Eötvös Loránd Tudományegyetem

Informatikai Kar

Informatikatudományi Intézet

Programozási Nyelvek és Fordítóprogramok Tanszék

ROBOTICOFFEE

Szerző:

Légrádi Bence

Programtervező informatikus BSc.

Témavezető:

Kaposi Ambrus

egyetemi docens

Budapest, 2025

Tartalomjegyzék

1. Bevezetés

Záróvizsga projektemnek egy olyan játékot választottam, amiben valós idejű programozással kell egy robotot irányítani, aki egy kávéház vendégeit szolgálja ki. A program ötletét főleg a sokak által ismert Logo teknős programozása adta, ahol a teknősnek bizonyos mozgató utasításokat adva formákat rajzolt. Ehhez hasonlóan az én robotom is a kétdimenziós síkban mozog, az irányításához szükséges utasításokat „saját programnyelv" fejlesztésével oldottam meg. A programnyelv a Java-hoz hasonlít, de ki van egészítve mozgató funkciókkal, továbbá állapot lekérdező metódusokkal, amik a robot helyzetét, vagy rendelési információkat adnak vissza. A projekt célja, hogy bemutassa, hogyan lehet egy valós idejű programozási környezetet létrehozni, amely nemcsak szórakoztató, hanem oktatási célokra is használható. A játékosok a programozási nyelv alapjait sajátíthatják el, miközben egy robot működését irányítják. A programot Java nyelven írtam JMonkeyEngine játékmotorral, a 3D modellezéshez pedig Blender programot használtam.

2. Felhasználói dokumentáció

2.1. Rendszer követelmények

2.1.1. Minimális hardver követelmény

Processzor:

Legalább két fizikai mag szükséges a szimuláció és a grafikus megjelenítés párhuzamos futtatásához.

Memória:

4 GB RAM

A szimulációs motor és a Java virtuális gép (JVM) futtatásához elegendő.

Tárhely:

500 MB szabad hely

A program, a modellek és a naplófájlok tárolásához szükséges.

Grafikus kártya:

Integrált grafikus kártya

OpenGL 2.0 támogatás szükséges a JMonkeyEngine futtatásához.

Képernyőfelbontás:

Legalább 1280x720 pixel

A felhasználói felület megfelelő megjelenítéséhez.

2.1.2. Szoftverkövetelmények

Operációs rendszer:

A program windows 10 rendszeren lett tesztelve, elméletileg más java kompatibilis operációs rendszeren is tud futni.

2.1.3. Java környezet

A program java alapú, ezért szükséges a java futtatókörnyezet vagy fejlesztői környezet telepítése. Ehhez útmutatót az oracle oldalán[1] lehet találni.

Java verzió:

Java Development Kit (JDK) 17 vagy újabb.

2.1.4. Könyvtárak és keretrendszerek

JMonkeyEngine 3.7:

A szimulációs motor, amely a 3D-s jelenetek kezeléséért és fizikai szimulációért felel.

JavaFX 17:

A grafikus felület megjelenítéséhez.

2.1.5. Hálózat

A program letöltéséhez és build elkészítéséhez internet kapcsolat szükséges.

2.1.6. Verziókezelés

A project verziókezeléséhez Git, a forráskód tárolásához GitHub használata.

2.2. Telepítés

A projekthez gradle-t használtam a telepítés és futtatás leegyszerűsítésére. A Gradle egy automatizált build eszköz, amely lehetővé teszi a projekt függőségeinek kezelését és a program egyszerű futtatását. A gradlew parancs a Gradle Wrapper-t használja, amely automatikusan letölti a szükséges Gradle verziót, így nincs szükség külön telepítésre. A zip fájl kicsomagolása után nyissunk egy windows parancssort. Navigáljunk el a kicsomagolt mappáig, és lépjünk bele a /RobotiCoffee mappába. Ezután

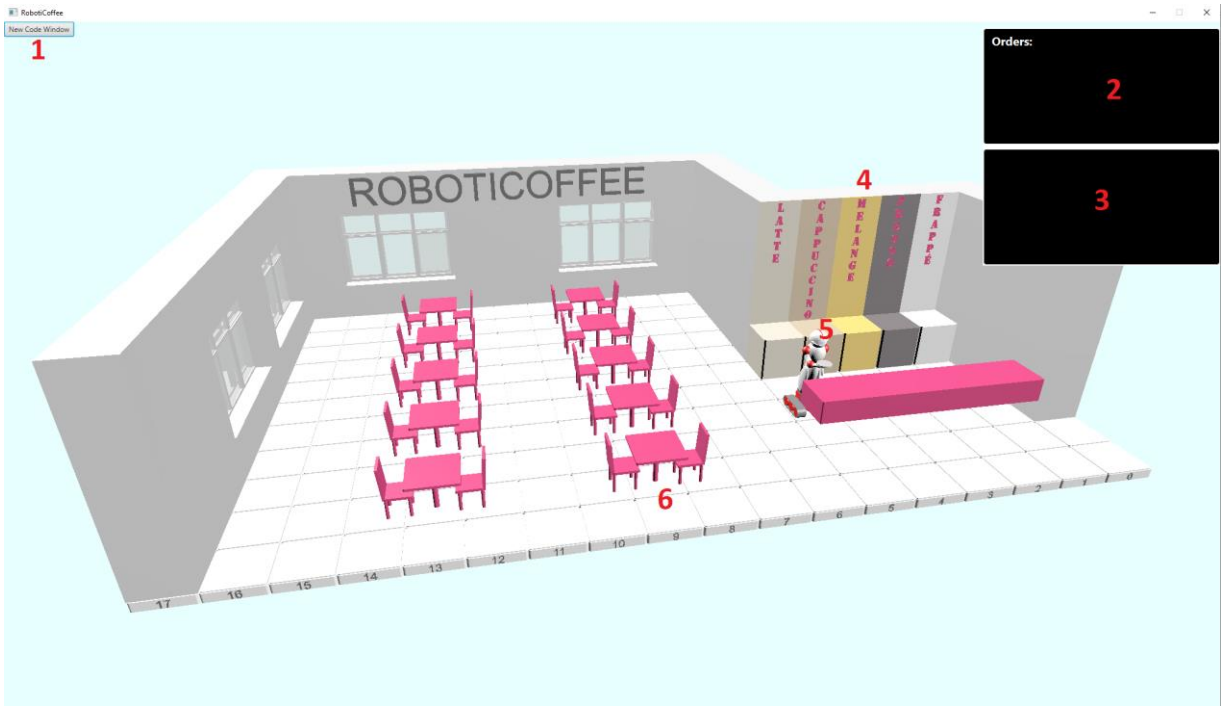
gradlew build

pranccsal megcsináljuk a buildet, majd a:

gradlew run

paranccsal már futtathatjuk is a programot, a gradle wrapper minden más függőséget letölt.

2.3. A játék felület

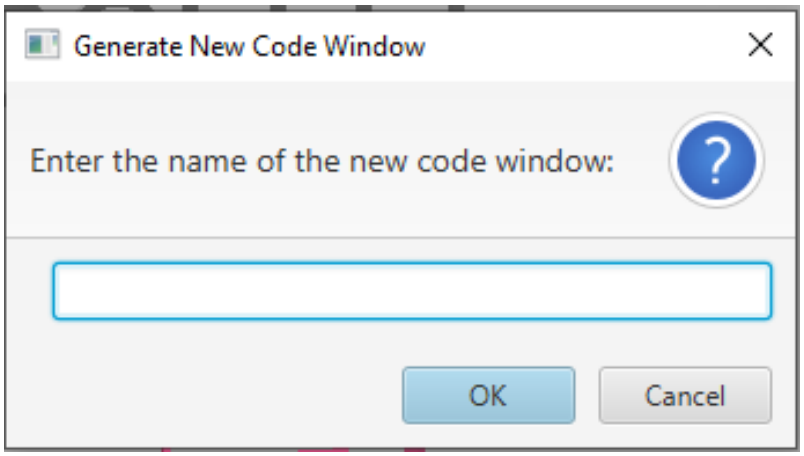


1. ábra: Játék felület

A játék során ezt a kávéházat kell üzemeltetnünk a robotunk által.

2.3.1. New Code Window

A gomb megnyomásával egy új programozó ablakot hozhatunk létre, először egy dialógus ablak jön elő.



2. ábra: Új kódablak létrehozása

Kötelező nevet adnunk az új ablaknak, továbbá, ha már hoztunk létre ablakot nem lehet egyező nevük, ez majd a későbbiekben lesz fontos.

Ha sikeres volt a generálás megjelenik az ablakunk.

3. ábra: A kódablak

1. Az ablakunk neve, aminek elneveztük. A szürke részen a Bal egérgombot nyomva tartva mozgathatjuk ezt az ablakot, a Jobb egérgomb nyomva tartása és egér mozgatásával pedig a méretét állíthatjuk az ablaknak.
2. Ide írhatjuk a programunkat, amit a robot szekvenciálisan végre fog hajtani.
3. A kódunkat az R (run) gombbal tudjuk futtatni.
4. Ha elindítottuk a P (pause) gombbal futás közben meg tudjuk állítani a futást, ilyenkor újra elérhetővé válik a futtató gomb, amivel folytathatjuk a program futást.

5. Az S (stop) gomb teljesen leállítja a futást, megnyomása után újra csak a run gomb lesz elérhető.

2.3.2. Orders ablak

Miután a robot felvesz egy rendelést itt fog megjelenni, egy rendelés tartalmazza milyen kávét és melyik asztalhoz kell kiszállítani. A rendelések sor adatszerkezettel vannak megoldva, így a legutolsó rendelés az aljára kerül, viszont nem vagyunk kötelesek tartani ezt a sorrendet a rendelések elkészítésénél, így ezek bármilyen sorrendben teljesíthetőek.

2.3.3. Console ablak

A konzol ablak eredetileg üres, a robottal kiíratott dolgok itt jelennek meg, továbbá, ha jól teljesítettünk egy rendelést az is ebben az ablakban látszik.

2.3.4. Kávéfajták

Egyrészt itt látszik a kávéboltunk választéka, másrészt a rendelés elkészítésénél is nagy szerepet játszik, a robotnak a kívánt kávéfajta elé kell állnia, hogy el tudja készíteni azt.

2.3.5. Robot

Ezt a robotot fogjuk irányítani kódunkkal, amit a kódablakokban írtunk. A robot kezében egy tálca van, amivel a rendeléseket tudja kiszállítani, de egyszerre csak egy kávé fér rá.

2.3.6. Asztalok

Ide ülnek le a vendégek miután felvettük tőlük a rendelést. Amíg vendég ül egy asztalnál addig azt foglalja, így, ha betelik minden asztalunk nem fog új vásárló érkezni, amíg valamelyik asztal fel nem szabadul. Az asztalok ezen felül akadályt is jelentenek a robotunk mozgásában. Amikor kiszállítunk egy rendelést, az asztal elé vagy mögé kell állnunk és az asztal felé kell fordulnunk ilyenkor tudjuk csak lerakni az asztalra a kávét. Az asztalok neve tartalmazza a pozíciójukat, a jobb alsó asztal például a „Table_9_1” mivel a 0-tól indexelt 9. oszlopban és 1. sorban van. A kávéház koordinátarendszere nem a hagyományos elrendezést követi. Az origó a jobb alsó sarokban található, és az X tengely értékei a negatív irányba növekednek, azaz balra haladva növekednek. A Z tengely a pozitív Y irányba mutat, így felfelé haladva növekszik.

4. ábra: Sorban álló emberek

2.3.7. Vásárlók

A vásárlók véletlenszerűen érkeznek, a pult előtt sorban állnak, amíg fel nem vesszük tőlük a rendelést, ilyenkor megjelenik az orders ablakban a rendelésük, elsétálnak egy asztalhoz ott leülnek, és addig várnak amíg meg nem kapják a kávéjukat. Miután kiszállítottuk a rendelést egy darabig még üldögélnek, majd távoznak.

2.4. Kódolás

2.4.1. Kulcsszavak

2.4.1.1. Alap programozási kulcsszavak

Azok a szavak, amik általában megtalálhatóak a programozási nyelvekben, ciklusok, típusok, bool értékek:

for, while, if, int, string, boolean, true, false

2.4.1.2. Mozgatásért felelős kulcsszavak

Mozgás és fordulás:

move, turn.

Fordulási irány lehet a 4 égtáj, jobbra balra vagy hátrafelé:

north, east, west, south, right, left, back.

2.4.1.3. Állapotlekérdező kulcsszavak

arePeopleWaiting, getRobotPosX, getRobotPosZ, getFirstOrderTableName, getFirstOrderCoffeeName

2.4.1.4. Rendelés

Rendelés felvétele, kávé elkészítése, kávé asztalra helyezése:

takeOrder, takeCoffee, placeCoffee

2.4.1.5. Konzolra írás

Korábban bemutatott konzol ablakra tudunk írni.

print

2.4.1.6. Programablak hívása

function

Azért kell elnevezni a kód ablakokat, mert ezek igazából függvények, amiket meg tudunk hívni más ablakokból, függvényhívás szerűen. A megfogalmazás nem egészen pontos, mivel paraméterátadásra nincsen lehetőség, így a rekurzív hívásokat is teljesen kizárjuk.

2.4.2. Operátorok

A programban három különböző típus létezik, ezek az egész szám, bool érték és a szöveg. Így az ezekre általában alkalmazott valamennyi operátor létezik a nyelvben.

2.4.2.1. Egész számokon végezhető műveletek

matematikai operátorok: +, -, *, /, %

értékadó operátorok: =, +=, -=, *=, /=

összehasonlító operátorok: ==, !=, <, >, <=, >=

incremeter, decrementer: ++, --

2.4.2.2. Logikai adattípuson végezhető műveletek

értékadás: =

összehasonlítás: ==, !=

logikai műveletek: &&, ||

2.4.2.3. Szöveg típuson végezhető műveletek

értékadás: =

összehasonlítás: ==, !=

konkatenáció: +

2.4.3. Szintaxis

Mint korábban említettem a programnyelv a Javahoz hasonlít, ez leginkább a szintaxisban jelenik meg. Deklaráció, értékadás, műveletek, függvényhívások, getterek után pontosvessző írása kötelező. A függvények, neve után nyitó és csukó zárójel használata.

2.4.3.1. Értékadás

A változónévvel később hivatkozhatunk a változónkra. A változókat egy scope stackben tároljuk, scope-on kívül létrehozott változókat látjuk scopeon belül, ez fordítva viszont nem igaz. Két változó neve egyezhet, ha eltérő scope mélységben vannak, ilyenkor a program jelenlegi scopehoz legközelebbi scopeban lévő változót használja. Új réteget nyitnak a ciklusok, elágazás, másik programablak hívása.

típus változónév = érték;

int szamertek = 10;

boolean logikaiertek = false;

string szoveg = ”Hello World!”;

2.4.3.2. Ciklusok, elágazás

A while ciklus egy logikai értéket vagy változót, vagy logikai értékre kiértékelhető kifejezést vár a feltételhez, a ciklus törzsét addig végzi amíg igaz a feltétel.

while(true) {

}

A for ciklus is hasonlóan működik, más programnyelvek for ciklusához, a ciklusfejben egy változót kell deklarálnunk, majd egy logikai értékre kiértékelhető kifejezést kell megadnunk végül egy kiértékelhető kifejezést. A for ciklus addig hajtja végre a ciklus belsejét amíg igaz a feltétel.

```
for(int i = 0; i<10; i++) {

}
```

Az if elágazés sem tér el a megszokott működéstől, logikai értékű kifejezés, ha igazra értékelődik lefut az elágazés belseje, ha nem akkor nem. Az egyedüli különbség, hogy else ág nem létezik, ez egy plusz nehézítő faktort ad a kód írásához.

```
if (1 == 1) {

}
```

2.4.3.3. Saját nyelvi elemek szintaxisa

A saját nyelvi elemeim is többnyire követik a Java szintaxisát, nyitó zárójel van a kulcsszó után, paraméter (ha van), csukó zárójel, pontosvessző, ha nem kifejezés része.

