra kattintunk tehát, egy, a toplistához nagyon hasonló táblázat fogad bennünket, benne az összes elkészített pályánkkal. A pályáknak látható a szöveges reprezentációja, illetve a felismerhetőség céljából az a becenév is, amit az elkészítésekor adtunk neki,



15. ábra egy felhasználó labirintusainak listája

Ebben a funkcióban van lehetőségünk visszavonni egy elkészített pályánkat. Lehet, hogy többet nem szeretnénk a készítők körébe tartozni, vagy szimplán túl könnyűnek, esetleg túl nehéznek ítéljük meg valamelyik elkészített pályát. Ha bármilyen itt leírt vagy nem leírt okból törölnénk, azt úgy tehetjük meg, hogy először kiválasztjuk azt a sort, amelyik a törlendő pálya adatait tartalmazza. A kijelölést kék színnel fogja a szoftver jelölni, majd ha biztosak vagyunk a döntésünkben, a sorok felett található „Delete Map” feliratra

kattintva törölhetjük az adott labirintust az adatbázisból. A sikeres törlést egy felugró ablakkal jelzi a szoftver. Előfordulhat, hogy kis idő eltelik mire a labirintusaink közül tényleg eltűnik a frissen kitörölt pálya. A legszélsőségesebb esetben itt is a következő futtatásig bezárólag garantált, hogy a pálya eltűnik a játszható labirintusok sorából.

2.5.6 Kilépés a játékból

Hogyha úgy döntöttünk, hogy elég volt a játékból, vagy abban az esetben, ha az internetkapcsolatunk elérhetősége változik, ez a funkció lehet a segítségünkre. A menüből utolsó gombként elérhető „Exit Game” megnyomásával, aktív internetkapcsolat esetén bontja a kapcsolatot a szerverrel a játék, amivel jelzi, hogy az általunk létrehozott új pályák, illetve esetlegesen megdöntött rekord egy ideig véglegesnek tekinthető. Ha offline állapotban vagyunk, de időközben lett internetkapcsolatunk, akkor egy gyors újraindítással, és bejelentkezéssel hamar elérhetővé tehetjük az online funkciókat.

2.6 Segítség funkciók a játékban

A pályakészítőben, illetve az online és offline játékmenetben az alul lévő Menu gombbal elérhető a „Help” alatt egy kis súgó, amivel az adott játékmenet, funkció világossá válik a kezdő, vagy az adott funkciót először használó játékosok számára.

3. Fejlesztői dokumentáció

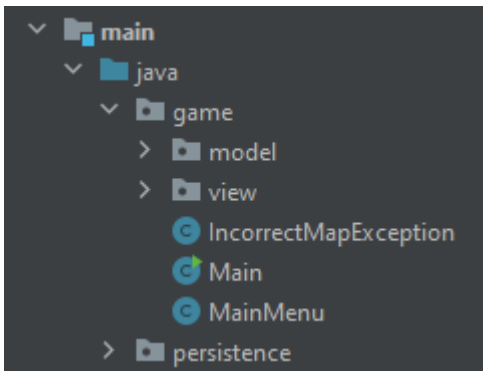
3.1 A program főbb technikai információi

Ez a program Java 17-es nyelvi szinten íródott, amely az úgynevezett Gradle projektépítő eszközt használja. A futtatáshoz, fejlesztéshez szükséges főbb függőségei:

- A Gradle projektépítő szoftver, amely néhány jelenlegi függőséget, és a jövőbeli függőségek tervezésében játszik jelentős szerepet. [3]
- Bármely 17-es, vagy újabb verziójú Java Development Kit (szintén elérhető a [1] weblapon).
- A betöltendő projekt egy JetBrains IntelliJ IDE (Integrated Development Environment, azaz integrált fejlesztői környezet [9]) projekt, amelyben minden be van konfigurálva a továbbfejlesztés, futtatás, tesztelés megkönnyítéséhez, így ezt a fejlesztői eszközt ajánlom.
  A program fontosabb, nem triviális függvényei Javadoc-kal vannak dokumentálva, és magyarázva. Elérhető itt: [10]
- Java Secure Channel 0.1.55-ös verzió, a porttovábbításhoz, és az ELTE Caesar szerveréhez való csatlakozáshoz. [11]
- Oracle Java Database Connector 11-es verzió, az ELTE Aramis Oracle SQL szerverhez történő csatlakozáshoz, az ott történő lekérdezések, frissítések elvégzéséhez szükséges. [12]
- JUnit 4.13.2 verziójú egységteszt-keretrendszer, ami a jelenlegi tesztkonfiguráció futtatásához szükséges.[13]
- Hamcrest 1.3 verziójú keretrendszer, amely a JUnit használatához szükséges.[14]

3.2 A program rétegjei

A program összessége, a programtervezési mintának megfelelően csomagokra van bontva, így is kihasználva az objektum elvű programozásnak ezen részéből eredő láthatósági szabályok előnyeit.



16. ábra: A program csomagjainak struktúrája

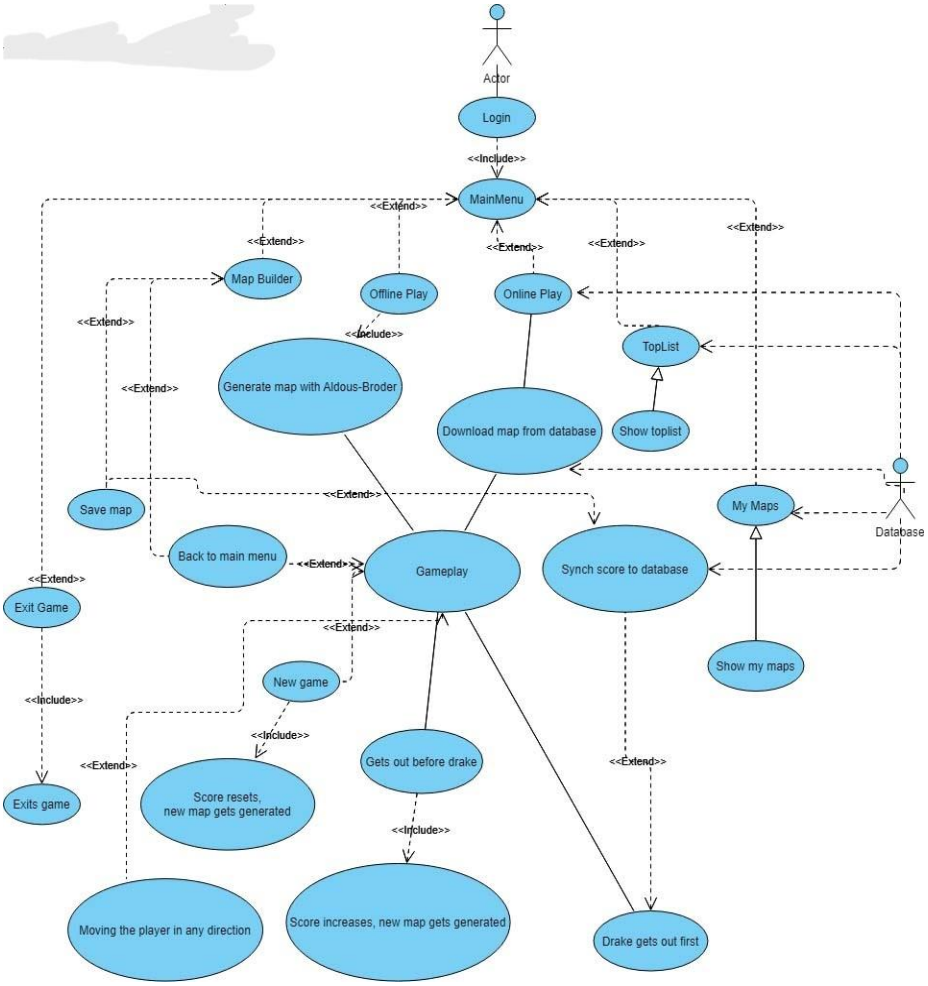
A program alapvetően a modell-nézet-vezérlő programtervezési mintát használja, így alapvetően a rétegek tekintetében igyekszik a felhasználói felületet, és az azon megjelenítendő, ábrázolandó adatot különválasztani. Ez azért fontos, hogy a felhasználói felület az adatkezelést ne befolyásolja, és az adatok átszervezhetők legyenek a felhasználói felület változtatása nélkül.

Tulajdonképpen esetünkben a vezérlőnek a főmenü tekinthető, hiszen ez az osztály hívja meg az osztályoknak az éppen használt funkcióhoz leginkább alkalmazkodó formáját. Ebben vannak azok a konstruktor, illetve metódushívások, amik aztán a komponensek további funkcionalitását elindítják, ezzel az egész programból egy összefüggő egészet alkotva.

A modell rétegben helyezkedik el minden olyan entitás vagy objektum, amelyet később majd a nézeti rétegben megjelenítünk. Ide tartoznak a Dragon, a *Cell*, a *Player*, és a *Labyrinth* osztályok. Ebben a csomagban van leimplementálva többek között a sárkány döntései mögött álló mesterséges intelligencia, a cellák esetében a falakhoz, illetve a cella labirintusban elfoglalt helyéhez tartozó adattagok és lekérdező, illetve beállító metódusai, vagy a játékoshoz tartozó pixel koordináták, illetve a mozgásának kezeléséhez szükséges egyéb változók.

A nézeti réteg lényegében az összes, főmenüből elérhető ablakot külön osztályban tartalmazza, egy olyan plusz kitétellel, hogy a játék komplexebb részein, például az offline vagy online játékban, a plusz izoláció érdekében további osztályok bevezetésére került sor. Az öröklődés lehető legjobb kihasználása érdekében létrehozásra került egy absztrakt osztály, illetve, hogy a játékban használt grafikus elemek egy helyen kerüljenek betöltésre és kezelésre, egy statikus osztály is.