Egész számra kiértékelhető getter függvények, a robot aktuális pozíciójának lekérdezéséhez.

```
getRobotPosX();

getRobotPosZ();

if(getRobotPosX() == 5){

}
```

A rendelések sorában az első rendelés tulajdonságait lekérdező getter függvények, ezek szöveg típusra értékelődnek ki, a rendelésben lévő kávé nevét, és az asztal nevét adják vissza ahova tartozik az adott rendelés.

```
getFirstOrderCoffeeName();

getFirstOrderTableName();
```

A pult előtt sorban állhatnak emberek, ennek a lekérdezésére is van egy függvény. Ez logikai értéket ad vissza, igazat, ha van ember a pult előtt, különben hamisat.

arePeopleWaiting();

A konzolra íratáshoz lehet használni, paraméterül átadhatunk változót, kifejezést vagy konkrét értéket is, a konzol legaljára fűzi.

print(”Hello World!”);

print(10);

print(true!=false);

A robot mozgatásához két fő funkció tartozik, az egyikel előre lehet léptetni a robotot, itt nagy szerepet játszik, hogy merre fele néz a robot, a másik a fordulás, amivel ezt tudjuk szabályozni. A lépés metódusnál paraméterül vár egy számot, vagy számra értékelhető kifejezést, ennyi négyzetet fog lépni előre, ha tud. Ha bármilyen akadályba ütközik eközben, akkor is megpróbálja egyesével lelépni ezeket a lépéseket, de már nem fog a helyzetén változtatni, ezzel is ösztönözve a felhasználót, hogy ne adjon meg feleslegesen nagy számot lépésnek. A robot forogni a négy égtáj fele tud ezzel konkrétan megadva, hogy melyik irányba nézzen, vagy jobb, bal, hátrafele kódszavakkal a jelenlegi forgatáshoz képest dinamikusan is megadhatjuk, melyik irányba nézzen.

move(1+1);

turn(left);

turn(north);

A rendeléshez kötődő metódusok miatt szükséges, hogy figyelembe vegyük milyen irányba néz a robot, nem elég például a rendelés kiszállításához, hogy a robot megáll az asztal melleti mezőn, csak akkor tudja ténylegesen az asztalra helyezni a kávét, ha arra is fordul. Így ezeknek a függvények sikeres működésének előfeltétele a pozíció mellet a forgás is. Rendelést felvenni csak a pult mögül tudunk, az (5, 3) -as pozícióból, ez egyébként a kiindulási pozíciója a robotnak a játék indításánál. A kávékészítéshez a megfelelő munkaasztal elé állítva a robotot, és a munkaasztal elé forgatva el tudja készíteni a megadott

kávét. A rendelés kiszállításánál az asztalokat vagy felülről vagy alulról kell megközelíteni, és értelemszerűen az asztal fele forgatni a robotot ilyenkor tudja letenni a kezéből a kávét. Ha nem megfelelő a rendelés, akkor a robot nem teszi le és a kezében marad a kávé. A robot akkor is fel tud venni kávét a kávéfőzőkből, ha már van nála kávé, ilyenkor csak szimplán kicserélődik a kezében lévő kávé.

takeOrder(); csak a pultnál dél fele nézve

takeCoffee(); csak a kávéfőzőknél észak fele nézve

placeCoffee(); csak az asztaloknál az asztal fele nézve

A kódismétlés elkerülése érdekében, vagy a kód kisebb egységekbe szervezése miatt lehetőség van több program ablakot létrehozni, és a nevükkel meghivatkozni őket.

function masikAblakNeve;

2.5. Játék menete

A játékos feladata, hogy olyan programkódot írjon, amely képes a kávéház működését automatizálni. A robotnak a játékos által írt kód alapján kell végrehajtania a következő feladatokat:

Rendelések felvétele:

A robotnak fel kell vennie a sorban érkező vásárlók rendeléseit. A vásárlók különböző típusú kávékat rendelhetnek, amelyeket a robot pontosan rögzít.

Kávék elkészítése:

A robotnak a felvett rendelés alapján el kell készítenie a megfelelő típusú kávét.

Rendelés kiszállítása:

Miután a kávé elkészült, a robotnak ki kell szállítania azt a megfelelő asztalhoz, ahol a vásárló helyet foglalt. A robotnak pontosan kell azonosítania az asztalt, hogy a rendelés a megfelelő vendéghez kerüljön.

A robotnak minden rendelést pontosan kell teljesítenie, miközben a játékos közvetlen beavatkozása nélkül működik. Ez azt jelenti, hogy a játékosnak előre kell gondolkodnia, és

olyan algoritmust kell terveznie, amely képes kezelni a különböző helyzeteket. A játékos törekedhet arra, hogy a programja minél hatékonyabb legyen, ezt főleg a legrövidebb útvonalak használatával érheti el.

A játékos kódjának minősége és hatékonysága közvetlenül befolyásolja a játék sikerességét. A jól megírt program nemcsak a kávéház működését automatizálja, hanem lehetővé teszi, hogy a robot gyorsabban és több vásárlót szolgáljon ki, miközben minimalizálja a hibákat.

3. Fejlesztői dokumentáció

3.1. Megoldási terv

A RobotiCoffee alkalmazás fejlesztése során a hordozhatóság és a platformfüggetlenség érdekében a Java programnyelvet választottam, a Java Standard Edition 17 verzióját használva. A program fejlesztéséhez a JMonkeyEngine 3.7 játékmotort alkalmaztam, amely lehetővé teszi a 3D-s megjelenítést és a fizikai szimulációt. A projekt buildeléséhez és a függőségek kezeléséhez Gradle keretrendszert használtam, amely egyszerűsíti a telepítést és a futtatást. A felhasználói felület megvalósításához JavaFX 17-et alkalmaztam, amely modern és testreszabható grafikus elemeket biztosít.

A 3D-s modellek elkészítéséhez a Blender programot használtam, amely lehetővé tette az alacsony poligonszámú, optimalizált modellek létrehozását. A modelleket GLTF (.glb) formátumban exportáltam, amelyet a JMonkeyEngine natívan támogat. A modellek integrálása során figyeltem arra, hogy azok méretezése és forgatása pontosan illeszkedjen a játék világához.

A játékos által írt kód feldolgozásához egy saját fejlesztésű interpretert készítettem, amely három fő lépésben dolgozza fel a kódot: lexikális elemzés, szintaktikai elemzés és végrehajtás. A lexikális elemzéshez egy Lexer osztályt hoztam létre, amely a kódot tokenekre bontja. A szintaktikai elemzéshez egy Parser osztályt használtam, amely a tokenekből szintaxisfát (AST) generál. Az interpreter a szintaxisfa alapján hajtja végre a játékos által írt utasításokat, például a robot mozgatását vagy a rendelések kezelését.

A fejlesztés során Visual Studio Code fejlesztői környezetet használtam, amely integrált támogatást nyújtott a Gradle és a JavaFX számára. A kód minőségének biztosítása érdekében egységteszteket írtam a Lexer, Parser és Interpreter osztályokhoz, valamint integrációs teszteket végeztem a 3D-s megjelenítés és a felhasználói felület működésének ellenőrzésére.

A projekt során különös figyelmet fordítottam a kód modularitására és bővíthetőségére. Az alkalmazás különálló komponensekre (AppState-ekre) lett bontva, amelyek felelősek a különböző funkciókért, például a vásárlók kezeléséért, a robot irányításáért és a felhasználói felület megjelenítéséért. Ez az architektúra biztosítja a kód átláthatóságát és könnyű karbantarthatóságát, valamint lehetővé teszi új funkciók egyszerű hozzáadását a jövőben.

Az alkalmazás három fő rétegre osztható: **felhasználói felület (UI)**, **3D megjelenítés, interpreter**.

3.2. Felhasználói felület

A felhasználói felület kezeléséért a UIState osztály felelős, amely a JavaFX és a JMonkeyEngine közötti integrációt valósítja meg. Mivel a két technológia eltérő megjelenítő fákat használ (JavaFX a SceneGraph-ot, míg a JMonkeyEngine a Node-alapú renderelést), szükség volt egy áthidaló osztályra, amely lehetővé teszi a két rendszer együttműködését. Ez az összekötés a UIState osztályban került implementálásra. Bár elsőre macerásnak tűnhetett a JavaFX és a JMonkeyEngine integrációja, a megvalósítás gördülékenyen ment, és elengedhetetlen volt a program működéséhez. Más próbált csomagok nem tartalmaztak olyan funkciókat, amelyek lehetővé tették volna:

- A felhasználói felület dinamikus kezelését (pl. kódablakok létrehozása, mozgatása, átméretezése).
- Az egér- és billentyűzetesemények megfelelő kezelését.
- A 3D-s világ és a felhasználói felület zökkenőmentes együttműködését.

A JavaFX és a JMonkeyEngine kombinációja biztosította, hogy a program mind a 3D-s világ, mind a felhasználói felület szempontjából teljes funkcionalitással rendelkezzen.

3.2.1. CodeWindow

Az egyik fő eleme a játéknak a kódablak, amely egy egyedi vezérlő, és a játékos által írt kód megadására, szerkesztésére, valamint futtatására szolgál. A kódablakot a JavaFX technológia segítségével valósítottam meg, amely lehetővé teszi a modern, testreszabható grafikus elemek használatát.

A kódablak egyedi vezérlő, amely a következő elemekből áll:

Név megjelenítő (Label):

- A kódablak bal felső sarkában található, és az ablak nevét jeleníti meg.

- Ez a név egyedi az ablakok között, és a generátor osztály biztosítja, hogy ne lehessen két azonos nevű ablakot létrehozni.

Kódbeviteli terület (TextArea):

- A játékos által írt kód megadására szolgál.
- A TextArea kétirányban görgethető, így nagyobb kódok esetén is kényelmesen használható.
- A kódot a játékos itt írhatja meg, amelyet később az interpreter dolgoz fel.

Gombok (Button):

- Run: A kód futtatásáért felelős.
- Pause: A kód futtatásának szüneteltetésére szolgál.
- Stop: A kód futtatásának leállítására használható.

3.2.2. Kódablakok létrehozása és kezelése

A kódablakokat egy **generátor osztály** hozza létre, amely biztosítja a megfelelő működést és az egyedi névhasználatot.

Generátor osztály:

A generátor osztály felelős az új kódablakok létrehozásáért. A generátor bekéri a felhasználótól az ablak nevét, és ellenőrzi az inputot. Nem enged üres névvel ablakot létrehozni és nem enged két azonos nevű ablakot létrehozni sem. Ha az input megfelelő, létrehozza az új kódablakot, és hozzáadja a játék felületéhez.

Mozgathatóság és átméretezhetőség:

A JavaFX vezérlők nagy előnye, hogy a létrehozás után dinamikusan változtatható a vezérlő pozíciója és mérete. Ez lehetővé tette, hogy a kódablakok mozgathatók és átméretezhetők legyenek, ami jelentősen növeli a felhasználói élményt.

3.2.3. Order és konzol ablak

A RobotiCoffee alkalmazásban a rendelések megjelenítésére és a konzolra íratáshoz egyedi vezérlők kerültek kialakításra. Az order és konzol ablakok a játékos számára valós idejű visszajelzést nyújtanak a játék állapotáról és a kód futtatásának eredményeiről.

Az order ablak a játékos által felvett rendeléseket jeleníti meg. A rendelések egy sor adatszerkezetben vannak eltárolva, amely a rendeléseket FIFO (First In, First Out) sorrendben kezeli. Az order ablak tartalma automatikusan frissül, amikor, új rendelést vesz fel a robot vagy kiszállítja azt.

A konzol ablak a játékos által írt kód futtatásának eredményeit és a hibákat jeleníti meg. Ez a vezérlő többféle információt biztosít a játékos számára. Ha a robot sikeresen kiszállít egy rendelést, vagy a játékos kódja szintaktikai hibát tartalmaz az itt jelenik meg, továbbá a kiíratás (print) funkció eredménye is itt jelenik meg. Ezzel funkcióval bármilyen szöveget, számot, változót vagy kifejezést írhatunk ki a konzolra.

3.3. Három dimenziós megjelenítés

A RobotiCoffee alkalmazás háromdimenziós megjelenítését a JMonkeyEngine biztosítja, amely egy nyílt forráskódú játékmotor. Ez a motor lehetővé teszi a 3D-s objektumok renderelését, a világ felépítését, valamint a játékos által írt kód hatásainak megjelenítését a 3D-s környezetben. A háromdimenziós megjelenítés kulcsfontosságú eleme a játéknak, mivel a robot mozgása, az asztalok és a vásárlók interakciói mind vizuálisan követhetők. A világ megalkotására három AppState osztály áll rendelkezésre.

3.3.1. CreateSceneState

A CreateSceneState osztály felelős a játék 3D-s világának alapvető felépítéséért. Ez az osztály inicializálja a világot, beállítja a kamera látószögét és pozícióját, a világítást, amit két irányított fényforrás ad, és betölti a kávéház modelljét.

3.3.2. PeopleState

A PeopleState osztály felelős a játék világában található vásárlók és asztalok kezeléséért. Ez az osztály biztosítja, hogy a vásárlók megjelenjenek, rendeléseket adjanak le, és interakcióba lépjenek a robot által vezérelt játékmenettel. Az osztály emellett a vásárlók mozgását és sorba állítását is megvalósítja. Memória takarékosság érdekében a modelleket csak egyszer tölti be, utána klónokat hoz létre belőle, ez biztosítja, hogy ne pazaroljunk erőforrást az újboli betöltésre és tárolásra. A mozgások megvalósításáról az AppState-ből leszármazó update metódusban valósítjuk meg, így szakadásmentes, folyamatos mozgást tudunk imitálni.

3.3.2.1. A vásárlók életciklusa

A vásárlók véletlenszerű időközönként érkezenek, ha van üres asztal és van hely a sorban állashoz, ilyenkor sorba állnak, és várják, hogy ők következzenek. A robot rendelést csak a soron következő embertől tud felvenni, ezután az ember elindul egy általa kiválasztott asztal felé, ahol leül, ilyenkor a sor frissül, a sorban álló vásárlók észreveszik, hogy nincs előttük más és előre lépnek a sorban. Miután leültek türelmesen várják rendelésüket, ha megkapták adott ideig várnak, mintha meginnák a kávét, majd kisétálnak a bejárat felé.

3.3.3. RobotState

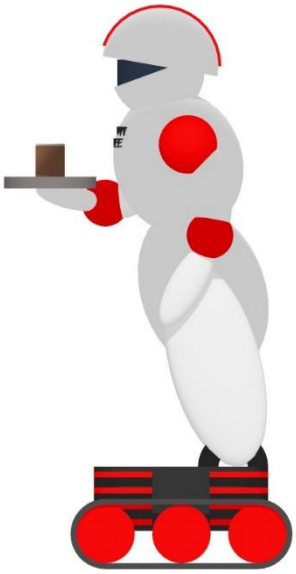
A RobotState osztály a RobotiCoffee alkalmazás egyik legfontosabb osztálya, mivel ez felelős a robot mozgásáért, forgásáért, a rendelések felvételéért és kiszállításáért, valamint a robot állapotának kezeléséért. A robot a játékos által írt kód alapján hajtja végre a műveleteket, amelyek valós időben jelennek meg a 3D-s világban. Ez az osztály biztosítja, hogy a robot viselkedése a játékmenet szempontjából releváns és valósághű legyen. A robot egyik fő funkciója a mozgás. A robot minden lépésénél ellenőrizzük, hogy érvényes lépést akar-e tenni, ebben a Rectangle osztály segít. A kávézó területét így le tudjuk fedni téglalapokkal, ezeken belül vannak csak érvényes lépések, majd ebből szintén téglalapokat tudunk kivágni, például az asztalok alapterületét, ahova nem engedjük a robotot lépni. A robot nem ismeri fel, ha akadályba ütközött, vagy elérte a pálya szélét, megpróbálja lelépni az összes fennmaradó lépését. A fordulás funkció is elengedhetetlen a pálya bejárása miatt is, de vannak szituációk, amikor a robotnak kötelező egy adott irányba néznie, hogy el tudjon végezni bizonyos műveleteket. A robot egy felsorolás típussal tárolja éppen mi van a tálcáján, ez a modelljében is látszik, ha van valamilyen kávé nála az megjelenik a tálcáján.

3.4. Három dimenziós modellek

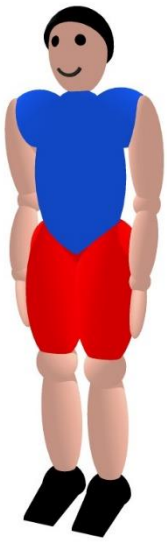
A RobotiCoffee alkalmazásban használt háromdimenziós modellek mind kézzel készültek a Blender program segítségével. A Blender egy nyílt forráskódú 3D modellező eszköz, amely lehetővé teszi a részletes és testreszabott modellek létrehozását. Ezek a modellek a játék világának vizuális elemeit alkotják, például a robotot, az asztalokat, a vásárlókat és a bolt környezetét. A modellek tervezése során különös figyelmet fordítottam arra, hogy azok illeszkedjenek a játék egyszerű, letisztult stílusához. A játék teljesítményének biztosítása érdekében a modellek alacsony poligonszámúak, így a renderelés gyors és hatékony marad. A kész modelleket GLTF (.glb) formátumban exportáltam, mivel ezt a formátumot a

JMonkeyEngine natívan támogatja, és könnyen integrálható a játékba. Az exportálás során ügyeltem arra, hogy a modellek méretezése és forgatása pontosan illeszkedjen a játék világához, így a megjelenítés zökkenőmentes és vizuálisan megfelelő legyen. A képek még a modell tervezési fázisában készültek.

5. ábra: A robot kiszállítja a rendelést



6. ábra: Robot oldalnézetben



7. ábra: Álló férfi model

8. ábra: Álló női model

3.5. A kód feldolgozása

A RobotiCoffee alkalmazásban a játékos által írt kód feldolgozása több lépésben történik, amelyeket különböző osztályok és komponensek valósítanak meg. A feldolgozás célja, hogy a játékos által megadott utasításokat értelmezze és végrehajtsa a játék világában, például a robot mozgatását vagy a rendelések kezelését. A kód feldolgozása három fő szakaszra osztható: **lexikális elemzés, szintaktikai elemzés, és végrehajtás**.

3.5.1. Lexikális elemzés

A lexikális elemzés során a játékos által írt kódot kisebb egységekre, úgynevezett tokenekre bontjuk. Ez a folyamat a Lexer osztályban történik, amely a kódot karakterenként olvassa be, és azonosítja a különböző tokeneket, például számokat, szövegeket, operátorokat vagy kulcsszavakat.

A tokenek típusait a TokenType enum definiálja, amely a következő kategóriákat tartalmazza:

- **NUMBER**: Számokat reprezentál.
- **STRING**: Szöveges értékeket reprezentál.
- **BOOLEAN**: Logikai értékeket (true/false) reprezentál.
- **IDENTIFIER**: Azonosítókat, például változóneveket reprezentál.
- **OPERATOR**: Matematikai vagy logikai operátorokat (pl. +, -, *, /).
- **KEYWORD**: Kulcsszavakat (pl. if, while, print).
- **OPEN_PAREN**, **CLOSE_PAREN**: Nyitó és záró zárójelek.
- **OPEN_BRACE**, **CLOSE_BRACE**: Nyitó és záró kapcsos zárójelek.
- **OPEN_BRACKET**, **CLOSE_BRACKET**: Nyitó és záró szögletes zárójelek.
- **SEMICOLON**: Pontosvesszők.
- **COMMA**: Vesszők.
- **UNKNOWN**: Ismeretlen karakterek.

A lexikális elemző kimenete a feldolgozott tokenek listája.

3.5.1.1. Példa

A szövegmezőben megadott programkód:

move(3);

ebből tokeneket generál a parser:

KEYWORD{ „move” }

OPEN_PAREN{ „(” }

NUMBER{ „3” }

CLOSE_PAREN{ „)” }

SEMICOLON{ „;” }

3.5.2. Szintaktikai elemzés

A szintaktikai elemzés során a lexikális elemzés által generált tokenekből egy **szintaxisfát (AST)** hozunk létre, amely a kód szerkezetét reprezentálja. Ez a folyamat a **Parser** osztályban történik, amely a tokenek sorrendjét és típusát vizsgálja, hogy ellenőrizze, megfelelnek-e a programnyelv szintaktikai szabályainak. A szintaxisfa egy hierarchikus adatstruktúra, amely **Node** típusú csomópontokból áll. Minden parancshoz létezik egy specifikus Node osztály, amelyek a közös **Node** ősosztályból származnak le.

3.5.2.1. ProgramNode

Az AST fa gyökerét a ProgramNode adja, ide kerül be egy listába minden más utasítás.

3.5.2.2. Utasítások feldolgozása

A parseStatement metódus a tokenek típusa alapján dönti el, hogy milyen csomópontot (Node) hozzon létre. Ez a metódus a tokenek aktuális értékét és típusát vizsgálja, majd a megfelelő parse metódust hívja meg, amely az adott utasítás vagy kifejezés szintaktikáját ellenőrzi, és létrehozza a megfelelő Node objektumot. A parse metódusok moduláris felépítésűek, így minden utasítás vagy kifejezés feldolgozására külön metódus felel, ami növeli a kód olvashatóságát és karbantarthatóságát.

Ezek a metódusok bizonyos esetekben egymást is hívhatják, például amikor egy összetett kifejezés vagy utasítás több részből áll. Például a while ciklus feldolgozását végző parseWhileStatement metódus a ciklus feltételének feldolgozásához ugyanazt a parseCondition metódust használja, mint az if elágazás fejlécének feldolgozásakor. Ez a megközelítés biztosítja, hogy a kód újra felhasználható legyen, és a feltételek feldolgozása konzisztens módon történjen.

A parse metódusok nemcsak a szintaktikai szabályokat ellenőrzik, hanem a tokenek sorrendjét és típusát is, hogy biztosítsák a program helyességét. Ha a tokenek nem felelnek meg a programnyelv szintaktikai szabályainak, a metódus kivételt dob (InterpreterException), amely tartalmazza a hiba helyét és leírását. Ez a hibakezelés segít a felhasználónak gyorsan azonosítani és kijavítani a kód hibáit.

A moduláris felépítés lehetővé teszi, hogy a különböző utasítások és kifejezések feldolgozása egymástól függetlenül történjen, miközben a közös elemek (például feltételek vagy kifejezések) feldolgozása egységes marad. Ez a megközelítés nemcsak a kód olvashatóságát és karbantarthatóságát növeli, hanem a bővíthetőséget is, mivel új utasítások vagy kifejezések könnyen hozzáadhatók a meglévő rendszerhez. A parser kimenete a szintaxisfa, ennek a gyökérelemét adja tovább.

3.5.2.3. Kifejezések feldolgozása

A kifejezéseket a parseExpression metódus dolgozza fel, amely támogatja az aritmetikai, logikai és összehasonlító műveleteket. A kifejezések műveleti precedenciáját a parseExpressionWithPrecedence metódus kezeli, ez biztosítja, hogy a kifejezés jó sorrendben értékelődjön ki.

3.5.2.4. Példa

A lexer példáját folyatva:

A lexer által generált tokenekből a parser egy szintaxisfát (AST) hoz létre. A példában szereplő move(3); utasítás feldolgozása során a parser a következő lépéseket hajtja végre:

A parser felismeri, hogy az első token egy KEYWORD, amely a move parancsot tartalmazza. Ennek alapján meghívja a parseMoveStatement metódust.

A parseMoveStatement metódus először ellenőrzi, hogy a következő token a move kulcsszó, majd „elnyeli” azt. Ezután ellenőrzi, hogy a következő token egy nyitó zárójel (OPEN_PAREN), és ezt is „elnyeli”.

A metódus ezután egy számot tartalmazó tokent vár, amelyből egy NumberNode objektumot hoz létre. Ez a csomópont reprezentálja a move parancs argumentumát.

A metódus ellenőrzi a záró zárójelet (CLOSE_PAREN) és a pontosvesszőt (SEMICOLON), hogy biztosítsa a szintaxis helyességét. Ha bármelyik ellenőrzés során nem a várt token található, a parser kivételt dob (InterpreterException).

A sikeres feldolgozás után a parser létrehoz egy MoveStatementNode objektumot, amely tartalmazza a NumberNode-ot, mint argumentumot.

A végeredmény egy szintaxisfa, amely így néz ki:

ProgramNode

- MoveStatementNode
    - NumberNode: 3

Ez a szintaxisfa reprezentálja a move(3); utasítást, és a parser ezt adja tovább az interpreter számára, amely végrehajtja a parancsot a játék világában.

3.5.3. Interpreter

Az interpreter bemenete egy **ProgramNode**, amely a szintaxisfa gyökéreleme, és tartalmazza a program összes utasítását. Az interpreter ennek segítségével képes bejárni az egész szintaxisfát, és a csomópontok típusának megfelelően értelmezni vagy végrehajtani azokat. A ProgramNode feldolgozása során az interpreter rekurzívan dolgozza fel a benne található utasításokat, és minden csomópontot a típusának megfelelő metódussal kezel. Ha egy csomópont például egy kifejezést tartalmaz, az interpreter kiértékeli azt, és visszaadja az eredményt. Ha a csomópont egy olyan utasítást reprezentál, amely mellékhatással jár, például a robot mozgatása vagy egy üzenet kiírása a konzolra, akkor az interpreter végrehajtja a csomópont által meghatározott műveletet.

A csomópontok feldolgozása során az interpreter figyelembe veszi a program logikáját és a szintaktikai szabályokat. Például egy while ciklus esetén az interpreter először kiértékeli a ciklus feltételét, majd addig hajtja végre a ciklus törzsét, amíg a feltétel igaz. Hasonlóképpen, egy if utasítás esetén az interpreter kiértékeli a feltételt, és ennek eredményétől függően hajtja végre az if törzsét. Az interpreter a változók kezelésére külön scope-okat használ, amelyek lehetővé teszik a változók lokális és globális szintű kezelését. A változók értékének

beállítása, módosítása és lekérdezése a scope-ok hierarchiájában történik, így biztosítva a változók megfelelő láthatóságát és érvényességét.

Az interpreter működése során a kód logikai szerkezetét követve biztosítja, hogy a program helyesen kerüljön végrehajtásra. A csomópontok típusától függően az interpreter különböző műveleteket hajt végre, például aritmetikai műveleteket számokkal, logikai műveleteket feltételekkel, vagy szöveges műveleteket stringekkel. Ha a program hibát tartalmaz, például egy nem létező változóra történik hivatkozás, az interpreter kivételt dob, amely tartalmazza a hiba helyét és leírását. Ez a megközelítés lehetővé teszi, hogy a játékos által írt kód hatásai valós időben megjelenjenek a játék világában, miközben a program logikája és működése átlátható és könnyen követhető marad.

3.5.3.1. Példa

A korábbi példákat folytatva:

A korábbi példákat folytatva, az interpreter bemenete a ProgramNode, amely tartalmazza a move(3); utasítást reprezentáló szintaxisfát. A példában szereplő program node-ok zárójelezett formája:

ProgramNode (MoveStatementNode (NumberNode (3)))

Az interpreter a ProgramNode feldolgozása során bejárja annak csomópontjait, és a MoveStatementNode típusú csomópontot találja.

A MoveStatementNode feldolgozásakor az interpreter kiértékeli annak argumentumát, amely egy NumberNode típusú csomópont. A NumberNode értéke 3, így az interpreter ezt az értéket visszaadja a MoveStatementNode-nak. Ezután az interpreter végrehajtja a move parancsot, amely a robotot három lépéssel előre mozgatja a játék világában.

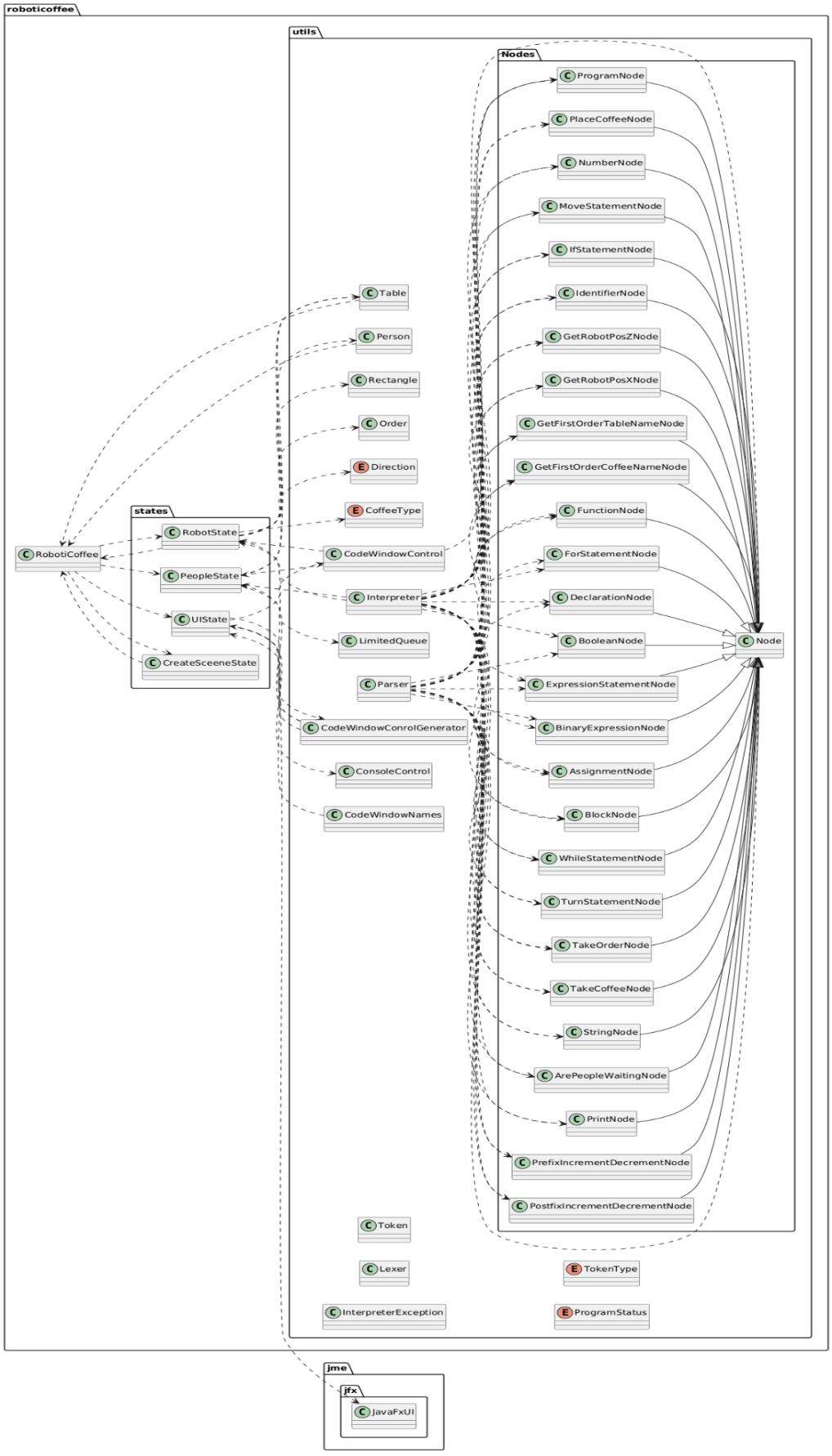
A folyamat során, ha bármelyik csomópont feldolgozása közben hiba történik (például egy hiányzó vagy érvénytelen argumentum miatt), az interpreter kivételt dob, amely tartalmazza a hiba helyét és leírását. Ebben az esetben a hibaüzenet a konzolon jelenik meg, segítve a játékost a probléma azonosításában és kijavításában.

A példában a move(3); utasítás sikeresen végrehajtódik, és a robot három lépéssel előre mozdul a játék világában, miközben az interpreter biztosítja a program logikai szerkezetének helyes értelmezését és végrehajtását.

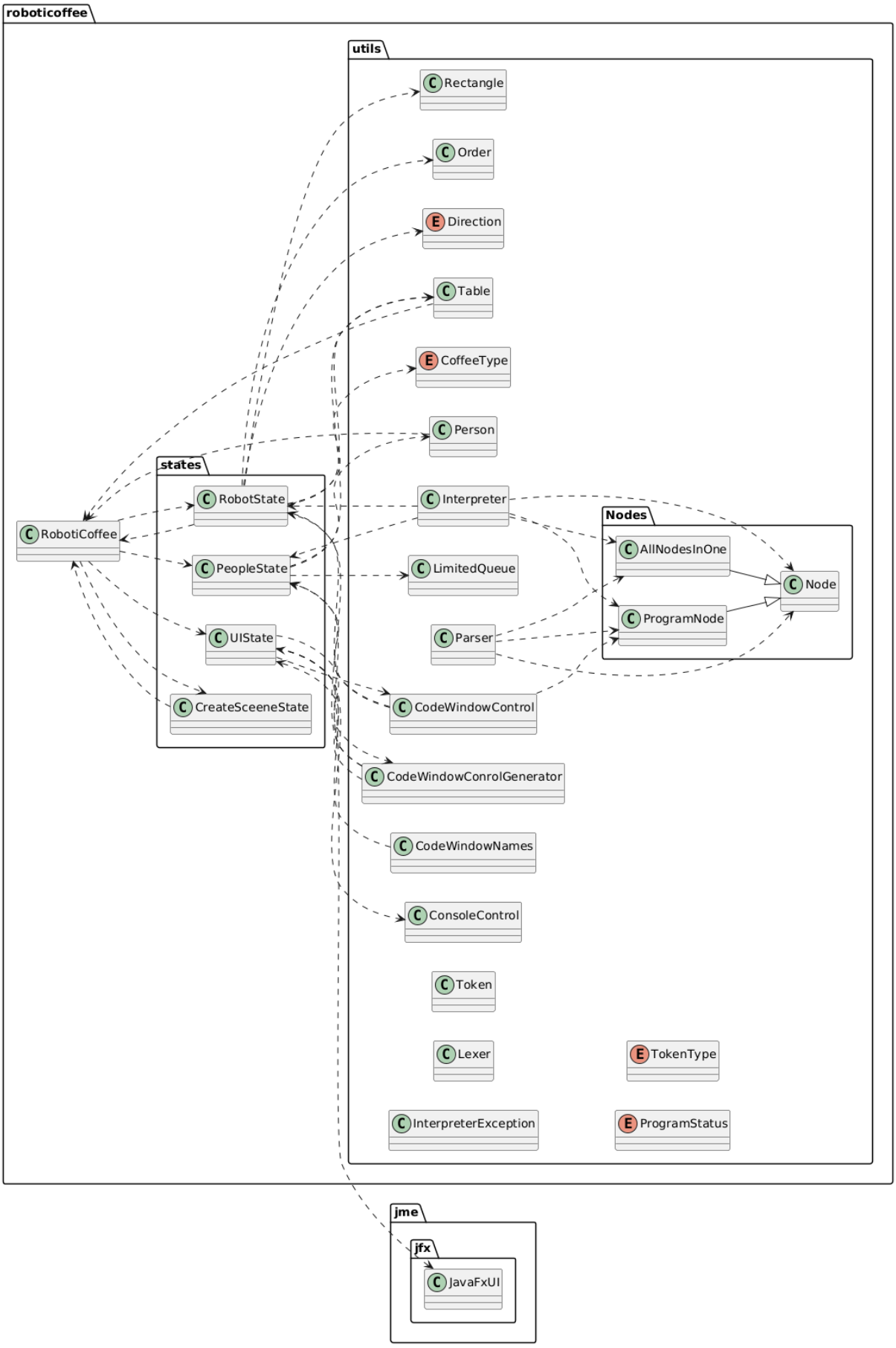
3.6. Osztályok, osztálydiagramok

A program teljes osztálydriagramja, elég kusza és így átláthatatlan, ezt főleg a Nodeból leszármazó specifikus Node osztályok okozzák, így készítettem egy olyan diagramot is, ahol ezeket a nodeokat egy nodeként ábrázolom, így jobban látszanak az osztályok közötti kapcsolatok. A kódban két különböző Node típus található, amelyek eltérő funkciókat látnak el, az egyik a JMonkeyEngine jelenetfájának a Node típusa. Ezek a Nodeok, általában rootNode, vagy localRootNode-nak elnevezett változók a 3D-s megjelenítésért felelnek. A másik Node típus az absztrakt szintaxisfa (AST) csomópontjait reprezentálja, amelyeket a Parser osztály hoz létre a játékos által írt kód feldolgozása során. Ezek a csomópontok a program logikai struktúráját tartalmazzák, például utasításokat (MoveStatementNode, IfStatementNode) vagy kifejezéseket.

Az osztályoknál csak a legfontosabb változókat és metódusokat fejtem ki.