3.3 A program használati eset diagramja

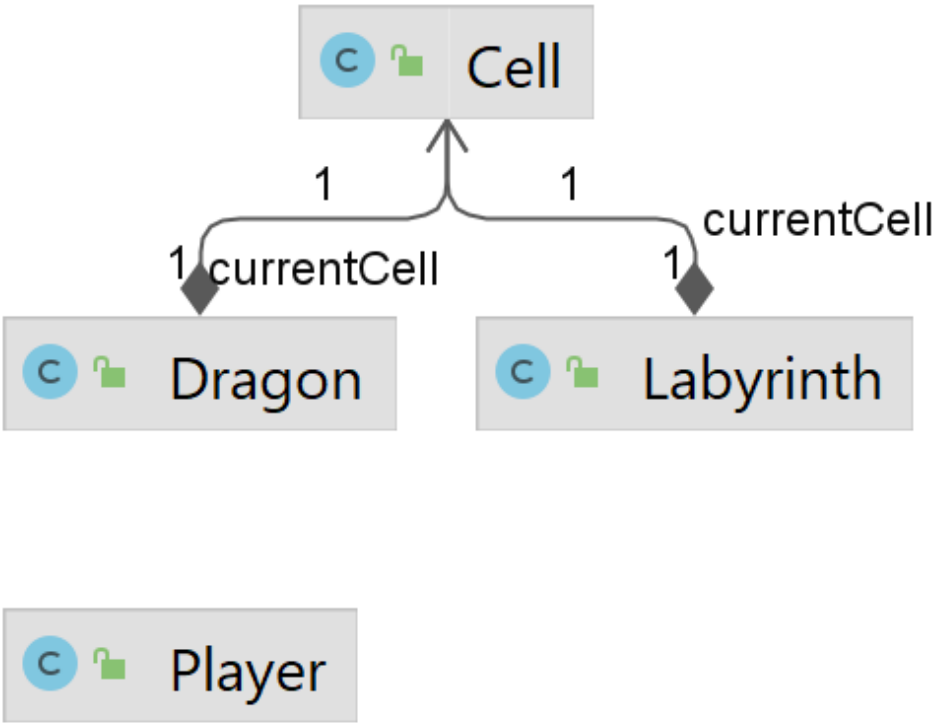


17. ábra A program használati eset diagramja

A használati eset diagram segítségével megérthető, hogy az egyes szereplők, esetünkben a játékos milyen típusú interakciókat tud végrehajtani, és annak a jelzésére, megjegyzésére is adódik lehetőség, hogy ha ehhez van valamilyen függőség, ami által elérhetővé válik, a speciális esetünkben néhány funkció esetén az adatbázis az ilyen. [6]

3.4 A program csomagjai

3.4.1 A modell csomag



18. ábra A modell csomag osztályainak UML-diagramja

A modell csomagban található minden olyan eleme a játéknak, mely később a nézet csomag által valamilyen módon megjelenítésre kerül. Ide tartozik a sárkány, aki az útkereső algoritmust használni fogja, a játékos, aki a sárkánnyal versenyezni szeretne a labirintusból történő kijutásban, a cella, ami az alap építőeleme a labirintusoknak, illetve végül a cellák gyűjteménye, a labirintus maga.

3.4.1.1 A cella osztály

19. ábra a Cella osztály UML diagramja

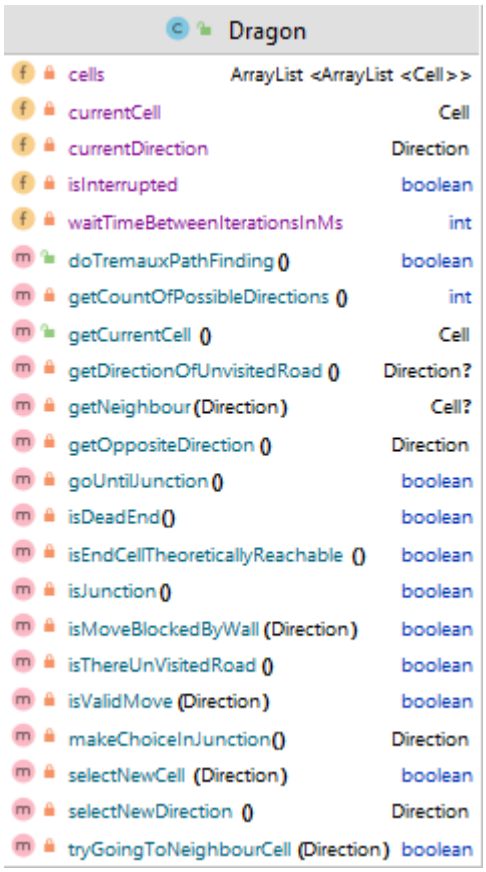
Ahogy a fenti ábrán is látható, a legfontosabb, legkomplexebb osztály a *Cell*, azaz a cellák megvalósítása. Ebben az osztályban a falak egyenként egy *boolean* típusú változóval állíthatóak be, illetve a cellában tároljuk azt is, hogy a játékmenet során látható-e a játékos számára, hogy a labirintuson belül milyen pozícióban van, illetve található néhány segédváltozó is a sárkány útkereső algoritmusához.

Mivel a cella esetében nagyon sok féle állapot lehetséges (például csak bal oldalon rendelkezik fallal, vagy alul-felül, vagy akár mind a négy oldalról blokkolt a bejutás az adott cellába) ezért a *selectImage* metódussal a cella a megfelelő paraméterű képet juttatja vissza a nézet csomagból érdeklődő osztályoknak.

A sárkány útkereső algoritmusához, a *hasBeenVisitedByDragon* és az *IsDeadEnd* nevű változók tartoznak, ezen értékeket a bejárás során változtatja, ezzel elősegítve, hogy ugyanazon útra ne menjen rá kettőnél többször.

Szintén itt van megoldva az *isVisibleForPlayer* boolean segédváltozóval az, hogy amikor a játék elkezdődik, a játékos számára nem láthatóak azon részei a labirintusnak, ahol még egyáltalán nem járt, ezzel kicsit realisztikusabbá és nehezebbé téve az útkeresést.

3.4.1.2 A sárkány osztálya, és Trémaux algoritmusa



20. ábra a sárkány UML diagramja

A sárkány személyesíti meg a mesterséges intelligenciát a játékszoftverben, így ebben az osztályban, az alapvető pozícióval kapcsolatos dolgokat kivéve leginkább a Trémaux útkereső algoritmus implementálása található meg. Az útkeresés, ezáltal a

felhasználóval való versenyzés a következő, egyszerű szabályok követésével magyarázható:

- Minden utat, ahol járunk, megjelölünk. A megjelölés módja bár esetünkben segédváltozó, a valóságban fontos az is, hogy egyértelműen el lehessen dönteni a jelölés alapján, hányszor jártunk arra.

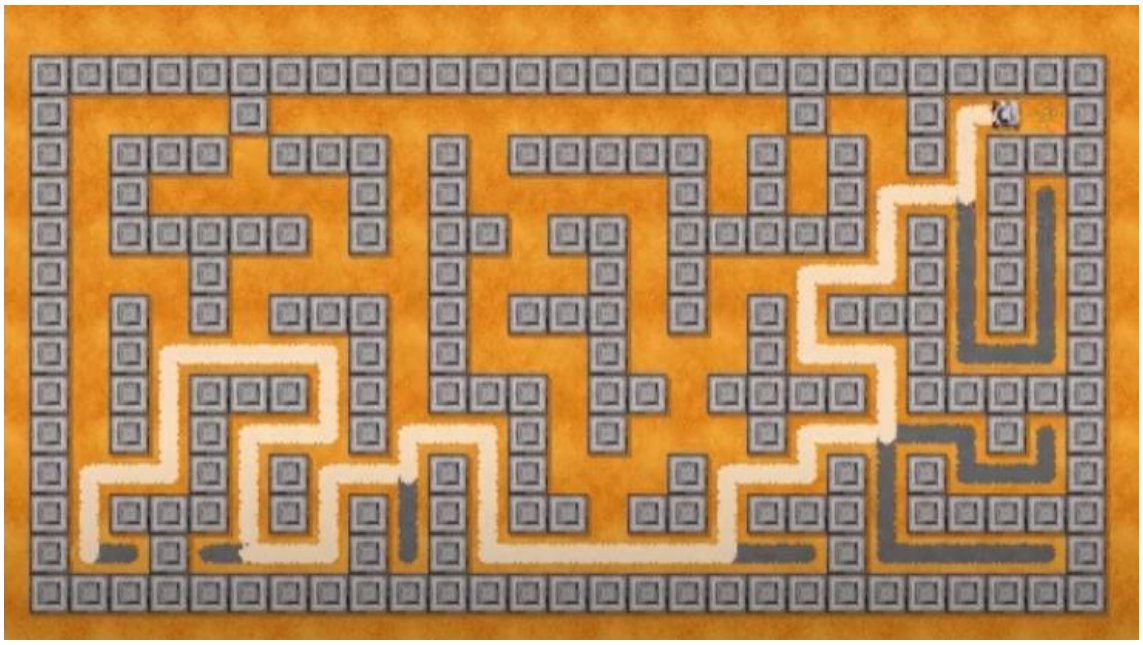
- Olyan útra, ami már kétszeres jelöléssel (vagy esetünkben az *isDeadEnd* változó *true* állapotával) rendelkezik, tilos harmadjára is lépni.

- Ha egy kereszteződésbe (olyan cella, amit legalább 3 különböző irányba lehet elhagyni) érünk, és a kereszteződésben nem jártunk még egyik úton sem (kivéve azt, amelyikből a kereszteződésbe érkeztünk), válasszunk véletlenszerűen egy utat a még nem felfedezettek közül, és haladjunk tovább.

- Ha olyan kereszteződésbe értünk, ahol már jártunk egyszer (tehát azon az úton kívül, amin érkeztünk, létezik legalább egy olyan másik út, ami már legalább egyszer meg lett jelölve), akkor válasszunk egy olyan utat, ahol a lehető legkevesebb jelölés van eddig (lehetőleg 0, különben 1.)