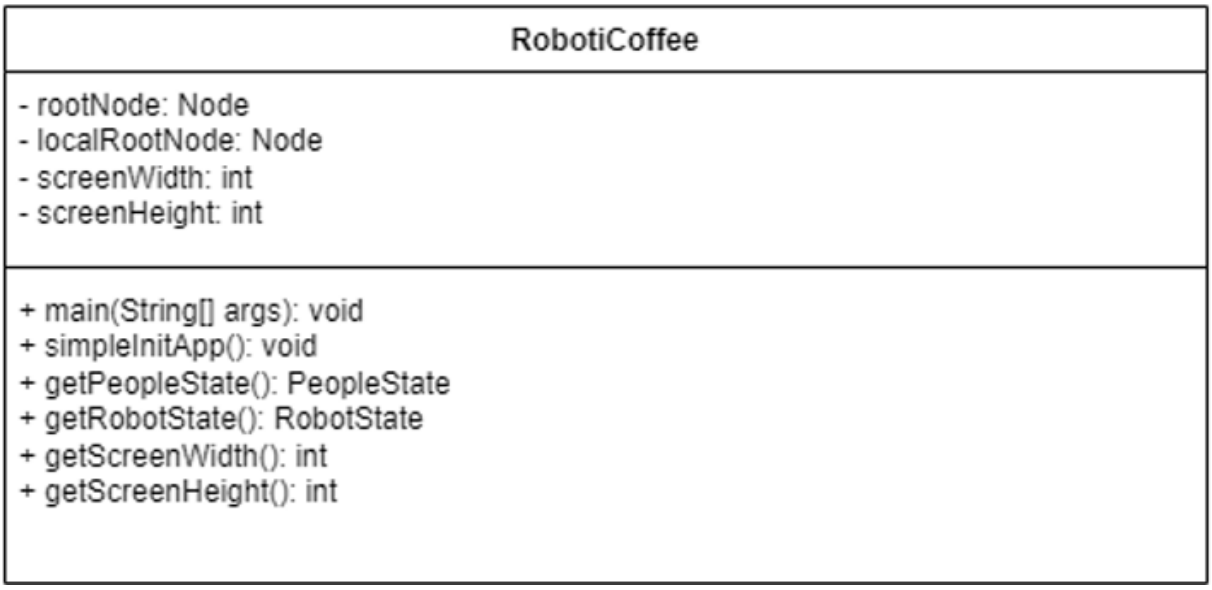


9. ábra: A teljes UML diagram



10. ábra: Uml diagram a specifikus Node osztályok nélkül

3.6.1. Roboticoffee



11. ábra: RobotiCoffee

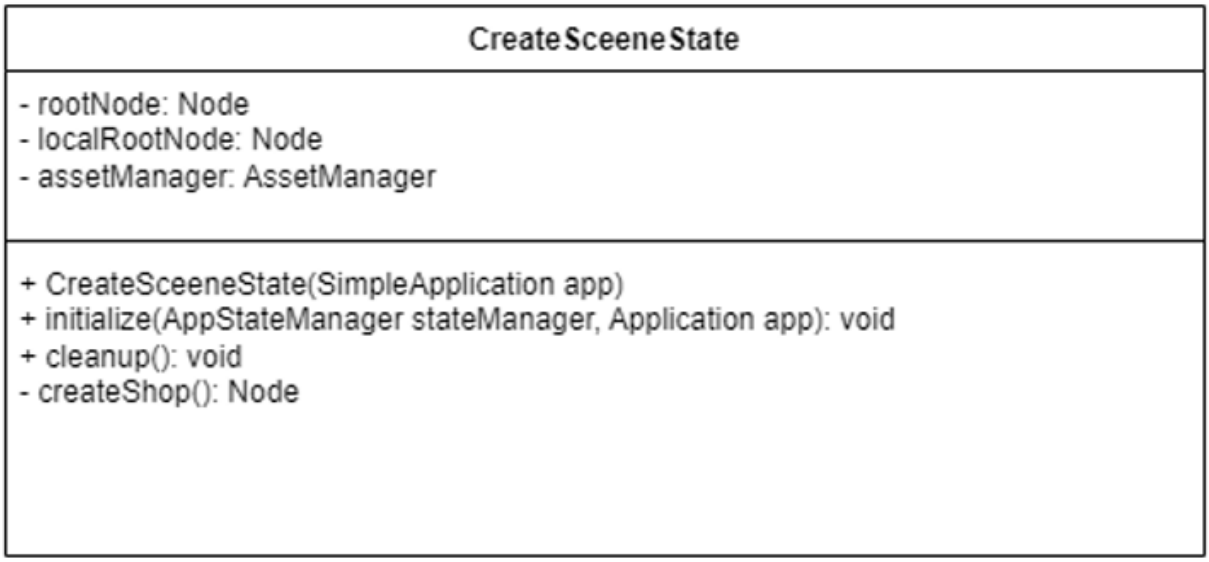
rootNode, localRootNode: jme3 jelenetfa Nodejai

screenWidth, screenHeight: A programablak méretei, és ehhez tartozó getterek.

main(): A program belépési pontja.

simpleInitApp(): A JMonkeyEngine inicializálása, State osztályok hozzákötése.

3.6.2. CreateSceneState



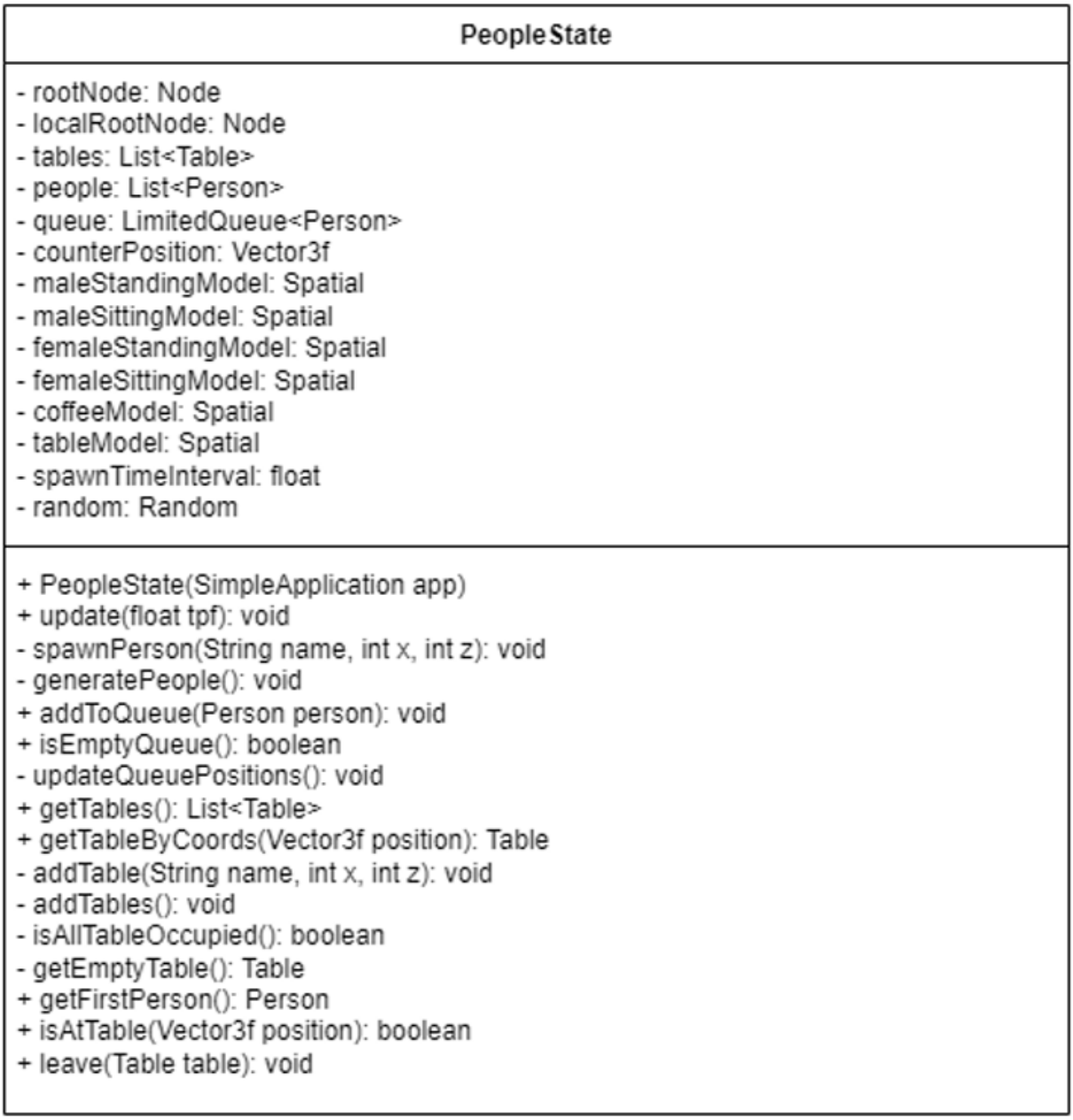
12. ábra: CreateSceneState

rootNode, localRootNode: jme3 jelenetfa Nodejai

initialize(): felépíti a 3D jelentet, fények, kamera pozíció

createShop(): betölti a kávéház modelljét

3.6.3. PeopleState



13. ábra: PeopleState

rootNode, localRootNode: jme3 jelenetfa Nodejai

tables: az asztalokat tárolja

people: a jelenetben lévő embereket tárolja

queue: az emberek sorbanállását szimulálja

counterPosition: constans a pult helyzetére

Model-ek: betöltött 3d modellek, amikből klónok jönnek létre

spawnPerson(): létrehoz egy embert

generatePeople(): az emberek létrehozását kezeli, a spawnPerson()-t hívja időközönként

updateQueuePositions(): frissíti az emberek sorban elfoglalt helyét

addTables(): létrehozza az asztalokat

getFirstPerson(): visszaadja a soron következő embert, a queue első elemét

isAtTable(): a paraméterben megadott pozíción áll-e asztal

3.6.4. RobotState

14. ábra: RobotState

robotWithCoffeeNode, robotCoffeeNode: a robot model kávéval a tálcán, a másik kávé nélkül

robotDirection: égtájakat tartalmazó felsoroló típus

validAreas: téglalapok listája, ami meghatározza melyik koordinákon mozoghat csak a robot

obstacles: téglalapok listája, amivel letiltjuk pozíciókról a robotot, például az asztalok alapterülete

inHand: felsoroló típus a kávéfajtákra, ha null akkor üres a tálca

orders: a rendelések listája

onPrint, onOrderChanged: az order és konzol ablakok frissítésére szolgáló futtatható változók

turn(): a robot forgatása

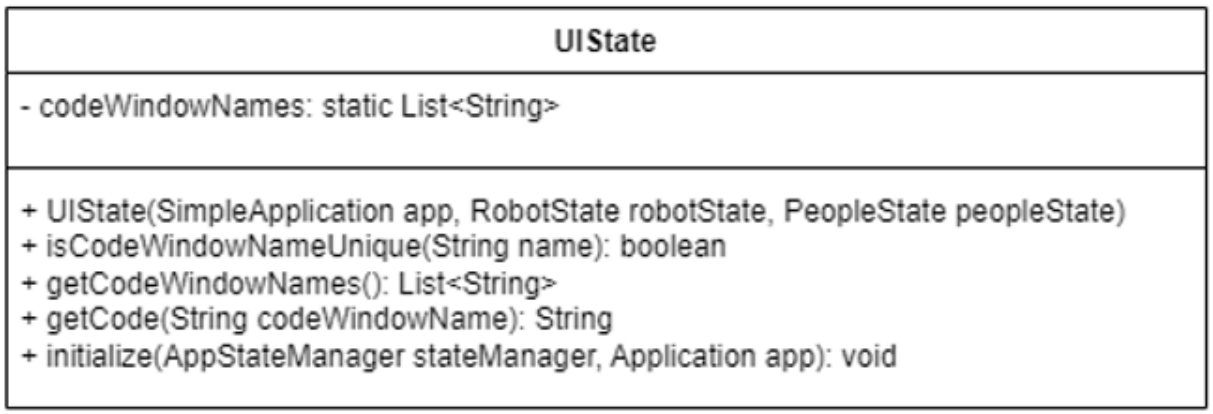
move(): a robot mozgatása

getHeadPosition(): a robot előtti pozíció, ez a robat forgatásánál változik

arePeopleWaiting(): igaz, ha emberek várakoznak a pultnál

getOrdersString(): a rendelések listáját szöveggé alakítja

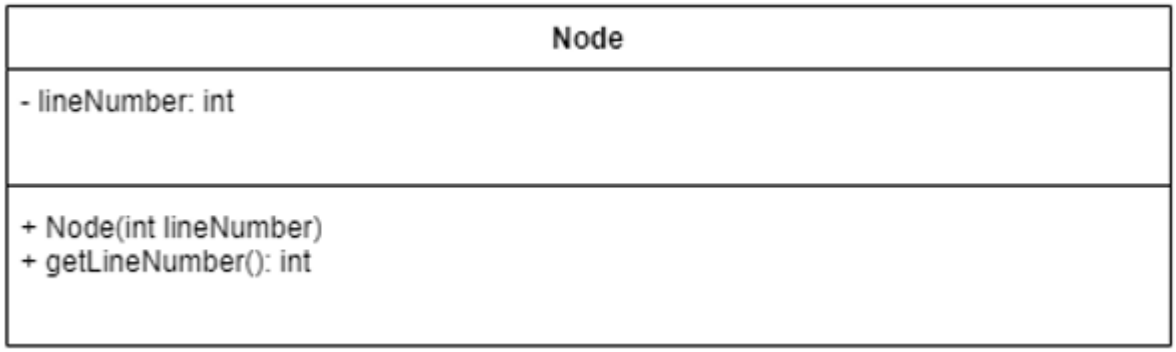
3.6.5. UIState



15. ábra: UIState

codeWindowNames: programablakok nevét tároló lista

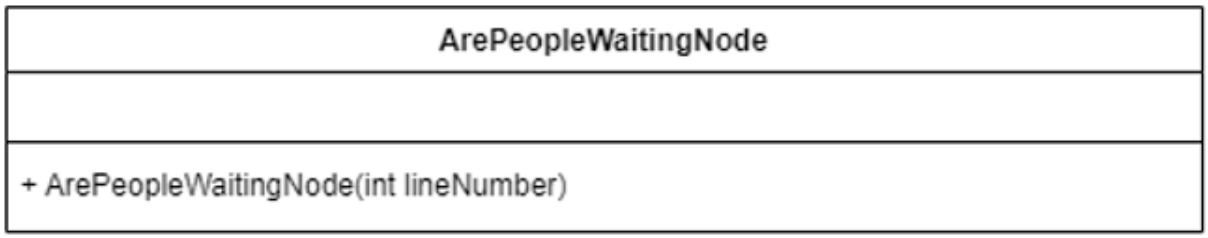
3.6.6. Node osztályok



16. ábra: Node

Minden *Node osztály ebből származik le így mindegyik tartalmazza a lineNumber változó, ami a kódablakban elfoglalt helyét tárolja

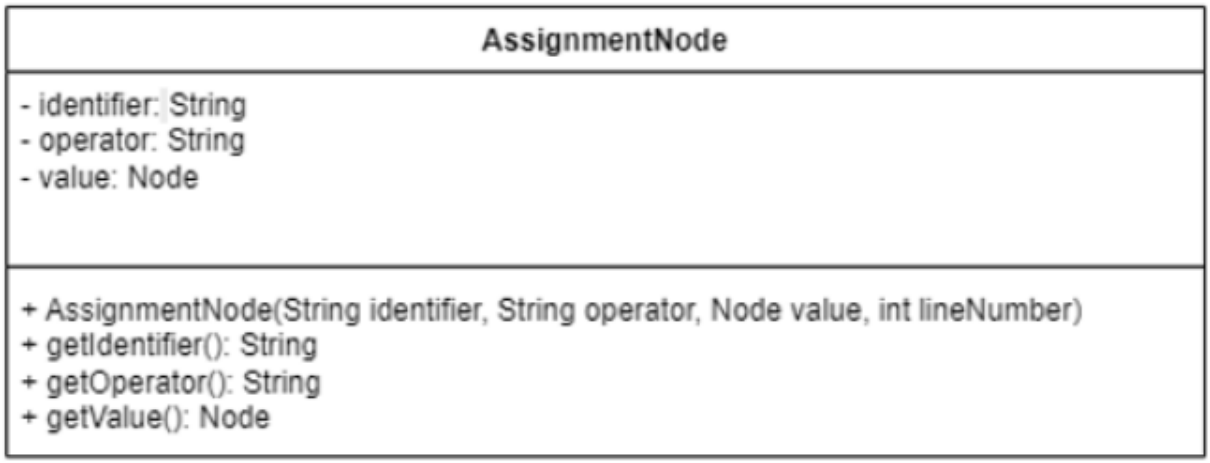
3.6.6.1. ArePeopleWaiting



17. ábra: ArePeopleWaitingNode

A sorban álló emberek lekérdezéséhez használt függvényhívást eltároló Node.

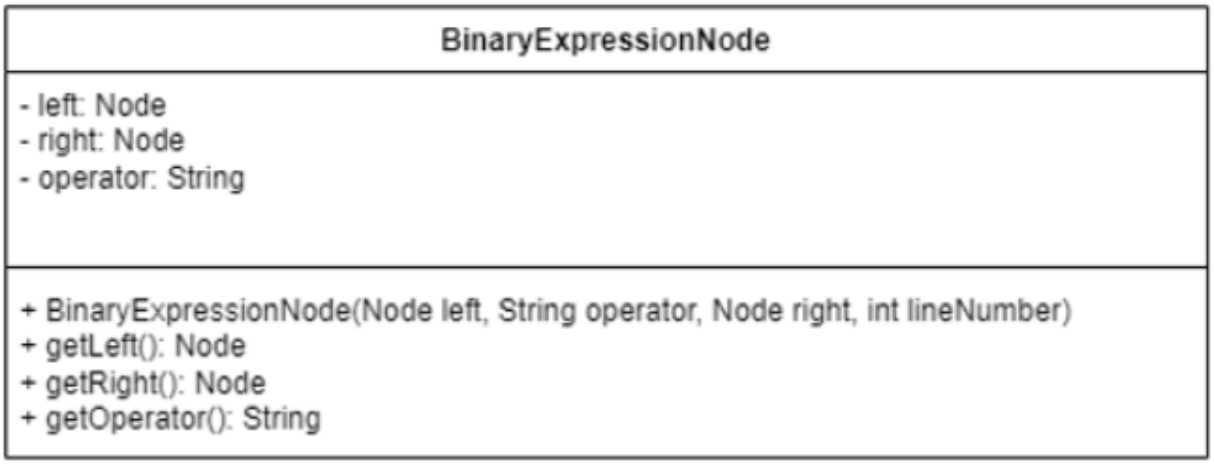
3.6.6.2. AssignmentNode



18. ábra: AssignmentNode

Értékadást tároló csomópont: i = 2

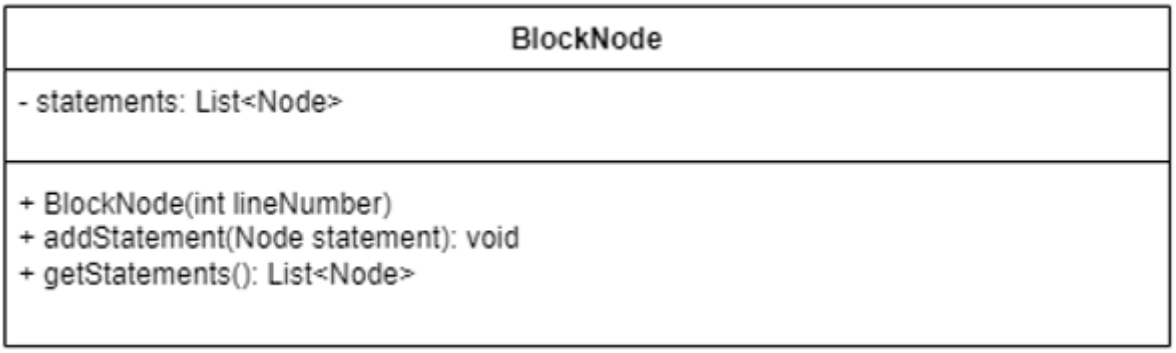
3.6.6.3. BinaryExpressionNode



19. ábra: BinaryExpressionNode

Két operandusú kifejezéseket és az operátort tárolja: 3 * 5

3.6.6.4. BlockNode



20. ábra: BlockNode

Egy programtömböt tárol:

{

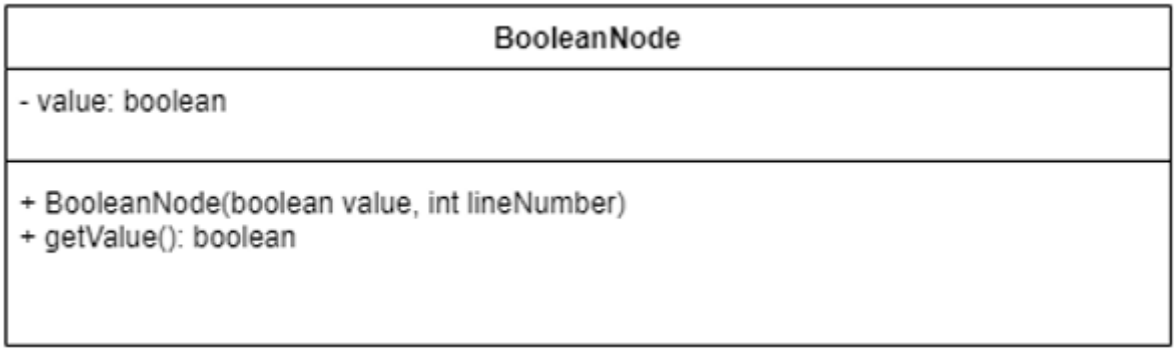
utasítás1

utasítás2

…

}

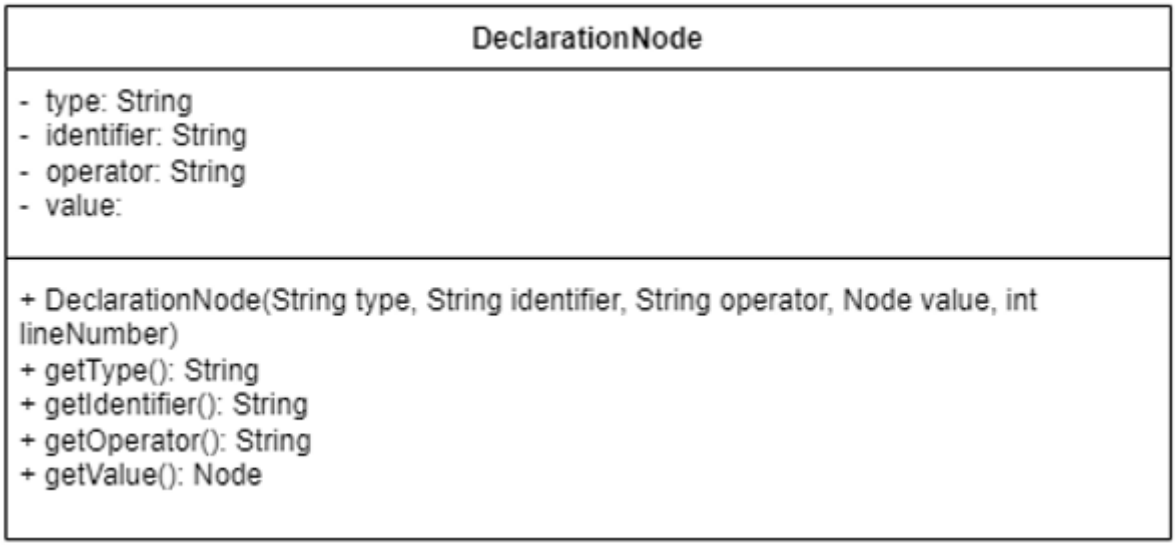
3.6.6.5. BooleanNode



21. ábra: BooleanNode

Logikai értéket tárol: true, false

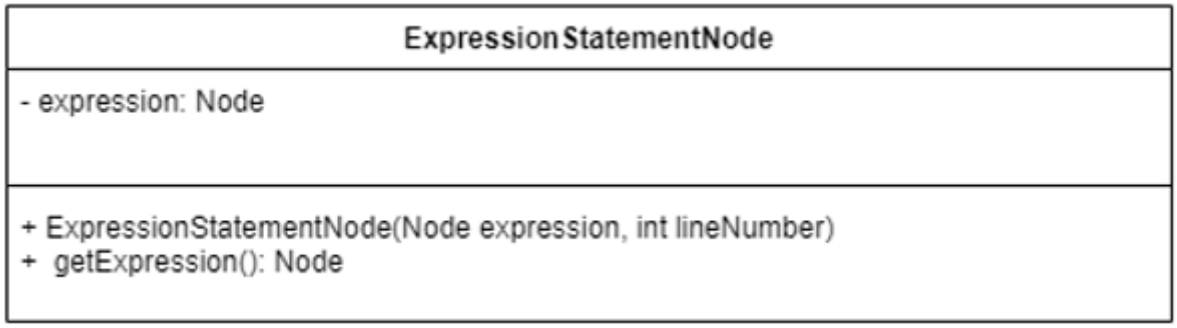
3.6.6.6. DeclarationNode



22. ábra: DeclarationNode

Változó deklarációt tárol: int változó_név = 11

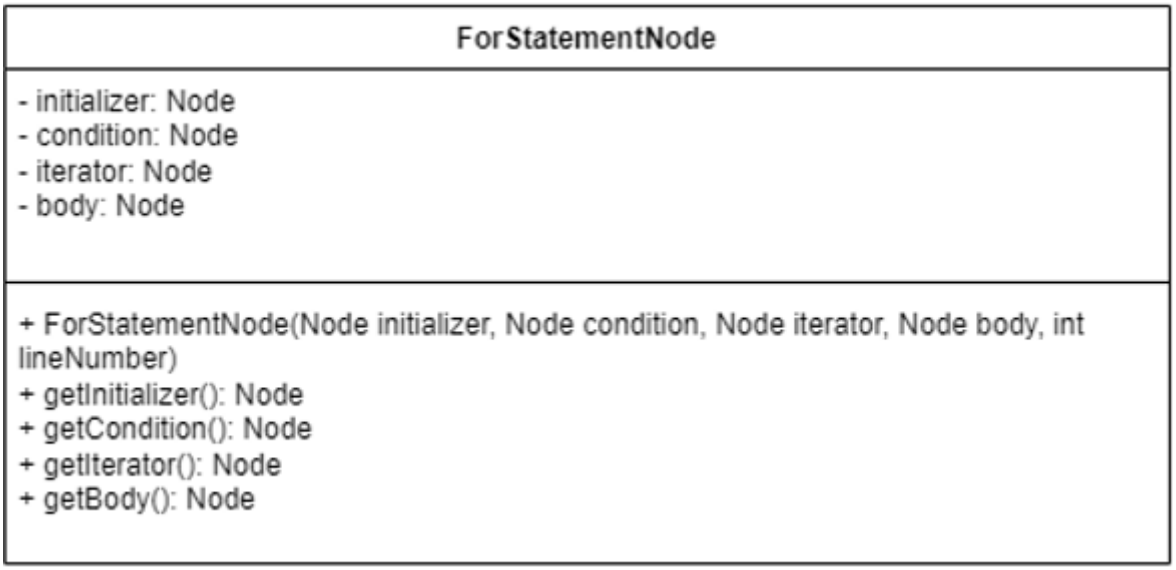
3.6.6.7. ExpressionStatementNode



23. ábra: ExpressionStatementNode

Egy kifejezést tárol: (2+4)*3

3.6.6.8. ForStatementNode

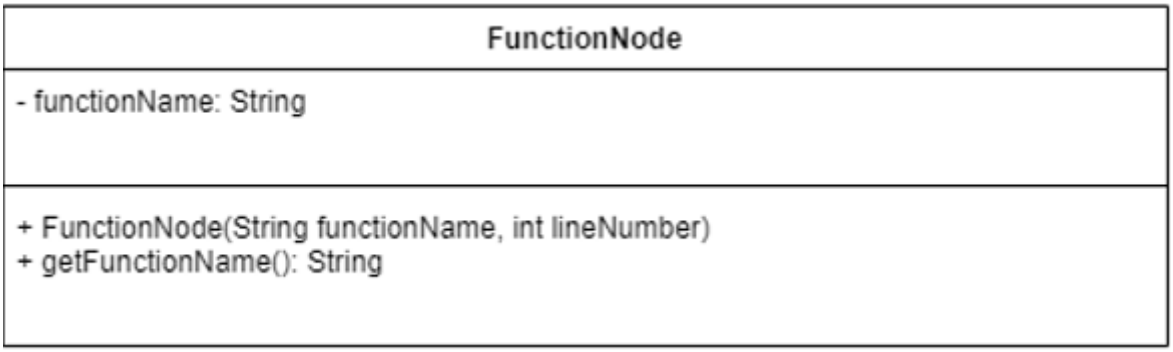


24. ábra: ForStatementNode

A for ciklust eltároló Node: for(int i = 0; i<10; i++) { utasitas1, …}

-> initializer: int i = 0, condition: i<10, iterator: i++ , body: { utasitas1, …}

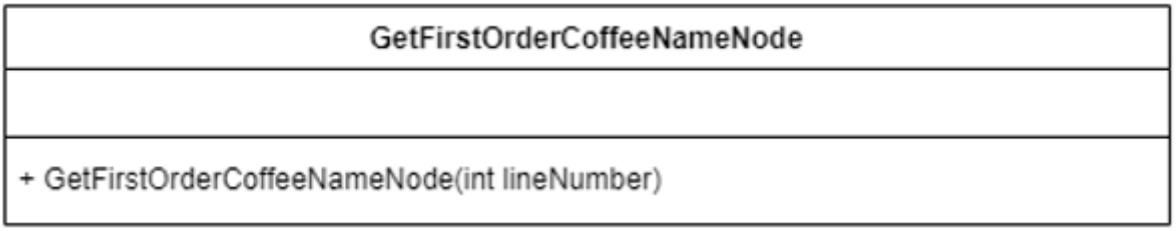
3.6.6.9. FunctionNode



25. ábra: FunctionNode

Programablak hívásokat eltároló csomópont: function program_ablak;

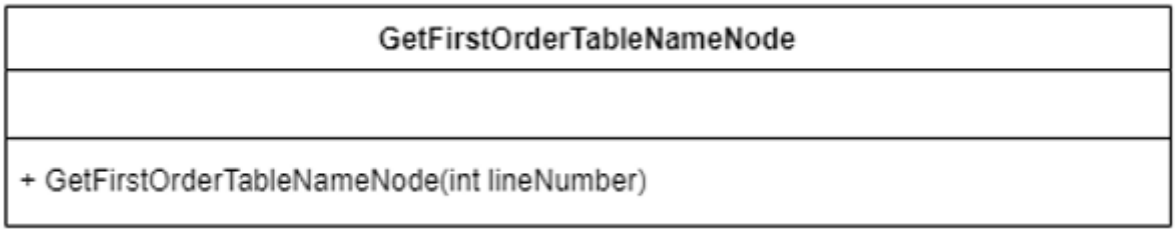
3.6.6.10. GetFirstOrderCoffeeNode



26. ábra: GetFirstOrderCoffeeNameNode

Az első rendelést lekérdező függvényhívást tároló Node.

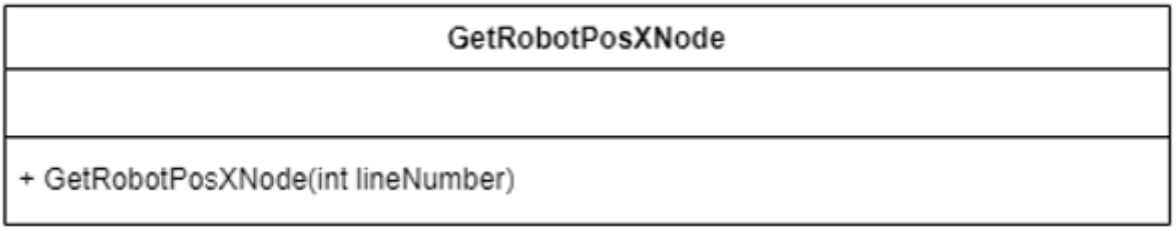
3.6.6.11. GetFirstOrderTableNameNode



27. ábra: GetFirstOrderTableNameNode

Az első rendelést lekérdező függvényhívást tároló Node.

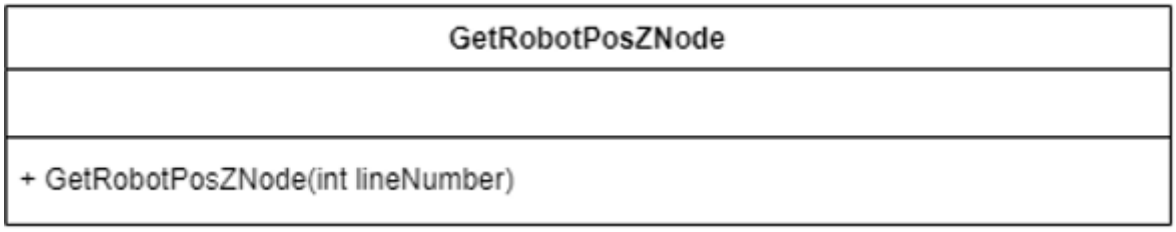
3.6.6.12. GetRobotPosXNode



28. ábra: GetRobotPosXNode

A robot helyzetét lekérdező függvényhívást tároló Node.

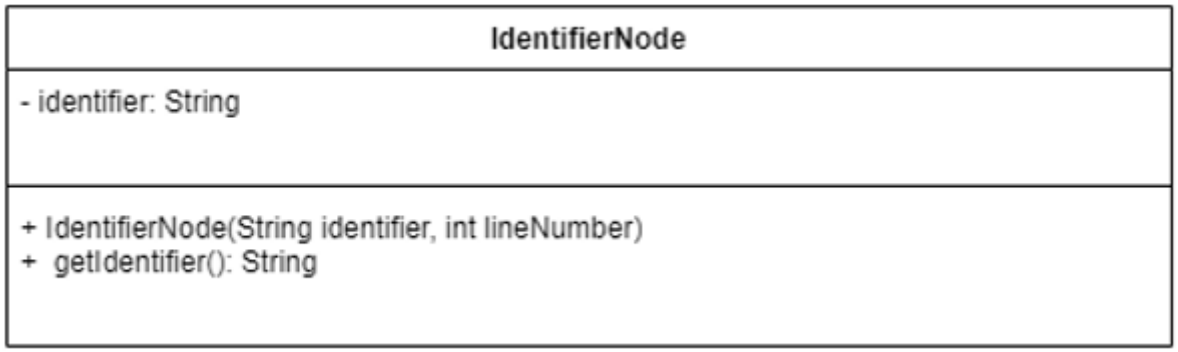
3.6.6.13. GetRobotPosZ



29. ábra: GetRobotPosZNode

A robot helyzetét lekérdező függvényhívást tároló Node.