- Ha olyan kereszteződésbe értünk, ahol már minden útnál két jelölést találunk, kivéve azt, amiből a kereszteződésbe érkeztünk, forduljunk vissza, és jelöljük meg a kereszteződést zsákutcaként.

Ez az algoritmus egyébként a 19. századból származik, és körülbelül száz évvel később generalizálták az algoritmust, és a mélységi keresés nevet kapta.



21. ábra A Trémaux algoritmus egy futtatása. Szürke vonal jelöli a már kétszer, fehér vonal jelöli a csak egyszer megjelölt utat. [4]

Tehát így a sárkány egy úton maximum kétszer fog végig haladni, ezzel pedig jóval hatékonyabban, gyorsabban tud adott esetben kijutni a labirintusból, mint a játékos.

Az algoritmus úgy került implementálásra, hogy az egy jelölést a *hasBeenVisitedByDragon true* értéke jelentette, a két jelölést, mivel ez magával hozza azt is, hogy arra az útra többet nem szabad lépni, a *DeadEnd* változóval helyettesítettem. Az implementációban a kisebb komponenseket a *doTremauxPathfinding* függvény fogja össze, ami igaz értékkel tér vissza akkor, ha megtalálta az utat, különben hamissal. Az algoritmusban a ciklus addig fut, amíg nem találtuk meg a végső cellát, vagy amíg az egyik belső függvénye nem mondja azt, hogy nem tud már hova menni, ezzel egyértelműsítve azt, hogy nincs kiút a labirintusból.
A ciklus egyszerűen működik mindaddig, amíg nem érkezünk el egy kereszteződéshez, ahol a leírt szabályokat kell majd alkalmazni. Tehát amíg nincs kereszteződés, addig triviális a sárkány számára, hogy mi a következő lépés, így ezt a goUntilJunction függvényben egyetlen lépésben meg tudja tenni.

Ez a függvény folyamatosan keresi a következő irányt, és többször felmerülhet a túlindexelés problémája, így erre egy *isValidMove* függvényt implementáltam, ami azt vizsgálja, kiindexelnénk-e a tömbből, hogyha a paraméterben megadott irányba

mennénk. Az irányok jobbra, balra, fel és le lehetnek, ezeket egy enumerációban, a *Direction* osztályban egyértelműsítettem.
Hogyha az *isValidMove* igazzal tér vissza, akkor tudhatjuk, hogy lekérhetjük a tömbből azt a szomszédot, amerre az adott iterációban épp mozogni szeretnénk. Ha lekértük a szomszédos cellát, megvizsgáljuk, hogy mozoghatunk-e abba az irányba egyáltalán (akadályozza-e fal a haladást), és hogy ha igen, akkor azt hogy hány jelölés van ott.Első körben mindig azt az irányt keressük, ami még felfedezetlen, ezzel elkerüljük azt hogy véletlenszerűen visszaforduljon a sárkány egy kanyargósabb folyosó közepén.Ha viszont ilyen utat nem talált, akkor valószínűleg abban a helyzetben vagyunk, hogy már visszafele jövünk ezen az úton, így ebben az esetben még egyszer megnézzük, hogy melyik irányba szabályos a mozgás, és a második esetben már csak az lesz a kitétel, hogy kétszer jelölt utat ne válasszunk. Ha ezek után sem sikerült olyan irányt választani, amely bármelyik feltételnek megfelel, akkor hamissal tér vissza a *goUntilJunction* függvény, ezzel jelezve az őt hívó függvénynek, hogy nincs kiút, ne próbálkozzon tovább. Ez az eset viszont ritkán fordul elő, így nézzük mi történik, ha bármelyik körben talál egy jó irányt.

Az új irány kiválasztása után megpróbáljuk változtatni a sárkány helyét, ezt a *selectNewCell* metódussal tesszük, ami egy irányt kap paraméterül, méghozzá azt, amit az előbb a szabályok alapján a legjobb iránynak véltünk. ez a függvény aztán ha ebben az esetben sem indexelnénk ki a tömbből, akkor a *tryGoingToNeighbourCell* függvényt hívja, ami végül meg is változtatja a sárkány aktuális pozícióját, ha azt a falak szerkezete, vagy a dupla jelölés nem gátolja. Abban az esetben, ha sikerült a helyváltoztatás, a képbe jön a késleltetés: mivel ez a kis szoftver egy szempillantás alatt lefut minden modern számítógépen, nem lenne túl fair a játékossal szemben, ha nem fognánk vissza a képességeit. Így az adott szálat, amin az algoritmus fut, bizonyos időre elaltatjuk, és ez a bizonyos idő a nézetben meghatározott paramétertől fog függeni, hiszen minél több ponttal rendelkezik a játékos, az algoritmus annál kevésbé van visszafogva. Tulajdonképpen ez volt egy iterációja a *GoUntilJunction* függvénynek, és ezt mindaddig csinálja, míg nem fut bele egy kereszteződésbe.

Kereszteződés esetén a fő függvénybe megyünk vissza, ahol a program fel fogja ismerni, hogy meg kell hoznunk az optimális döntést, azaz, hogy melyik irányba megyünk tovább. Hogyha minden másik út, ahova mehetnénk, már kétszeres jelöléssel rendelkezik,

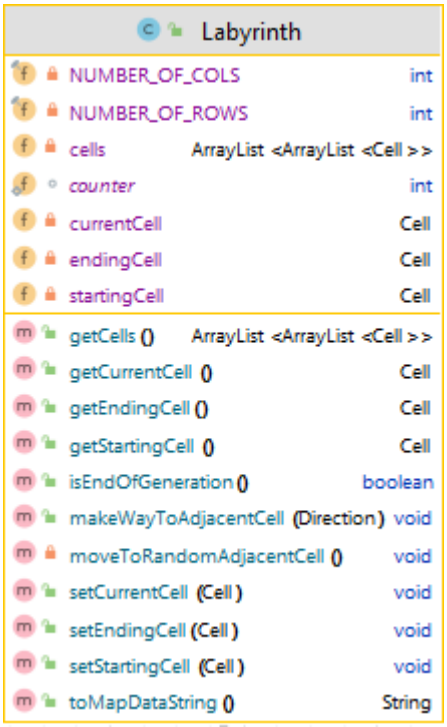
könnyű a dolgunk, mert csak vissza kell fordulnunk. Ha nem ilyen egyszerű a dolgunk, a *makeChoiceInJunction* függvényt hívjuk meg, hogy elemezze ki a helyzetet.

a *makeChoiceInJunction* függvény a lehetséges irányokat véletlenszerűen próbálgatja, és keresi közülük azt, amelyik a legkevesebb jelöléssel rendelkezik. Ha csak egyetlen utat talált, azt jelenti, hogy zsákutcában vagyunk, így meg is jelöli azt a területet, de mindenképpen visszatér egy iránnyal, amit aztán a fő hívó függvényünk megkap. A fő függvény ezután újra mozdulni próbál, a megállapított legjobb irányba, és ezzel végig is értünk egy iterációján a fő függvénynek, hiszen elértünk egy kereszteződésig, hoztunk egy döntést a következő irányról, majd meg is tettük a lépést abba az irányba. A következő iterációban újra vizsgálni fogjuk, hogy megint kereszteződésbe kerültünk-e, és így megy ez egészen addig, amíg nem találjuk meg a kiutat, vagy rá nem jövünk arra, hogy valójában nem is elérhető számunkra a kiút. Alább látható a fő hívó függvény struktúrája.

```
public boolean doTremauxPathFinding() {
    ArrayList<Cell> retList = new ArrayList<>();
    if(!currentCell.isStartingCell()) return false;
    currentCell.setHasBeenVisitedByDragon(true);
    while(!currentCell.isEndingCell())
    {
        if(!isJunction() || isThereUnVisitedRoad())
        {
            if(!goUntilJunction()) return false;
        }
        else if(isDeadEnd())
        {
            currentDirection = getOppositeDirection();
        }
        else
        {
            currentDirection = makeChoiceInJunction();
            selectNewCell(currentDirection);
        }
        retList.add(currentCell);
    }
    return currentCell.isEndingCell();
}
```

22. ábra A Trémaux algoritmus fő függvénye

3.4.1.3 A Labirintus osztály



23. ábra A labirintus osztály UML diagramja

A labirintus osztály felelős azért, hogy előteremtse a játékos számára a labirintust vagy egy adatbázisból letöltött adat segítségével, vagy pedig az Aldous-Broder algoritmussal, valamint azért, hogy a labirintusnak csak egy kezdete és egy kiútja legyen.

Az Aldous-Broder algoritmust a játék offline állapotában hívjuk meg. Ez az algoritmus egy olyan komplex labirintust készít, amelynek minden pontjából minden pontjába el lehet jutni. Komplex labirintusnak hívjuk azokat a labirintusokat, amelyek kiútja nem található meg az úgynevezett *Wall-Follower* algoritmussal, amely annyiból áll, hogy végig a menetirányhoz képest jobb oldali falat követjük. Azzal a példával is szoktak élni, hogy ha a labirintus falai olyan anyagból lennének, ami vezeti az áramot, akkor ha egy szimpla labirintusban áramot vezetnénk a falba, az végighaladna a labirintus összes falán, de a komplex útvesztő esetében nem.

Az algoritmus, ami a labirintust generálja, egész egyszerű, és a véletlenszerűsége elősegíti a játék újrajátszhatóságát és változatosságát. Ez az algoritmus is igényli azt hogy a cellákat valamilyen módon meg tudjuk jelölni a generálás során, de a jelenlegi implementációban ezt úgy oldottam meg, hogy ha a cellába valamelyik oldalról be lehet

lépni, akkor arra mondhatjuk azt, hogy már legalább egyszer meglátogattuk. Az algoritmus tehát a következő lépésekből áll:

1. Válasszunk egy véletlenszerű cellát, és jelöljük meg, mint már látogatott cella.
2. Csináljuk a következőt, amíg van olyan cella, ami még nem volt feldolgozva:
    a. Válasszunk véletlenszerűen egy szomszédos cellát.
    b. Hogyha a választott szomszédos cella még nem volt látogatva:
        i. Tüntessük el a falat az előző és a jelenlegi cella között
        ii. Jelöljük meg a jelenlegi cellát látogatottként.
    c. Jelöljük meg a választott szomszédot látogatottként.

```
Random random = new Random();
currentCell = cells.get(random.nextInt(cells.size())).get(random.nextInt(cells.get(0).size()));
setStartingCell(currentCell);
while(!isEndOfGeneration())
{
    moveToRandomAdjacentCell();
}
setEndingCell(currentCell);
```

24. ábra az Aldous-Broder algoritmus a kódban.