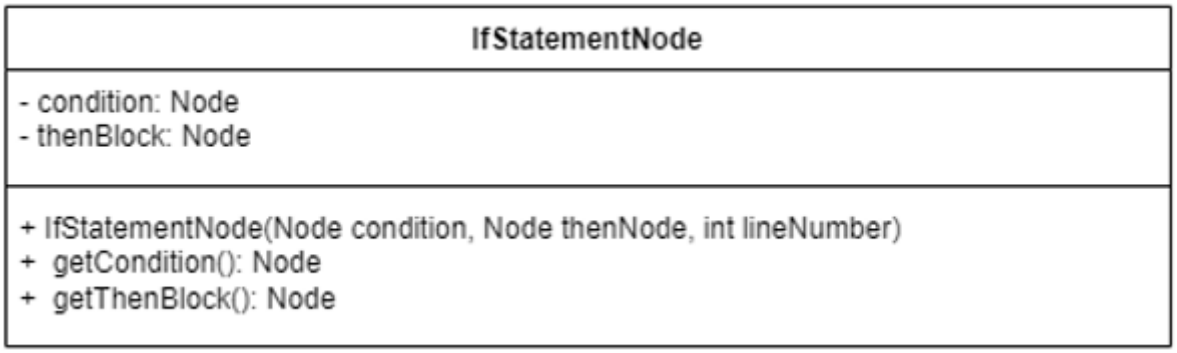
3.6.6.14. IdentifierNode



30. ábra: IdentifierNode

Egy változó nevét tároló Node.

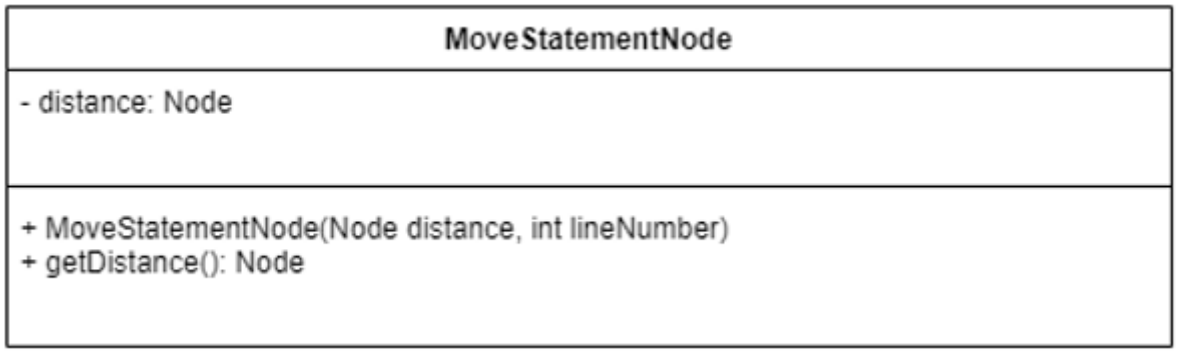
3.6.6.15. IfStatementNode



31. ábra: IfStatementNode

Egy ha elágazést tároló Node.

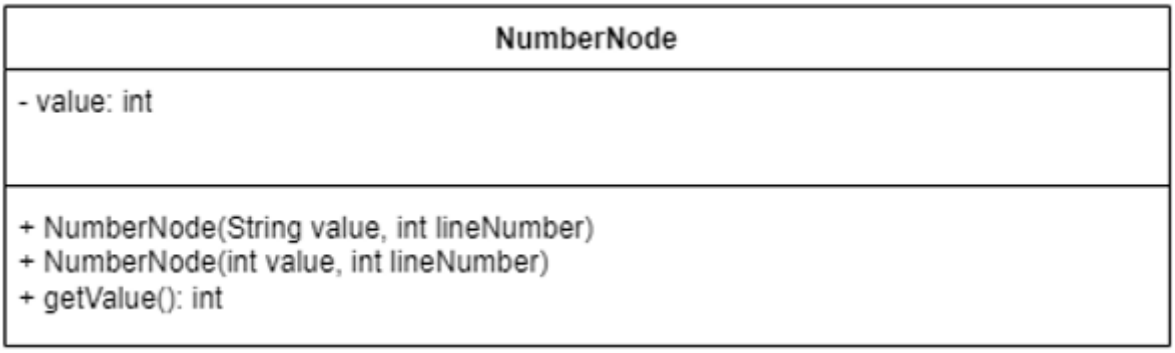
3.6.6.16. MoveStatementNode



32. ábra: MoveStatementNode

A mozgás metódus meghívását tároló Node: move(10);

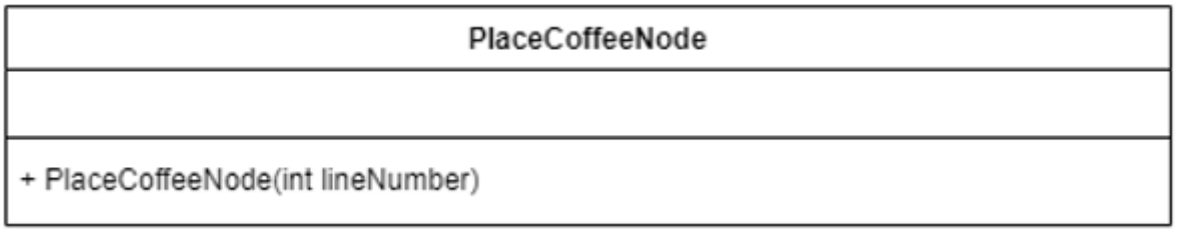
3.6.6.17. NumberNode



33. ábra: NumberNode

Egy szám konstanst tároló Node: 5

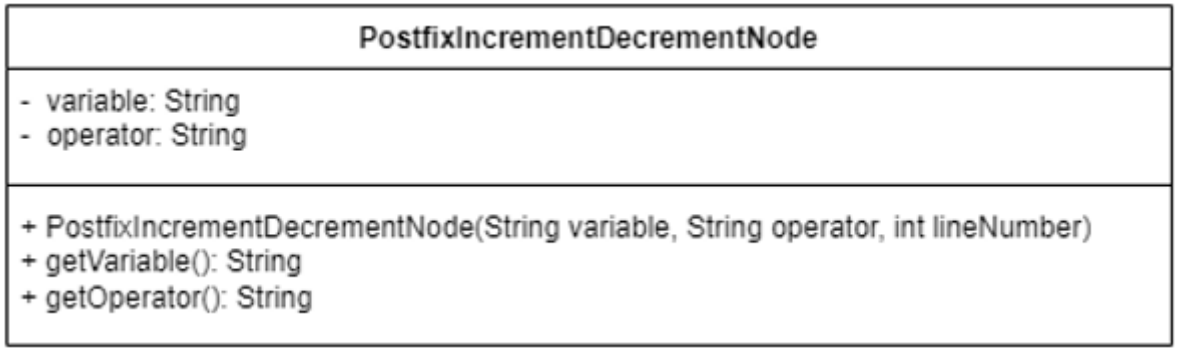
3.6.6.18. PlaceCoffeeNode



34. ábra: PlaceCoffeeNode

A kávé asztalra rakását meghívó metódust tároló Node: placeCoffe();

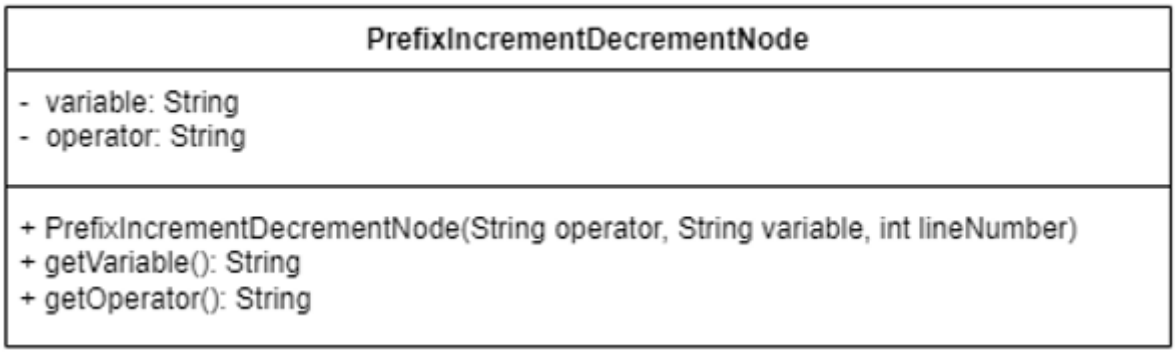
3.6.6.19. PostfixIncrementDecrementNode



35. ábra: PostfixIncrementDecrementNode

Az egyel növelés vagy csökkentés operátorát és operandusát tartalmazza: i++

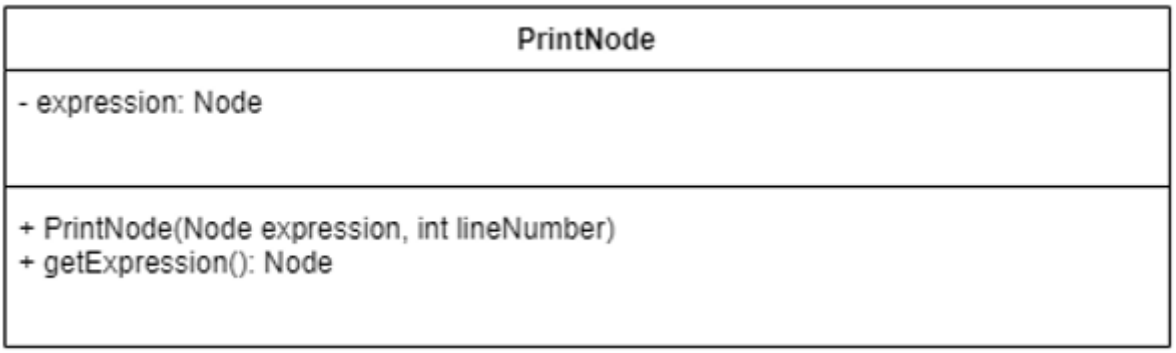
3.6.6.20. PrefixIncrementDecrementNode



36. ábra: PrefixIncrementDecrementNode

Az egyel növelés vagy csökkentés operátorát és operandusát tartalmazza: --i

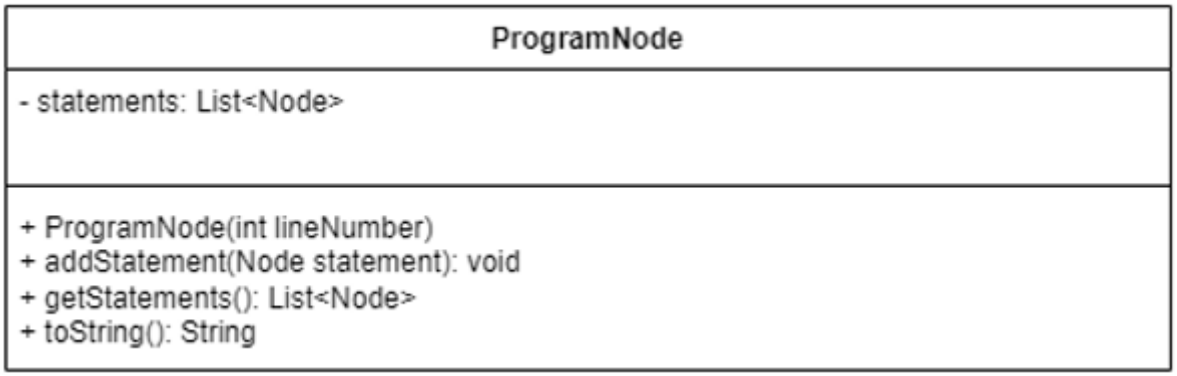
3.6.6.21. PrintNode



37. ábra: PrintNode

A kiíratás függvényt tartalmazó node: print(”Hello World!”);

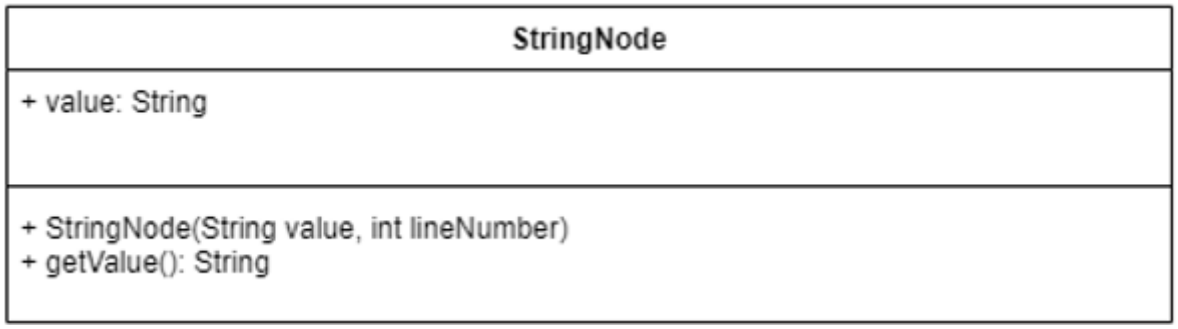
3.6.6.22. ProgramNode



38. ábra: ProgramNode

Az egész kódot tartalmazó Node.

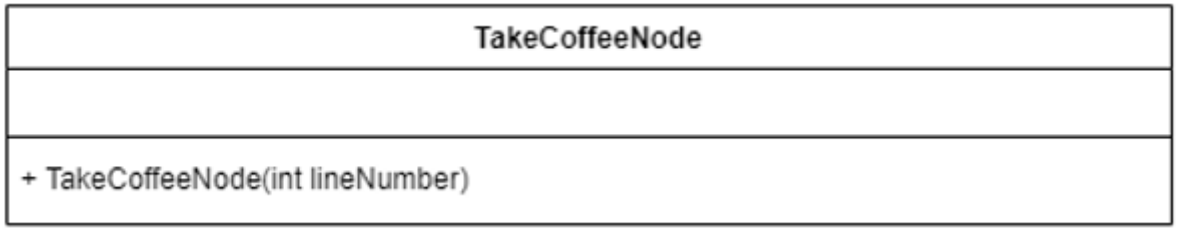
3.6.6.23. StringNode



39. ábra: StringNode

Szöveget tároló Node.

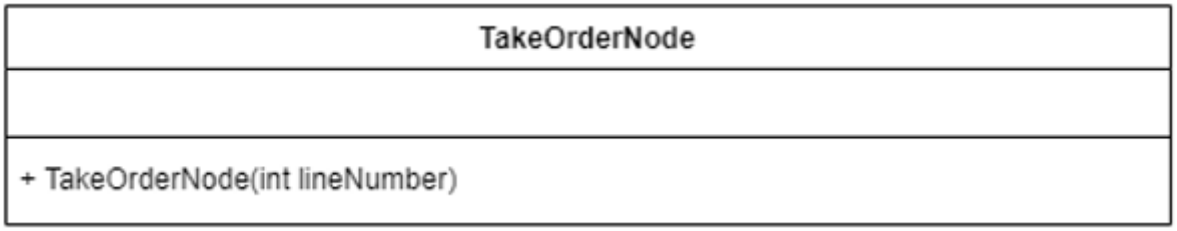
3.6.6.24. TakeCoffeeNode



40. ábra: TakeCoffeeNode

Kávé felvételének metódushívását tartalmazó Node: takeCoffee();

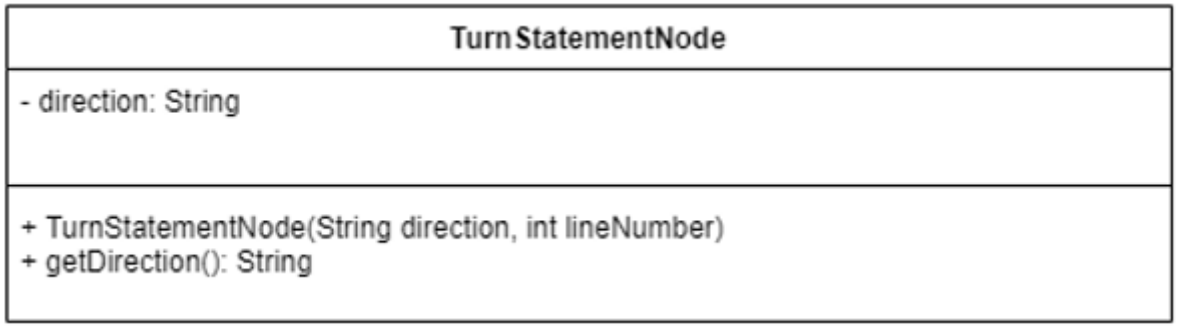
3.6.6.25. TakeOrderNode



41. ábra: TakeOrderNode

Rendelés felvételének metódushívását tároló Node: takeOrder();

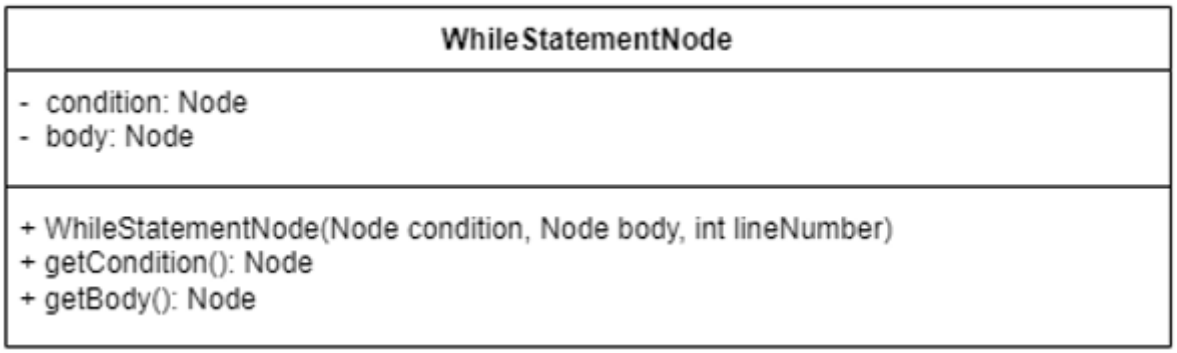
3.6.6.26. TurnStatementNode



42. ábra: TurnStatementNode

A forgatás metódushívást tartalmazó Node: turn(left);

3.6.6.27. WhileStatmentNode

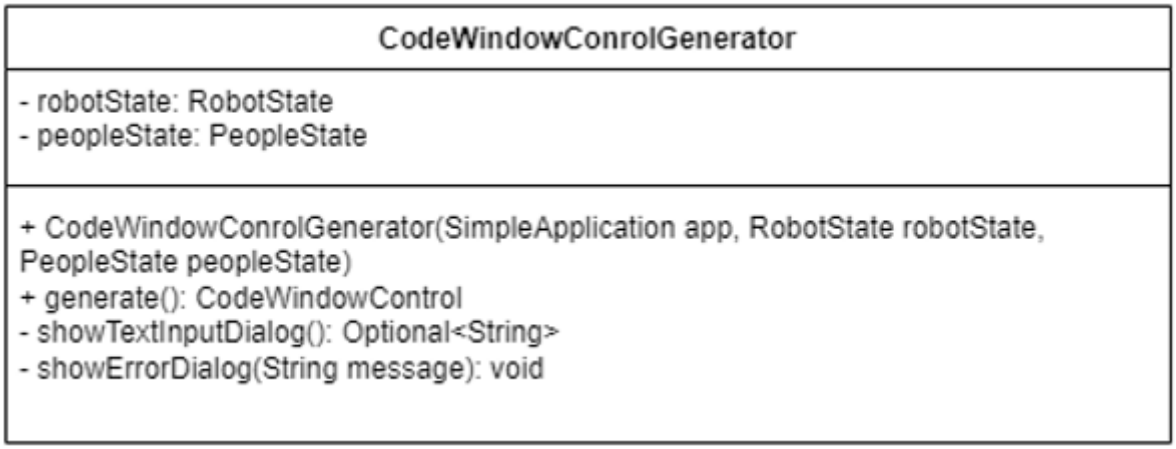


43. ábra: WhileStatementNode

A while ciklust tároló Node: while(i < 10){ utasítás1; … }

-> condition: i < 10 , body: { utasítás1; … }

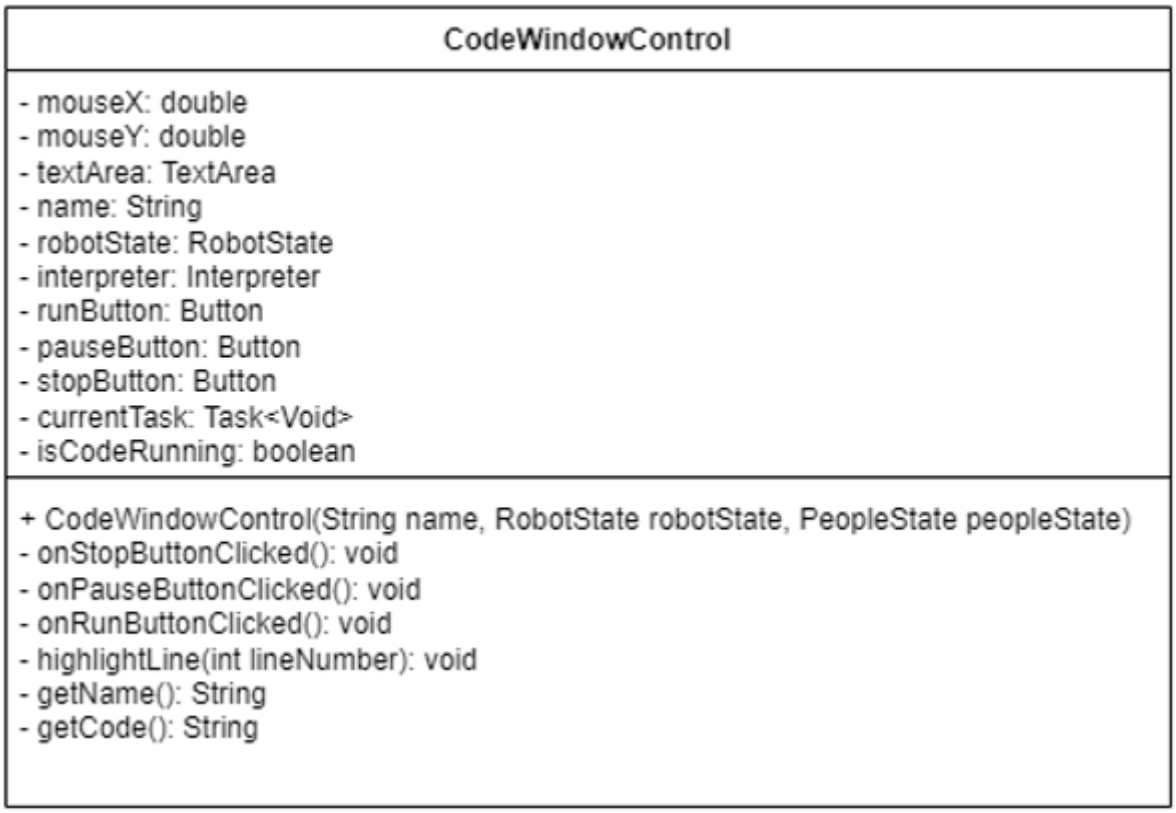
3.6.7. CodeWindowConrolGenerator



44. ábra: CodeWindowConrolGenerator

generate(): A kódablakok létrehozásáért felelős metódus.

3.6.8. CodeWindowControl



45. ábra: CodeWindowControl

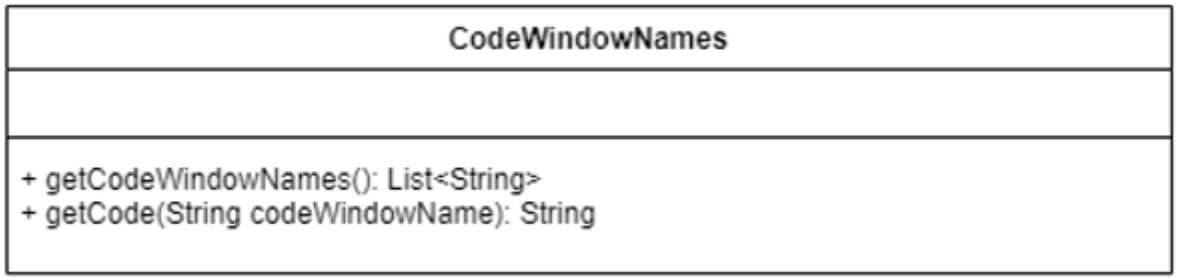
mouseX, mouseY: Az egér pozíciója.

textArea: A Kódot tartalmazó szövegbeviteli mező.

*Button: Gombok a kód futtatására, megállítására, leállítására.

highlightLine(): A futó kódban kijelöli éppen melyik programsor fut.

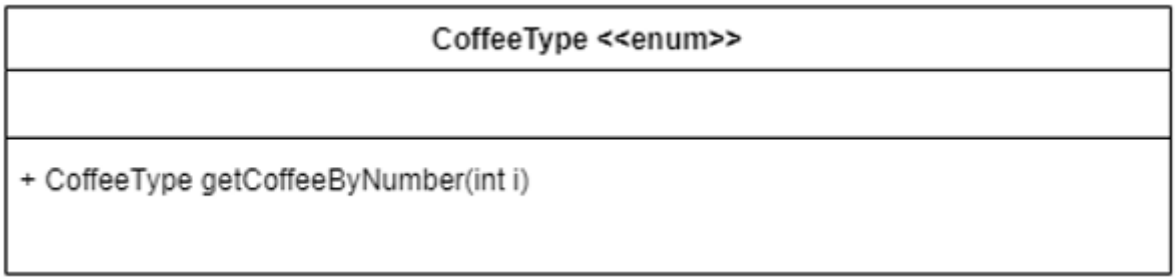
3.6.9. CodeWindowNames



46. ábra: CodeWindowNames

Az osztályon keresztül lekérdezhető a kódablakok neveinek listája, és a kódablak neve alapján a tartalmazott kód.

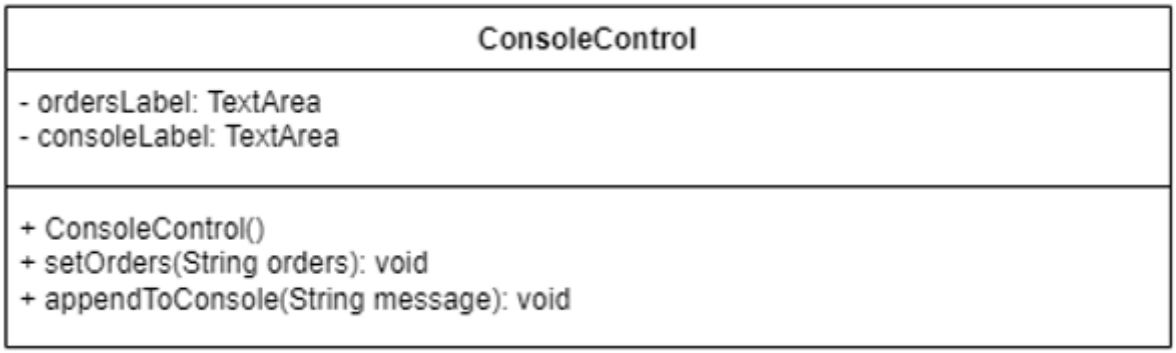
3.6.10. CoffeeType



47. ábra: CoffeeType

Kávétípusokat tartalmazó felsoroló típus.

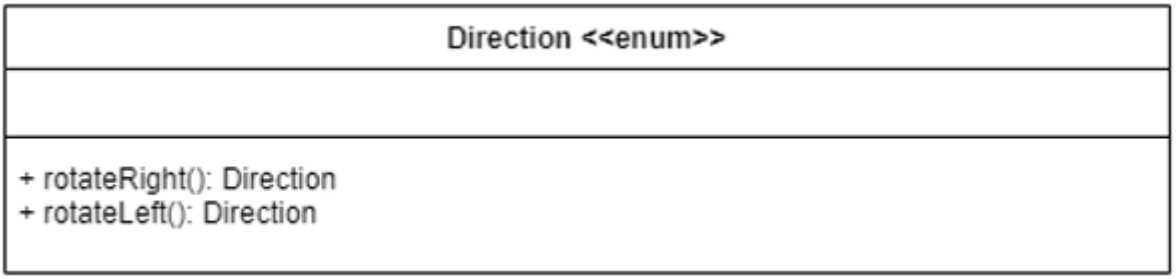
3.6.11. ConsoleControl



48. ábra: ConsoleControl

A rendelések ablakot és konzol ablakot tartalmazó vezérlő.

3.6.12. Direction



49. ábra: Direction

A 4 égtáját megvalósító felsoroló típus.

3.6.13. Interpreter

50. ábra: Interpreter

scopeStack: A változókat és értéküket tároló szintezett hashmap.

functionCache: A prgram által meghívott kódablakok nevei és ProgramNodejuk.

enterScope(): Új szintet nyit a változó stackben.

exitScope(): Visszalép egy szintet a változó stackben.

setVariable(): Új változó létrehozása.

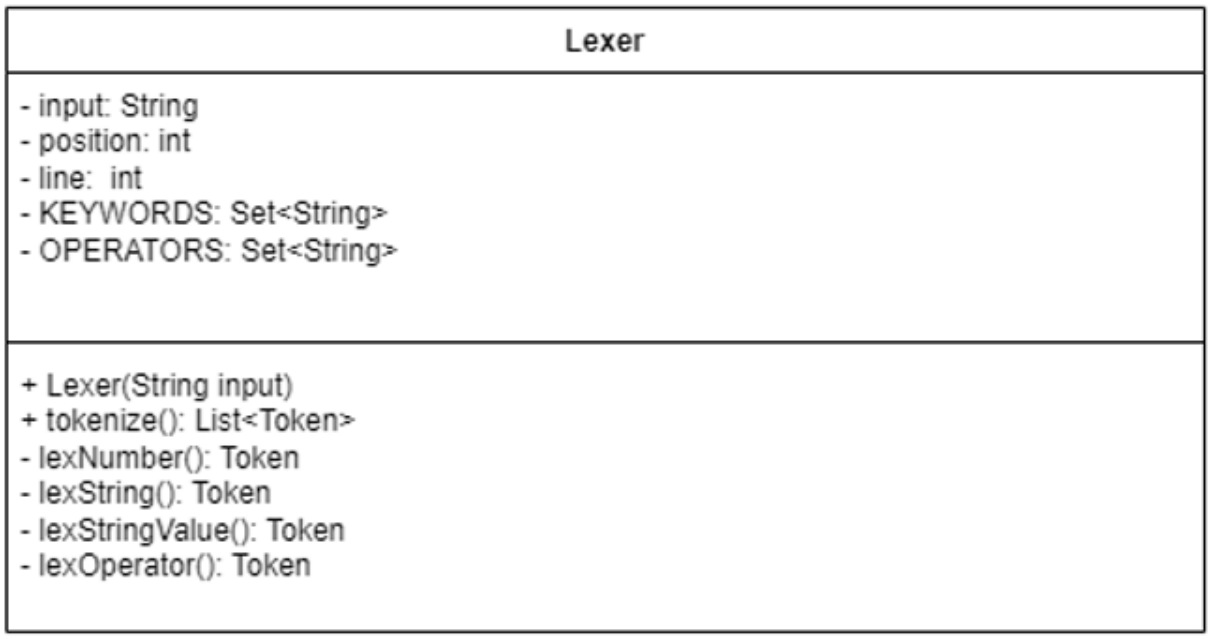
getVariable(): Változó lekérdezése.

changeVariable(): Meglévő váltózó értékének változtatása.

execute(): A mellékhatással rendelkező Node-ok feldolgozása.

evaluate(): A visszatérési értékkel rendelkező Node-ok feldolgozása.

3.6.14. Lexer



51. ábra: Lexer

input: A kódablakban megadott egész szöveg.

position: Karakter pozíció.

line: Sor száma.

KEYWORDS: Kulcsszavak: if, for, while….

OPERATORS: Operátorok: + - * / …

tokenize(): Token listát gyárt.

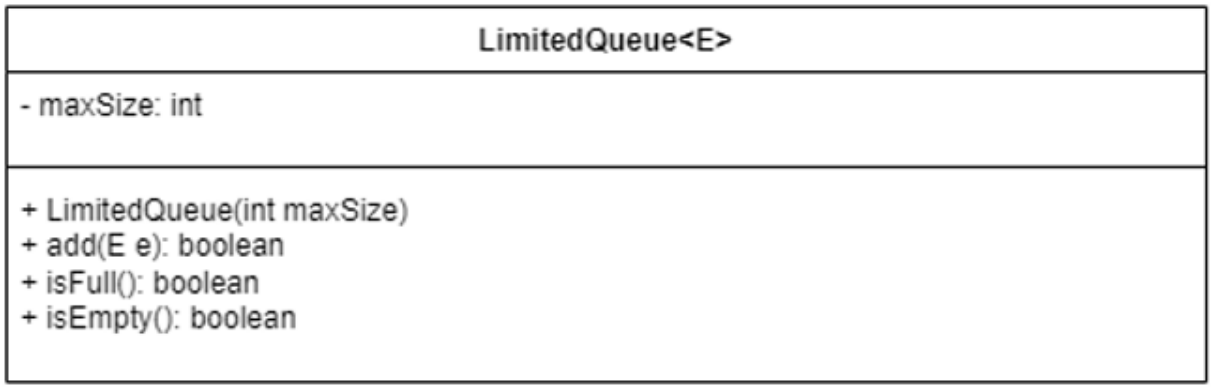
lexNumber(): Szám felismerése és tokenizálása.

lexString(): Kulcsszavak, változónevek felismerése és tokenizálása.

lexStringValue(): Szöveg felismerése, tokenizálása.

lexOperator(): Operátor felismerése, tokenizálása.

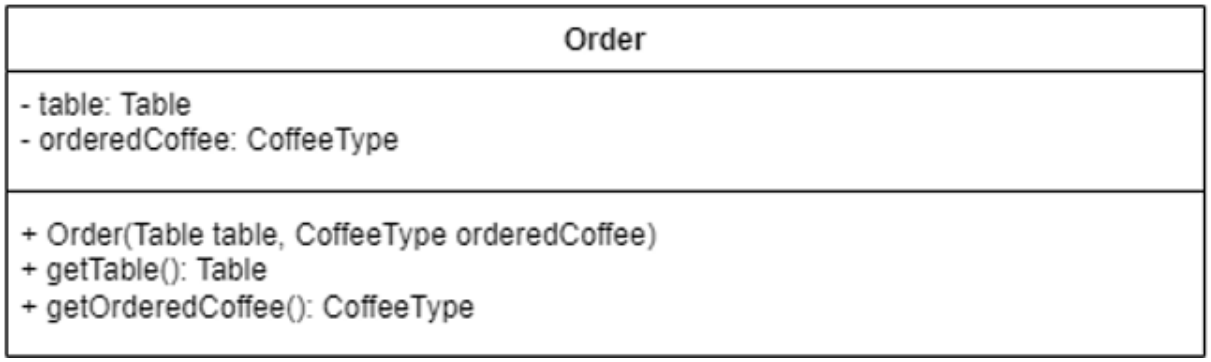
3.6.15. LimitedQueue



52. ábra: LimitedQueue

Queue osztály kiegészítése, hogy figyelje az elemei maximum darabszámát.

3.6.16. Order



53. ábra: Order

Egy rendelés tartalmaz egy kávéfajtát, és hogy melyik asztalhoz tartozik.