A labirintusnak ez az algoritmusa tehát két nagyon fontos függvénnyel rendelkezik: A szomszédra átváltó metódus, esetünkben a *moveToAdjacentCell*, és a generálás végét vizsgáló függvény, az *isEndOfGeneration*. Előbbi véletlenszerűen kiválaszt egy szomszédos cellát, ami, ha még nem volt látogatva, meghívja a *makeWayToAdjacentCell* metódust, átadva neki azt hogy melyik irányba lévő szomszédjához csináljon utat. Utóbbi szimplán azt vizsgálja, hogy a tömbünk minden cellájához lett-e csinálva legalább egy út, ezzel biztosítva azt, hogy amikor kezdeti és végpontot választunk a labirintusnak, biztosan legyen a kettő között út.

Így a cellák látogatottsági állapota a *Cell* osztályban található *GetHasBeenSelected* metódussal érhető el, ami egyszerűen megnézi, hogy minden fala fent van-e.

Viszont mindezt csak akkor használjuk, ha a konstruktor ennek megfelelően van felparaméterezve, különben csak feldolgozzuk a labirintus szöveges reprezentációját, és annak megfelelően állítjuk be a cellák falait. A labirintus szöveges reprezentációja a következőképpen néz ki:

- A sorrendje azonos a tömbben eltárolt cellák sorrendjével, így a tömbbel együtt iterálható a szöveg is a labirintus betöltésekor. A cellák információi szóköz karakterrel vannak elválasztva, és a cellák állapota a következőképpen derül ki:
    - Minden cella állapotát 4, vagy 5 karakter reprezentálja.
    - Az első 4 karakter vagy 1, vagy 0 értékkel rendelkezik, ezzel reprezentálva azt, hogy az adott irányból kell e fal a cellához, vagy sem. Az irányok a következő sorrendben vannak: Fel, le, balra, jobbra.
    - Az 5. karakter, ha van, akkor egy „s”, vagy egy „e” betű. Ezekből *pályánként* csak egy van a szövegben, hiszen a labirintusoknak csak egy kezdőpontja van. Előbbi a kezdő cellát, utóbbi a kiutat jelzi.

Tehát ha a cella szöveges reprezentációja „1101s” akkor egy olyan kezdőcellát kell létrehozni, amelybe csak a bal oldaláról lehet belépni, vagy ha „0000” lenne, akkor egy olyan celláról van szó, amely minden oldalról nyitott.

Így már minden részinformáció ismert ahhoz, hogy a konstruktor paraméterezhetőségéről beszéljünk. A két paraméter, amit fogad, a *boolean* típusú *isMapGenerationNeeded*, és a mapData. az *isMapGenerationNeeded* értéke attól függ, hogy szeretnénk-e az Aldous-Broder algoritmust futtatni, míg a mapData egy szöveges típusú paraméter, és nem üres akkor, hogyha van olyan adat amit be szeretnénk tölteni. Így egyetlen konstruktor lesz használható a később bemutatandó pályaépítő, az offline és online játékokhoz, ezzel elősegítve a kód karbantarthatóságát.

3.4.1.4 A játékos osztálya

Ez az osztály felelős azért, hogy a labirintusban egy bizonyos pozícióban elhelyezkedő játékos állapota tisztán elérhető legyen a nézet csomag számára, így tartalmazza azt, hogy a játékos milyen irányba néz, éppen mozog-e, melyik cellában, illetve azon belül is hogy hányadik pixelnél jár éppen. Mivel a játékos az egyetlen jelenlegi osztály, melynek

futásidőben gyakran változik a reprezentálásra használt képe, ezt az osztályban egy külön tömbben, a *myLooks*-ban tárolom. Ez a paraméter aztán folyamatosan változik a példány állapotának függvényében, amikor a nézet csomag valamely komponense meghívja az *updateLook* függvényt.

```
public void updateLook()
{
    switch(myDirection)
    {
        case RIGHT:myLooks = ResourceLoader.stever; break;
        case LEFT: myLooks = ResourceLoader.stevel; break;
        case UP:   myLooks = ResourceLoader.steveu ; break;
        case DOWN: myLooks = ResourceLoader.steved; break;
    }
    if(amIMoving)
    {
        myLook = myLook == myLooks[0] ? myLooks[1] : myLooks[0];
    }
}
```

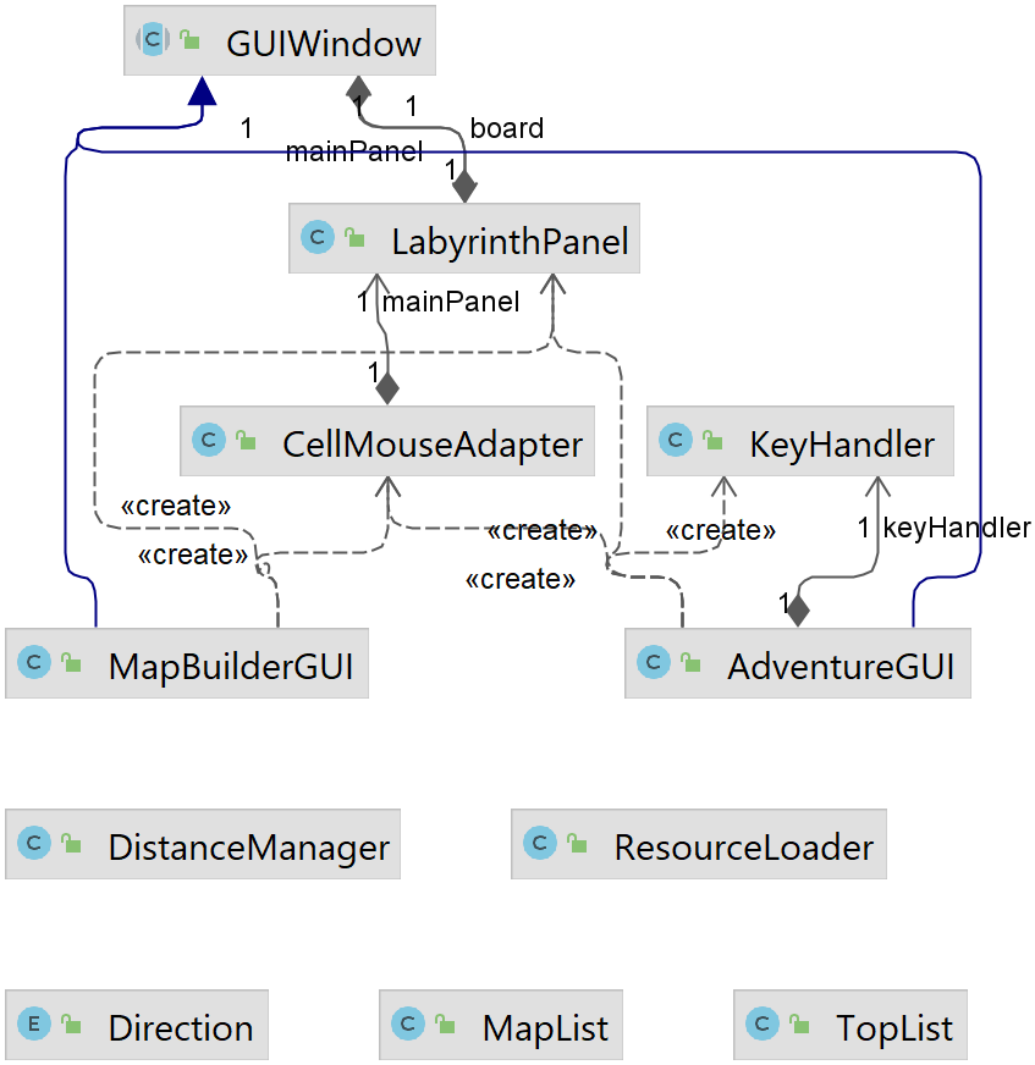
25. ábra az *updateLook* függvény, amely a példány állapotának vizuális reprezentációjáért felel.

Ez a függvény a később bemutatandó, vizuális kellékek elérhetőségéért felelős osztály adattagjai közül választja ki megfelelőt.

3.4.2 A nézet csomag

Ez a csomag felelős mindenért, ami a képernyőn vizuálisan megjelenik, hiszen a fő feladata, hogy az imént bemutatott, modell csomagban helyet foglaló osztályokból példányosított objektumokat az állapotuk szerint megjelenítse. Itt van a pályakészítő, azaz *MapBuilderGUI* komponens, az offline és online játékért egyaránt szolgáló *AdventureGUI*, vagy a toplista és a labirintusok megtekinthetőségéért,

adminisztrálhatóságáért felelős MapList osztály is. A csomag UML osztálydiagramja a következőképpen fest:



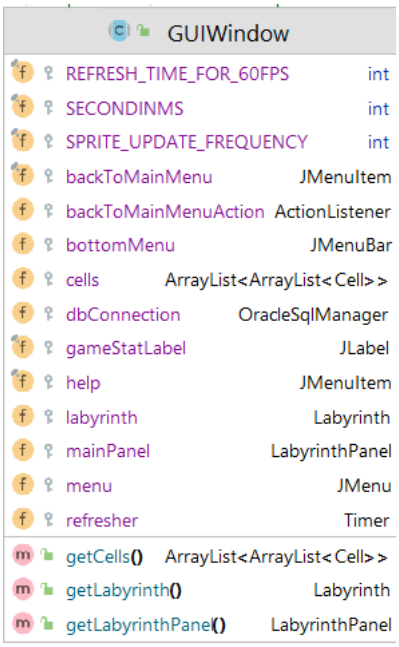
26. ábra a nézet csomag UML osztálydiagramja

3.4.2.1 A ResourceLoader *osztály*

Azért ezzel az osztállyal kezdem a csomag bemutatást, mert ennek az osztálynak az adattagjait később minden másik osztály használni fogja, és esszenciális hogy az adattagjai inicializálva és elérhetőek legyenek a többi osztály számára. Az alapvető ötlet az volt az osztály mögött, hogy a képek betöltése és memóriában tartása alkothatna egy külön statikus osztályt, amivel biztosítom azt, hogy csak egyszer lesznek a memóriába betöltve, de mégis mindig elérhető mindegyik másik osztály számára. Így az adattagjai mind kép formátumúak, és az *initResources* metódust a játék elején meghívva

megbizonyosodhatunk arról, hogy ezek a fájlok végig a memóriában maradnak, és bármelyik használandó funkció esetén rövid idő alatt elérhetővé válik a játékszoftver számára.