3.6.17. Parser

54. ábra: Parser

parse(): A tokenek listáján szintaktikai vizsgálatot végez és felépíti a szintaxis fát.

peek(): A következő tokent adja vissza, de nem mozgat a pozíción.

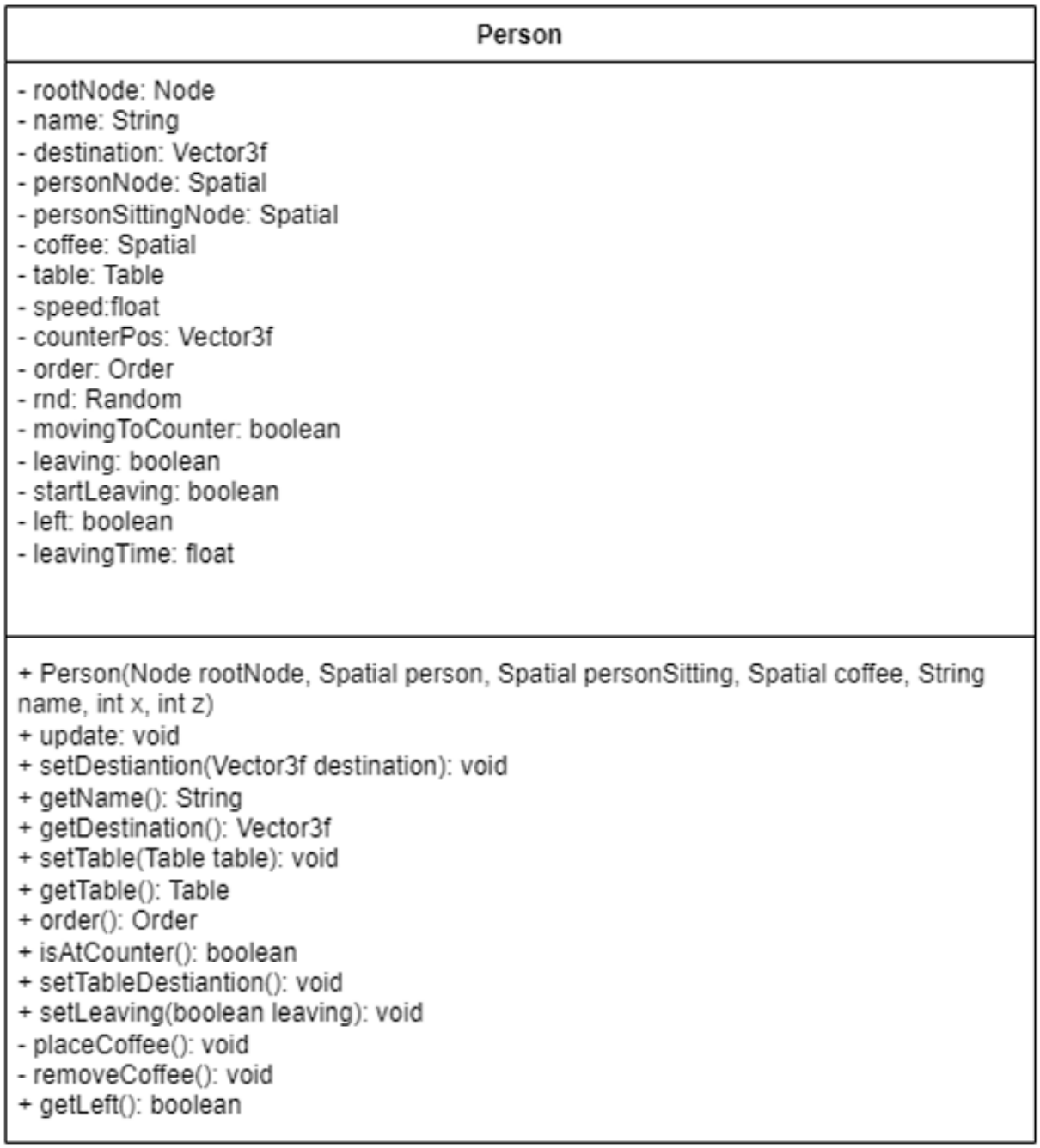
advance(): A jelenlegi tokent adja vissza, majd lépteti a pozíciót.

previous(): Az előző tokent adja vissza.

check(): Ellenőrzi a token típusát.

consume(): Lépteti a pozíciót.

3.6.18. Person



55. ábra: Person

destination: Célpont, ahova tart éppen az ember.

table: Az asztal, ahova leül.

movingToCounter: Igaz ha a pulthoz áll sorban éppen.

leaving: Amikor megkapja a kávéját igazra vált, és elindul egy visszaszámlálás.

startLeaving: Ha letelt a visszaszámlálás, igazra vált és elindul a bejárat fele.

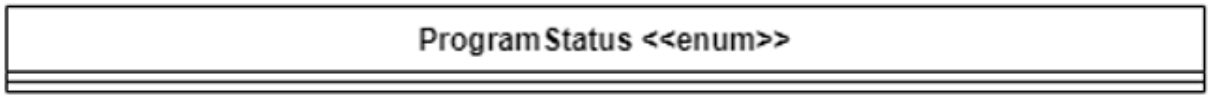
left: Ha megérkezett a bejárathoz.

leavingTime: A visszaszámláló.

update(): A mozgatásokat tartalmazza, framenként hívódik, így a mozgások szakadásmentesek.

order(): Kiválaszt egy véletlenszerű kávét.

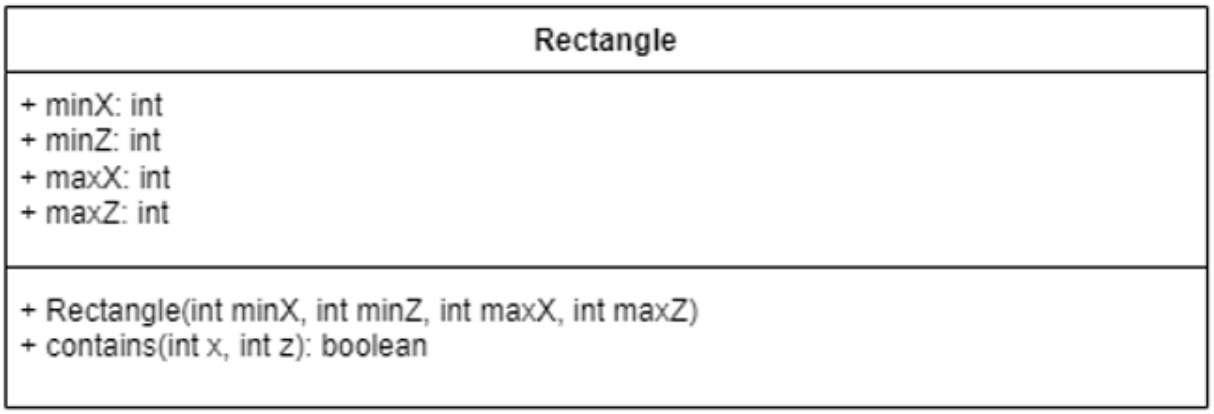
3.6.19. ProgramStatus



56. ábra: ProgramStatus

Felsorolás típus a program állapotára: fut, megállítva, leállítva

3.6.20. Rectangle

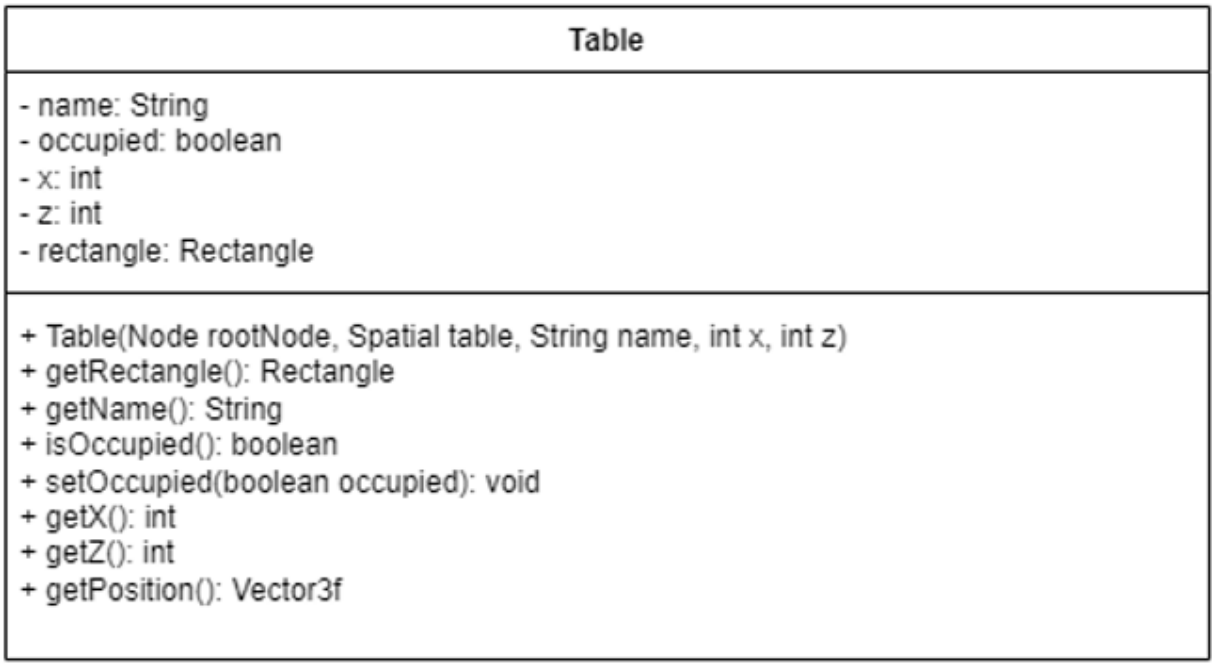


57. ábra: Rectangle

Két ellentétes sarka által megadott téglalap.

contains(): A megadott pont tartalmazza-e a téglalap.

3.6.21. Table



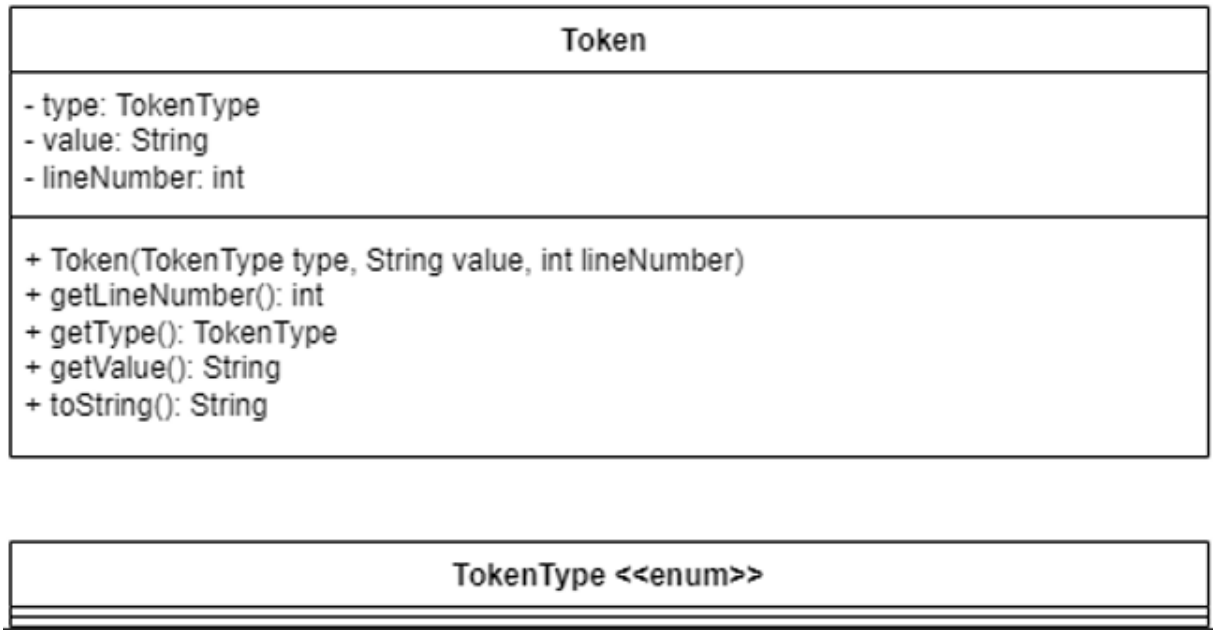
58. ábra: Table

occupied: Foglalt-e egy ember által az asztal.

x,z: Az asztal közepének koordinátáját tárolják.

rectangle: Az egész asztal területe székekkel együtt.

3.6.22. Token, TokenType



59. ábra: Token, TokenType

type: A token típusa.

value: A token tartalma.

lineNumber: Melyik sorban foglal helyet a token.

TokenType: Tokenek típusainak felsorolási típusa: OPERATOR, NUMBER, STRING, KEYWORD…

3.7. Tesztelés

3.7.1. Manuális tesztelés

Mivel a program 3D-s megjelenítést és interaktív felületet használ, ezek tesztelése manuálisan történt. A manuális tesztelés során ellenőriztem a felhasználói interakciókat, a vizuális elemek helyes működését, valamint a program által generált eredményeket.

3.7.2. UnitTest

A program azon részei, amelyek tisztán adatfeldolgozással foglalkoznak, mint például a fordításhoz használt Lexer, Parser és Interpreter osztályok, kiválóan alkalmasak egységtesztelésre. A tesztelési folyamatot rétegekre bontottam, hogy minden komponens külön-külön és integráltan is ellenőrizhető legyen. A teszteléshez Junit keretrendszert használtam.

3.7.2.1. Lexer tesztelése

A Lexer osztály felelős a bemeneti kód tokenizálásáért. A tesztek során kisebb kódrészleteket használtam annak ellenőrzésére, hogy a Lexer helyesen generálja-e a tokeneket. Például a print(10 + 20);

bemenet esetén ellenőriztem, hogy a megfelelő tokenek jönnek-e létre, mint a KEYWORD, NUMBER, OPERATOR stb. Hibás szintaxisú bemenetekkel is teszteltem, hogy a Lexer megfelelően dob-e kivételt.

3.7.2.2. Parser tesztelése

A Parser osztály a Lexer kimenetét használja bemenetként, és abból absztrakt szintaxisfát (AST) épít. A tesztek során ellenőriztem, hogy a Parser helyesen építi-e fel az AST-t. Mivel a Parser bemenete a Lexer kimenete, a Parser tesztelése közben a Lexer működése is ellenőrzésre kerül.

3.7.2.3. Interpreter tesztelése

Az Interpreter osztály tesztelése már hosszabb, komplexebb programkódokkal történt, amelyek integrációs tesztekhez hasonló módon lefedik a Lexer és a Parser működését is. Az Interpreter működését a mellékhatások, például a konzolra írt értékek figyelésével ellenőriztük. Ez lehetővé tette, hogy nagyobb kódbázist teszteljek anélkül, hogy minden egyes lépéshez külön assert hívásokat kellett volna írni. Például egy

```
int a = 10;

print(a + 20);
```

bemenet esetén ellenőriztük, hogy a konzolra a 30 érték kerül kiírásra.

3.7.2.4. Hibakezelés tesztelése

A tesztek során figyelmet fordítottam a hibás bemenetek kezelésére is.

4. További fejlesztési lehetőségek

4.1. Fejlesztési lehetőségek

A program továbbfejlesztésével számos új funkcióval lehetne gazdagítani a játékélményt, amelyek még szórakoztatóbbá és kihívásokkal telibbé tennék a játékot:

4.1.1. Monetizáció bevezetése

- A vásárlók fizethetnének a kávéért, és a megszerzett pénzből a játékos új asztalokat vásárolhatna, bővíthetné a kávéházat, vagy fejleszthetné a robot képességeit.
- A pénz bevezetése lehetőséget adna gazdasági mechanikák implementálására, például költségvetés kezelésére vagy profit maximalizálására.

4.1.2. Megvásárolható kódok

- A játékos kezdetben csak alapvető kódszavakhoz férhetne hozzá, például move vagy turn.
- Ahogy halad előre a játékban, új kódszavakat vásárolhatna meg a megszerzett pénzből, például takeOrder vagy placeCoffee.
- Ez a mechanika ösztönözné a játékost, hogy hatékonyabb kódokat írjon, és jobban gazdálkodjon az erőforrásaival.

4.1.3. Többfázisú kávékészítés

- A kávékészítés folyamata több lépésből állhatna, például: őrlés, főzés, tejhabosítás, díszítés.
- A játék előrehaladtával a rendeléseket egyre bonyolultabbá lehetne tenni, például speciális kávék vagy egyedi igények megjelenítésével.

4.1.4. Dizájntárgyak és testreszabás

- A játéktérhez különböző dizájntárgyakat lehetne hozzáadni, például növényeket, festményeket vagy bútorokat, amelyekkel a játékos testreszabhatná a kávéházat.
- Ezek a tárgyak nemcsak esztétikai értéket adnának, hanem akár bónuszokat is nyújthatnának, például gyorsabb kiszolgálást vagy több vásárlót.

4.1.5. Véletlenszerű események

- A játék során véletlenszerű események akadályozhatnák a robotot, például kiömlik a kávé, elromlik egy gép, vagy egy vásárló reklamál.
- Ezek az események növelnék a játék dinamikáját, és arra ösztönöznék a játékost, hogy gyorsan és hatékonyan reagáljon a váratlan helyzetekre.

Ezek a fejlesztési lehetőségek nemcsak a játékélményt tennék gazdagabbá, hanem hosszabb távon is fenntartanák a játékos érdeklődését, miközben új kihívásokat és célokat kínálnának.

5. Irodalomjegyzék

[1] https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html (2025.05.01)