3.4.2.2 A GUIWindow *osztály*

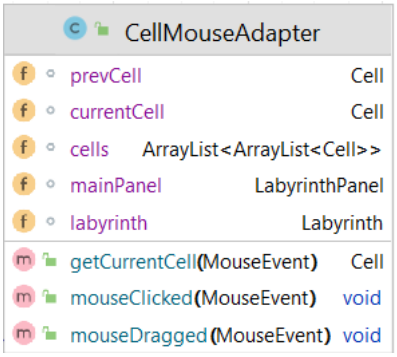


27. ábra A *GUIWindow* osztály felépítése

Ez az absztrakt osztály egy jó alapként szolgált arra, hogy a többi osztályban ennek a használatával a futásidejű polimorfizmus könnyen megvalósítható legyen, hiszen így elég csak ezt az osztályt használni az *AdventureGUI* és a *MapBuilderGUI* helyett, majd amikor ténylegesen valami osztályspecifikus adatot szeretnénk megjeleníteni, például majd az *AdventureGUI* esetében a játékost, vagy a sárkányt, akkor azt könnyebben meg lehet oldani. Az osztály konstruktora is használható arra, hogy az ebből származtatott osztályokban közös, alapértelmezett adatokat már előre be lehessen állítani, így aztán a későbbi osztályokban ez könnyen meghívható az ősosztály konstruktorának hívásával.

Ide szúrnék egy rövid kitérőt arról, hogy a játékban felhasznált képek egy része a saját képszerkesztői munkám, ahol felhasználtam néhány Minecraft játékból érkező textúrát a sajátjaim elkészítéséhez, a játékos képét egy YouTube sorozat alatt megosztott, egyébként ingyenesen felhasználható képekből használtam fel,.[15] A sárkány kinézete pedig a Minecraft nevű számítógépes játékának főellensége, az Ender Dragon egy rajzolt változata lett felhasználva. [16]

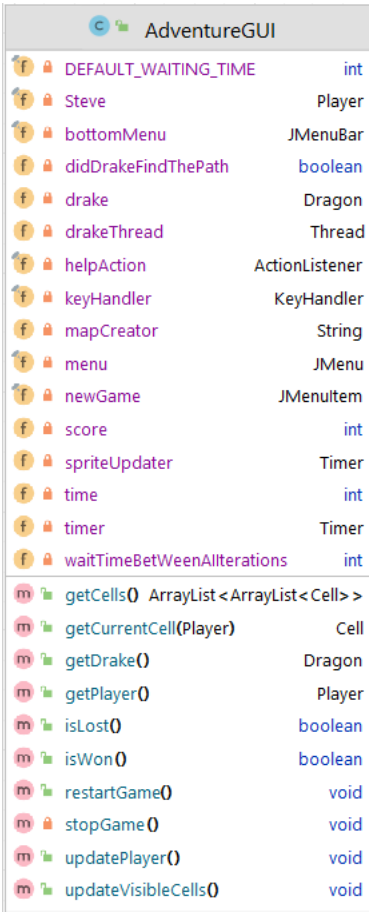
3.4.2.3 A pályakészítő, és a CellMouseAdapter *osztálya*



28. ábra a *CellmouseAdapter* osztály felépítése

A pályakészítő, azaz a *MapBuilderGUI* osztálya a *GUIWindow*-hoz képest nagyon kevés extra információt tartalmaz, hiszen tulajdonképpen annyira van szükség, hogy megjelenítsük a cellákat, és különböző egérműveletekre változtassuk a kiválasztott cellák állapotát. Az alapvető ablak felépítése egy *JFrame*-ből, egy *JPanel*-ből és egy *JMenu*-ből áll, amit aztán később feltöltünk plusz adatokkal, például a segítség vagy a pálya elmentése funkciókat előidéző gombokkal. Az imént bemutatott labirintus adattagot így úgy paraméterezzük fel, hogy sem az Aldous-Broder, sem a pályabetöltős részt nem alkalmazza, tehát egy falakkal teli tömböt kapunk a végén. A labirintusra egy *CellMouseAdapter* figyelő lett beállítva, amely az egérmutató pixel koordinátái alapján kiszámolja, hogy melyik cellára kattintottunk, vagy kattintás közbeni húzás vagy angol nevén „drag” esemény esetén megadja azt, hogy melyik celláról milyen irányba húztuk el az egérkurzort. Így, ha rendelkezésre áll a jelenlegi cella, és az irány, hogy milyen irányba kell mozogni, akkor onnantól már meg tudja hívni a labirintus *makeWayToAdjacentCell* metódusát a kiszámolt iránnyal, és így meg is valósult a pályakészítő fő része.

3.4.2.4 A játékmenet osztálya, az AdventureGUI



29. ábra az *AdventureGUI* osztály felépítése

Ez az egyetlen osztály a csomagon belül, amely az összes modellben definiált és dokumentált osztályt példányosítja, és használja: A háttérben lévő labirintusra a játékost és a sárkányt is egyaránt felrajzolja, miközben fut a sárkány algoritmusa, másodpercenként nagyjából hatvanszor újra rajzoljuk a panelt, számoljuk hogy mennyi ideje próbálunk az adott labirintusból kijutni, és gondoskodik a sárkány folyamatos gyorsításáról. Emellett fontos kiemelendő rész, hogy a két konstruktora van definiálva, ebből az egyik egy *OracleSQLManager* típusú paramétert vár, amivel ez a konstruktor lesz felelős az online játékért, míg a másik az offline módot aktiválja.

Az osztályban 3 időzített metódus, és egy plusz *Thread* fut:

- a *timer* nevezetű időzítő, amely egy egyszerű 1 másodperces, azaz 1000 milliszekundumos számláló, amibe belekerült az eltöltött idő számolása, és annak az ellenőrzése, hogy a játék véget ért-e.

    - A játékok aktuális eredményének (tehát hogy nyertünk, esetleg vesztettünk már) eldöntésére az *isLost* és *isWon* metódusok szolgálnak: előbbi abban az esetben igaz, ha a sárkány metódusa végigfutott, és ezáltal a *didDrakeFindPath* változó igaz értéket kapott, utóbbi pedig akkor lesz igaz, hogyha a játékos jelenlegi cellája, azaz pozíciója megegyezik az éppen generált labirintus kiútjával. Ezeket a kondíciókat a játék másodpercenként nézi.
- a *refresher* időzítő, amely egy 15 milliszekundumos késleltetéssel rendelkezik, ezzel elérve a nagyjából 60-szoros frissítést, amely egy egészséges terhelés melletti optimális, kellően reszponzív és folyamatos képet biztosít. A következő metódusokat hívja:
    - A labirintus panelnak a teljes újrarajzolását, amelyben implementálva van a játékos és a sárkány kirajzolása is
    - A látható cellák ellenőrzését és frissítését, ezzel elősegítve, hogy tényleg akkor, abban a pillanatban lesznek láthatóak az újonnan felfedezett cellák, amikor a játékos odaér.
- a *spriteUpdater* időzítő, 100 milliszekundumonként kerül meghívásra az egyetlen hozzárendelt függvény:
    - A *Player* osztály *updateLook* metódusát hívja, ami aztán beállítja magának az iránynak megfelelő, illetve a lépés előidézése érdekében folyamatosan váltakozó vizualizációt. Ez azért pont ide került, mert a többi időzítőben a két metódushívás között eltelő idő vagy túl sok, vagy éppen túl kevés ahhoz, hogy a lépegető, reszponzívan irányt váltó játékos látványát előteremtsük.
- a *drakeThread* szál, mely folyamatosan fut, de a sárkány paraméterében megadjuk a *waitTimeBetweenAIIterations* paramétert, amelyet a sárkány később a szál altatására fog használni, így olyan tempót adva az algoritmus egyes lépéseinek, amellyel az ember is tud versenyezni. Ez a paraméter aztán természetesen a további pályák elérésével folyamatosan csökken, egészen pontosan alkalmanként az előző érték 0.95-szörösére.

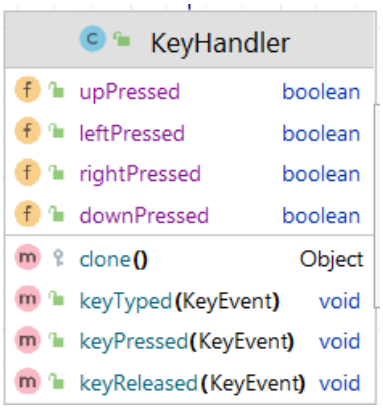
A játékok újraindítására a *restartGame* metódus szolgál, amely megállítja az imént felsorolt időzített funkciókat, megszakítja a *drakeThread* szálat, új labirintust tölt vagy generál a konfigurációtól függően, ezeket újra elhelyezi az ablakon belül, majd, ha elkészítette az eredeti ablakot ismét, akkor újra elindítja az időzített metódusokat és a sárkány szálát.

Arra az esetre, ha a játék a játékos kérésére, vagy egyéb okok miatt véget ér, a *stopGame* metódus a legalkalmasabb. Ez csak leállít minden párhuzamosított folyamatot, nullázza az időt és a pontszámot, majd bezárja az ablakot. Természetesen, ha az elveszített játék végett kerül meghívásra, előtte megtörténik a pontszám szinkronizálása, ha az adatbázis kapcsolat (még mindig) elérhető.

3.4.2.5. DistanceManager, *távolság számoló osztály*

Egy egyszerű statikus metódus, amely a játékos mozgása közben segít távolságot számolni a kirajzolt játékos képe és a cellák szélei között. Ha a cella adott oldalán fal található, akkor ez az osztály, de leginkább egyetlen metódusa, az *isClosetoEdges* lesz felelős azért, hogy eldöntse, mehet-e még az adott irányba a játékos vagy sem. A játékos képének négy sarkának koordinátáit figyeli, és számolja ki a cella falai közötti távolságot. Egy felbontás alapján skálázódó, *sizeMultiplier*-re keresztelt segédváltozó dönti el azt, hogy hány pixel lesz az a távolság, aminél közelebb már nem lehet az adott cella adott széléhez a játékos.

3.4.2.6. KeyHandler, *a játékos mozgásáért felelős osztály*

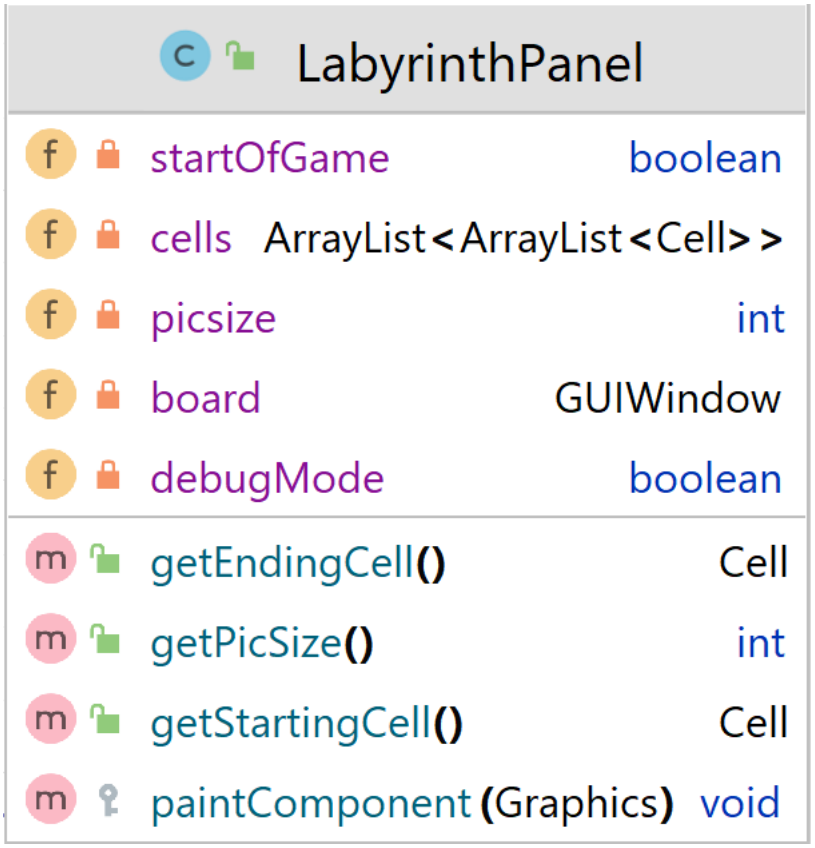


30. ábra a KeyHandler osztály diagramja

Ebben az osztályban van implementálva az, hogy éppen melyik billentyű van lenyomva, és irányonként egy segédváltozóval el van intézve az, hogy a lenyomott billentyűk

folyamatos mozgást idézzenek elő. Tehát ha például folyamatosan fölfele szeretnénk menni, akkor az *upPressed* nevű segédváltozó igaz értéket fog kapni, és mindaddig tartja ezt az értéket, amíg nem engedjük fel a felfele nyilat, vagy a W billentyűt. Amint megtörténik a felengedés, újra hamis értéket fog kapni, és így az AdventureGUI-ban a mozgató metódus mindig egy éppen aktuális állapotot fog kapni, és ezáltal folyamatos lesz a játékos mozgása.

3.4.2.7. A labirintust vizualizáló metódus, azaz a LabyrinthPanel



31. ábra a LabyrinthPanel felépítése

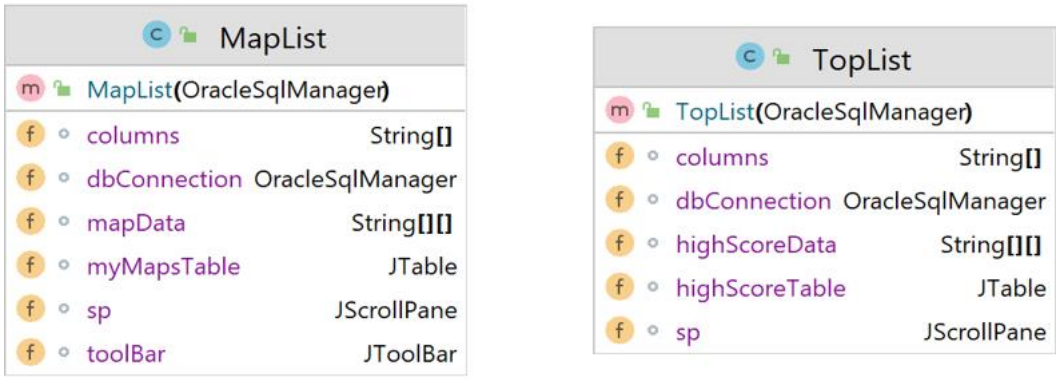
Ez a panel rajzolja ki a legtöbb elemét a játéknak, például a játékost, a sárkányt vagy a cellákat. Az *AdventureGUI* csak tartalmazza ezt az ablakot, ehhez még hozzákerül a menü, ami már annak az osztálynak a felelőssége. Itt a két legfontosabb metódus az a konstruktor, ahol az *AdventureGUI* ősosztálya van átadva, hogy aztán ez az osztály alkalmas legyen a *MapBuilderGUI* és az *AdventureGUI* kezelésére is.

a *paintComponent* metódus az ősosztályának egy felülbírált függvénye, amelyben először meghatározásra kerül az, hogy pontosan mekkora pixelméretet kell cellánként alkalmazni ahhoz, hogy minden kép pont kiférjen az ablakra. Ezután a cellákat lentről

fölfelé, balról jobbra egyenként kirajzoljuk, úgy, hogy egymással ne érintkezzenek. Eztán futásidejű polimorfizmus használatával megnézzük, hogy a board adattag milyen típusú (vagy AdventureGUI típusú, vagy nem), és ha AdventureGUI, akkor ezáltal megbizonyosodhatunk arról, hogy a játékos és sárkány is elérhető, így azok pozícióit is lekéri és kirajzolja.

Valamint mivel itt derül ki, hogy pontosan melyik pixelre kell a játékost kirajzolni a játék elején, hiszen ez minden esetben függ a képernyő aktuális felbontásától, ezért egy *startOfGame* segédváltozóval egy egyszeri pixelbeállítást végzünk, hogy aztán ahhoz a pixelhez képest tudjon relatív mozogni a játékos. A sárkány kirajzolása a cellánkénti kirajzolásnál egy elágazás formájában lett implementálva. Így végül ez a metódus van hatvanszor meghívva másodpercenként.

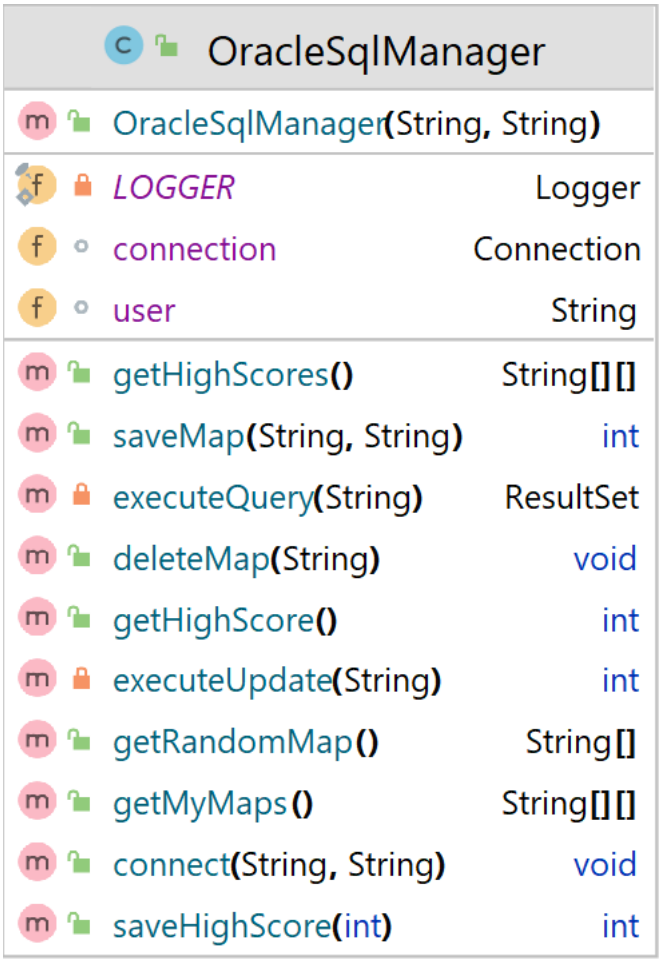
3.4.2.8 A táblázatok, azaz a MapList és TopList



32. ábra a MapList és a Toplist osztályok diagramjai

Ezek az osztályok a JFrame Swing osztályból van származtatva, a legfontosabb komponensük pedig egy JTable szintén a Swing csomagból. Ebben az *isCellEditable* függvényt felülbíráljuk, hogy mindig hamis értékkel térjen vissza, ezzel elérve hogy a táblázat cellái nem lesznek a felhasználó által átírhatók, elkerülve néhány bizonsági kockázatot. Tulajdonképpen egyedül a konstruktorban össze is áll a komponens, feltöltve adatokkal. A konstruktornak át kell adni az adatbázis kapcsolatot jelképező *OracleSqlManager* példányt, majd a kettő közötti a különbség az az hogy míg az MapList a *getMyMaps* metódusból, addig a TopList a *getHighScores* függvényből szerzi be az adatot, amivel feltölti a belső JTable adattagot.

3.4.3 A persistence osztály, és egyetlen osztálya, az OracleSqlManager



33. ábra az OracleSqlManager diagramja

A csomag egyetlen osztálya az *OracleSqlManager* tehát, amely az Oracle Java Database Connectivity, vagy rövidebben OJDBC-ben definiált függvényeket használva hajt végre lekérdezéseket, adat törlést, adat frissítést az adatbázisban. A lekérdezések String formában kerülnek összeállításra, így bármilyen információ hozzáadható a paraméterekből.

Az ELTE szervereihez történő csatlakozás a JSCH, azaz a Java Secure Channel segítségével jön létre. Ez a Caesar szerverre csatlakozik, ott egy alapvető, előre megadott userrel autentikálja magát, majd onnan megpróbál az aramis szolgáltatáshoz kapcsolódni a connect metódusnak paraméterben megadott adatok segítségével.

Az osztály tehát az Aramis szerveren található IDU27K nevű felhasználó tábláiból kérdez le, így ez előzetesen beállításra került az Oracle-ben, hogy mindenki számára elérhető, módosítható, és sorai törölhetők legyenek.

Az osztályban a két fő metódus, amit aztán a többiek használnak az az *executeQuery* és *executeUpdate*, amelyből az első egy lekérdezés *ResultSet* típusú eredményét adja vissza, míg az *executeUpdate* azt, hogy hány sort sikerült frissíteni az adatbázisban

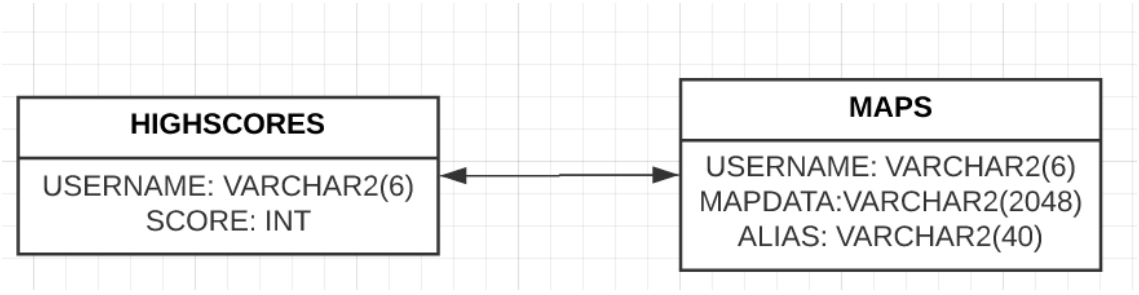
A *getHighScores* és *getMyMaps* metódusok egyenként egy lekérdezést valósítanak meg, előbbi a HIGHSCORES tábla tartalmát, pontosabban az első 10 találatot kérdezi le, utóbbi a MAPS tábla tartalmát, visszaadva ezeket egy String mátrixban.

A *saveMap* funkciót a pályakészítő osztálya használja, ahol már átadja az alias-t, és a *saveHighScore* akkor kerül meghívásra csak, ha a *getHighScore*, a bejelentkezett felhasználó pontszámát lekérdező metódus visszatérési értéke kisebbnek bizonyul a játékban éppen aktuális pontszámnál, ez tulajdonképpen először megpróbálja beilleszteni az új sort a táblába, majd ha ez nem sikerült, akkor csak simán frissíteni próbálja azt a sort, amiben az éppen bejelentkezett felhasználó neve szerepel.

3.5 Az adatbázis

Az adatbázis, amelyet az imént bemutatott OracleSqlManager használ tehát, az ELTE Aramis SQL szervere, így az autentikáció és regisztráció a szerver karbantartói által konfigurált beállítások szerint történnek. A játékosok az IDU27K felhasználó két tábláját módosítják és kérdezik le, név szerint az IDU27K.HIGHSCORES és az IDU27K.MAPS táblákat.

az IDU27K.HIGSHCORES táblának kettő oszlopa van, az első az USERNAME, amely egy VARCHAR2(6) típusú, amely azt eredményezi, hogy a felhasználó neve 6 vagy kevesebb karakter lehet. Ez az oszlop nem rendelkezhet null értékkel, ráadásul PRIMARY KEY megszorítás van alkalmazva rá, tehát egy felhasználóhoz csak egy pontszám tartozhat. A második oszlop a score, melyre különösebb megszorítást nem alkalmazunk, akár 0 is lehet az érték.



34. ábra Az adatbázis táblái, és a kapcsolataik

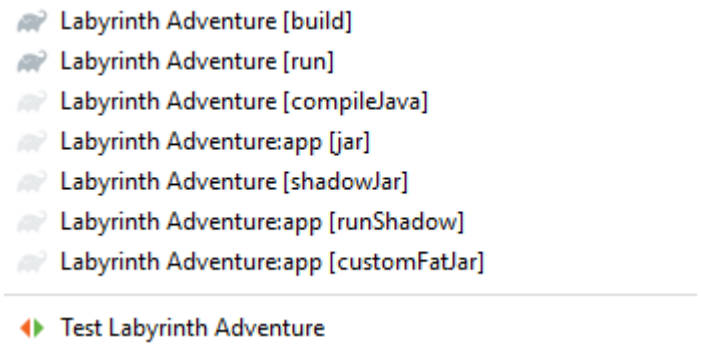
Az IDU27K.MAPDATA táblának viszont már 3 oszlopa van, az első a MAPDATA, amely akár 2048 karaktert képes tárolni a jelenlegi konfigurációban, így a későbbiekben marad jócskán hely arra, hogy a nagyobb labirintusokat is tudja kezelni, mert jelenleg egy labirintus szöveges reprezentációja körülbelül 580 bájt helyet foglal. a második oszlop ugyanúgy az USERNAME, mely még mindig 6 karaktert tartalmazhat, végül egy maximum 20 karaktert tartalmazó ALIAS, amely egy becenévként szolgál arra a célra, hogy a labirintus készítői megismerjék a labirintusukat a táblázaton belül. Megszorítások terén elvárjuk, hogy a beszúrt sorok esetén egyik érték se legyen null.

3.6 Tesztelés

A program egységteszteket használ, a JUnit 4-es keretrendszert használva. Ennek a segítségével mindegyik komponens alapvető, és esszenciális funkciói a kritikus eseteket beleértve is tesztelésre kerülnek.[15] Az esszenciális funkciók közé tartozik:

- adatbázishoz való csatlakozás
- a labirintus generáló algoritmus
- a pályabetöltő algoritmus,
- a sárkány, és leginkább annak útkereső metódusa
- a toplista
- a pályák listája

A tesztelés jelenlegi konfigurációja bármilyen tesztelő által elindítható, ha a projektet importálja az IntelliJ IDEA fejlesztői környezetbe.



35. ábra futtatási konfigurációk, közte a teszt konfigurációval

A tesztesetek rövid leírása:

- *connectWithSQLIsSuccessful* : Egy előre megadott felhasználóval megpróbál a szerverre bejelentkezni, és ellenőrzi, hogy sikeres volt-e.
- *TestSteveInitPosition:* Egy teljes AdventureGUI osztályt képez, amit az Aldous-Broder algoritmussal generál, és ellenőrzi, hogy a labirintus elkészültekor a játékosunk (Steve fantázianévre hallgatva) ténylegesen a kezdeti cellán van-e.
- *mapListInitialization*: Tulajdonképpen a „My maps” gombra kattintás tesztelése, először csatlakozunk az adatbázishoz, majd ha ez sikerült, akkor lekérjük a hozzánk tartozó pályákat, és ellenőrizzük hogy például az első helyen tényleg egy értelmes pálya szerepel-e.
- *topListInitialization*: Az előző mintájára ez a teszt a „Toplist” funkciót próbálja ki. Szintén csatlakozik az adatbázishoz, lekéri az SQL szerveren található highscores névre hallgató táblából az adatot, majd ellenőrzi hogy az első helyezett kicsoda, és még mindig ugyanaz-e a pontszáma.
- *EmptyLabyrinthCantPass*: A sárkány útkereső metódusát teszteli, hogy teljesen zárt, mindenhol fallal rendelkező labirintusra hamissal tér-e vissza.
- *IncorrectStringThrowsException*: Ez a teszteset azt ellenőrzi, hogy ha nem megfelelő hosszú a pálya szöveges reprezentációja, amit ebben az

esetben lokálisan adtunk meg, akkor ténylegesen kivételt dob-e a program.

- *buildLabyrinthWithAldousBroder*: Száz darab labirintust generál az Aldous-Broder algoritmussal, és a sárkány útkereső algoritmusát meghívja rá. Ha a sárkány minden esetben kitalál a labirintusból, sikeres a futás.
- *TestMapBuilderCells*: Egy Labyrinth objektumot hoz létre, azokkal a paraméterekkel ahogy azt a MapBuilderGUI-ban használjuk. Eztán cellánként leellenőrzi, hogy tényleg a funkcióhoz megfelelően, minden cellának mind a négy fala aktiválva van.

Természetesen ezek a tesztesetek mellett a játék tesztelése manuálisan végig folyamatban volt, a következő tesztek kerültek rendszeresen elvégzésre:

- Teszt: A játékos bejelentkezik egy jó felhasználónév és jelszó párossal
  Elvárt eredmény: Megjelenik a főmenü az online funkciókkal
- Teszt: A játékos bejelentkezik egy rossz felhasználónév és jelszó párossal
  Elvárt eredmény: Csak az offline állapotban elérhető menük jelennek meg.
- Teszt: A játékos elindítja az offline/online játékot
  Elvárt eredmény: A sárkány a kezdőpontba kerül, és elkezdi keresni a kijáratot. A játékos is a kezdőpontba kerül, és mozdítható lesz a nyilakkal.
- Teszt: A játékos egyszerre több irányba szeretne mozogni, több gombot tart lenyomva.
  Elvárt eredmény: A játékos karaktere egyszerre csak egy irányba fog mozogni.
- Teszt: A játékos elindítja az offline/online játékot, majd kilép belőle a menübe.
  Elvárt eredmény: Minden futó szál leáll, amely az adott ablakhoz kötődött.

- Teszt: A pályakészítőben minden falat kitörlünk, és megpróbáljuk elmenteni
  Elvárt eredmény: A pálya mentésre kerül.
- Teszt: A pályakészítőben semmilyen falat nem törlünk, és megpróbáljuk elmenteni
  Elvárt eredmény: Hibaüzenetet kapunk amiatt, hogy nincs kiút a labirintusból.
- Teszt: Offline vagy online játék során a játékos bármilyen irányban mozogni próbál, de fal áll előtte.
  Elvárt eredmény: A játék nem engedi, hogy a karakter abba az irányba mozogjon, amerre a fal van. Minden olyan irányba engedi mozogni, ahol nincs fal.
- Teszt: A pályakészítőben készít egy pályát a játékos, de nem jelöli meg a kiutat a mentés előtt.
  Elvárt eredmény: A játék kiútként jelöli meg a legutolsó cellát, ahova utat csinált a játékos, majd elmentheti a pályát.
- Teszt: A játék során a játékos hamarabb kijut a labirintusból, mint a vele versenyző sárkány.
  Elvárt eredmény: A játék érzékeli, hogy a sárkány nem ért be, leállítja az összes párhuzamosan futó folyamatot, az ablakot újra betölti, ezzel generálva vagy betöltve egy új labirintust, majd a labirintus kezdőpontjára állítja a sárkányt és a játékost egyaránt. A pontszámot megnöveli eggyel, frissíti az alul látható menüsávban a készítő nevét (ha az különbözik az előző labirintus készítőjétől) a sárkány lépések közötti várakozási idejét pedig 0.95-szörösére szorozza.
- Teszt: A játék során a játékosnak nem sikerül elég gyorsan a labirintusból kijutnia, de a sárkány hamarabb kiér.
  Elvárt eredmény: Egy hibaüzenet jelenik meg, jelezve azt, hogy elvesztettük a játékot, és kiírja az éppen aktuális pontszámunkat. Miután a felugró ablakot bezárta a játékos, a játék újra a főmenübe navigál minket, miután aktív internetkapcsolat, és megdöntött legjobb pontszám esetén a pontszámunkat az adatbázissal szinkronizálta.

- Teszt: A játékos a főmenüben a „My Maps” menüpontra kattint, kiválaszt egy labirintust és a „Delete Map” gombra kattint.
  Elvárt eredmény: Egy felugró ablak jelzi, hogy sikeres volt a törlés, és ha az ablakot bezárjuk, majd újra megnyitjuk, már nem látható ott a törölt pálya.
- Teszt: A játékos a főmenüben a „Toplist” menüpontra kattint.
  Elvárt eredmény: Megjelenik a top pontszámokat tartalmazó ablak, oszlopnevekkel ellátva, a pontszámok szerint csökkenő sorrendbe rendezve. Ha 10-nél több jegyzett felhasználó van, csak az első 10 jelenik meg.
- Teszt: A játékos megnyitja a toplistát, vagy a labirintusait, és egy cellát kijelölve megpróbálja szerkeszteni a tartalmát.
  Elvárt eredmény: A táblák cellái nem lesznek szerkeszthetőek.

4. A további fejlesztési lehetőségek

A játék természetesen, mint bármelyik program, sosem készül el teljesen, mindig lenne rajta mit javítani, fejleszteni. A következő, szerintem a játék összképén sokat dobó ötleteim merültek fel a fejlesztés során:

1. Egy új grafikus motor: a játék jelenleg csupán Swing elemeket és képeket megjelenítve, azokat gyorsan újrarajzolva működik. Egy olyan könyvtár, amely ezeknél modernebb, hatékonyabb megoldást kínál, esetleg 3D-s vizualizációt is, megkönnyítené a fejlesztő feladatát, és a játékos számára is egy sokkal immerzívebb, valósághűbb élményt adhatna.
2. Valós idejű többjátékos mód: Egy olyan mód elkészítése, ahol egy előszobába belépve, majd elég játékost összegyűjtve a játékosok teljesen párhuzamosan, egymás mozgását valós időben látva próbálkoznának a kijutásban, egy többkörös felvonásban, ahol körönként fokozatosan csökkenő pontszámmal jutalmazzuk a leggyorsabb kijutókat.

3. Nagyobb, skálázható labirintusok: Jelenleg is működne minden algoritmus bármilyen méretű labirintusra, egyedül a megjeleníthetőséggel akadna probléma. Akár össze is köthető az új grafikus motorral ez a fejlesztés, de összességében arról szól, ha például az egérgörgő hatására nagyítható/kisebbíthető lenne a cellák mérete, ezáltal több is kiférne a képernyőre, az még növelné a játék komplexitását, és nehézségét, ezáltal a kompetetív jelleget is.

5. Összefoglalás

A dolgozatom végére érkezve megismerhettük az én elképzelésemet arról, milyen lehet egy kellően nehéz, mesterséges intelligenciát alkalmazó, akár teljesen offline működő számítógépes alkalmazás. A felhasznált technológiák bár nem a legmodernebbek, talán, ami a programot a leginkább régi hatásúként állítja be, az a Swing használata, de a játékot továbbfejleszteni vágyókat semmi sem köti ehhez. Az implementáció során sok akadályba ütköztem, főképp az algoritmusok implementálása az, amit nem klasszifikálnék leányálomként.

A játék manuális tesztelése során gyakran segítségemre voltak egyéb harmadik személyek is, amibe beletartozik a családom, illetve egyéb közeli barátaim, és a siker érzetével töltött el annak a látványa, hogy élvezik, elgondolkodtatja őket.

Ezen program játékosai tehát, ha hozzám hasonló tudásvággyal rendelkeznek, azaz szeretnék gyakorolni, megtanulni, hogy hogy lehet egy labirintusból effektíven, optimális lépésszámmal kijutni, esetleg olyan játékot keresnek, melyben aztán akár egymással tudják gyorsaságukat, memóriájukat, és józan eszüket összemérni, eleget tudnak tenni ezen ambíciójuknak.

6. Idézett forrásmunkák

[1] A Windows 10 rendszerkövetelményei, Microsoft, [Online]
https://support.microsoft.com/hu-hu/windows/a-windows-10-rendszerk%C3%B6vetelm%C3%A9nyei-6d4e9a79-66bf-7950-467c-795cf0386715
[Hozzáférés dátuma: 10 05 2022]

[2] Java downloads | Oracle, [Online]
https://www.oracle.com/java/technologies/downloads/#jdk18-windows
[Hozzáférés dátuma: 10 05 2022]

[3] Gradle, Wikipédia, [Online],
https://hu.wikipedia.org/wiki/Gradle
[Hozzáférés dátuma: 10 05 2022]

[4] Tremaux Algorithm , Youtube, [Online],
https://www.youtube.com/watch?v=gVSEJdSQZVQ
[Hozzáférés dátuma: 10 05 2022]

[5] Maze generation Algorithm, Wikipédia, [Online]
https://en.wikipedia.org/wiki/Maze_generation_algorithm
[Hozzáférés dátuma: 10 05 2022]

[6] JUnit , Wikipédia, [Online]
https://hu.wikipedia.org/wiki/JUnit
[Hozzáférés dátuma: 10 05 2022]

[7] Use case diagram, Wikipédia, [Online] ,
https://en.wikipedia.org/wiki/Use_case_diagram
[Hozzáférés dátuma: 10 05 2022]

[8] Unified Modeling Language, Wikipédia, [Online],
https://hu.wikipedia.org/wiki/Unified_Modeling_Language
[Hozzáférés dátuma: 10 05 2022]

[9] Integrált fejlesztői környezet, Wikipédia, [Online]
https://hu.wikipedia.org/wiki/Integr%C3%A1lt_fejleszt%C5%91i_k%C3%B6rnyezet

[10] Jetbrains IntelliJ letöltés, [Online],
https://www.jetbrains.com/idea/
[Hozzáférés dátuma: 10 05 2022]

[11] Java Secure Channel, [Online]
http://www.jcraft.com/jsch/
[Hozzáférés dátuma: 10 05 2022]

[12] Java Database Connectivity, Wikipédia, [Online]
https://hu.wikipedia.org/wiki/Java_Database_Connectivity
[Hozzáférés dátuma: 10 05 2022]

[13] JUnit, Wikipédia, [Online]
https://hu.wikipedia.org/wiki/JUnit
[Hozzáférés dátuma: 10 05 2022]

[14] Hamcrest, Wikipédia, [Online]
http://hamcrest.org/

[Hozzáférés dátuma: 10 05 2022]

[15] How to make a 2D Java Game,YouTube, [Online]
https://www.youtube.com/watch?v=vztluVKH4P4&list=PL_QPQmz5C6WUF-pOQDsbsKbaBZqXj4qSq
[Hozzáférés dátuma: 10 05 2022]

[16] A sárkány kinézete, KindPNG, [Online]
https://www.kindpng.com/imgv/hibxmmT_drawn-minecraft-ender-dragon-dragon-de-minecraft-png/
[Hozzáférés dátuma: 10 05 2022]

[17] Egységtesztelés, Wikipédia, [Online]
https://hu.wikipedia.org/wiki/Egys%C3%A9gtesztel%C3%A9s
[Hozzáférés dátuma: 10 05 2022]