3. Beadandó feladat dokumentáció

Készítette: Zágoni Márton

E-mail: cl8524@inf.elte.hu

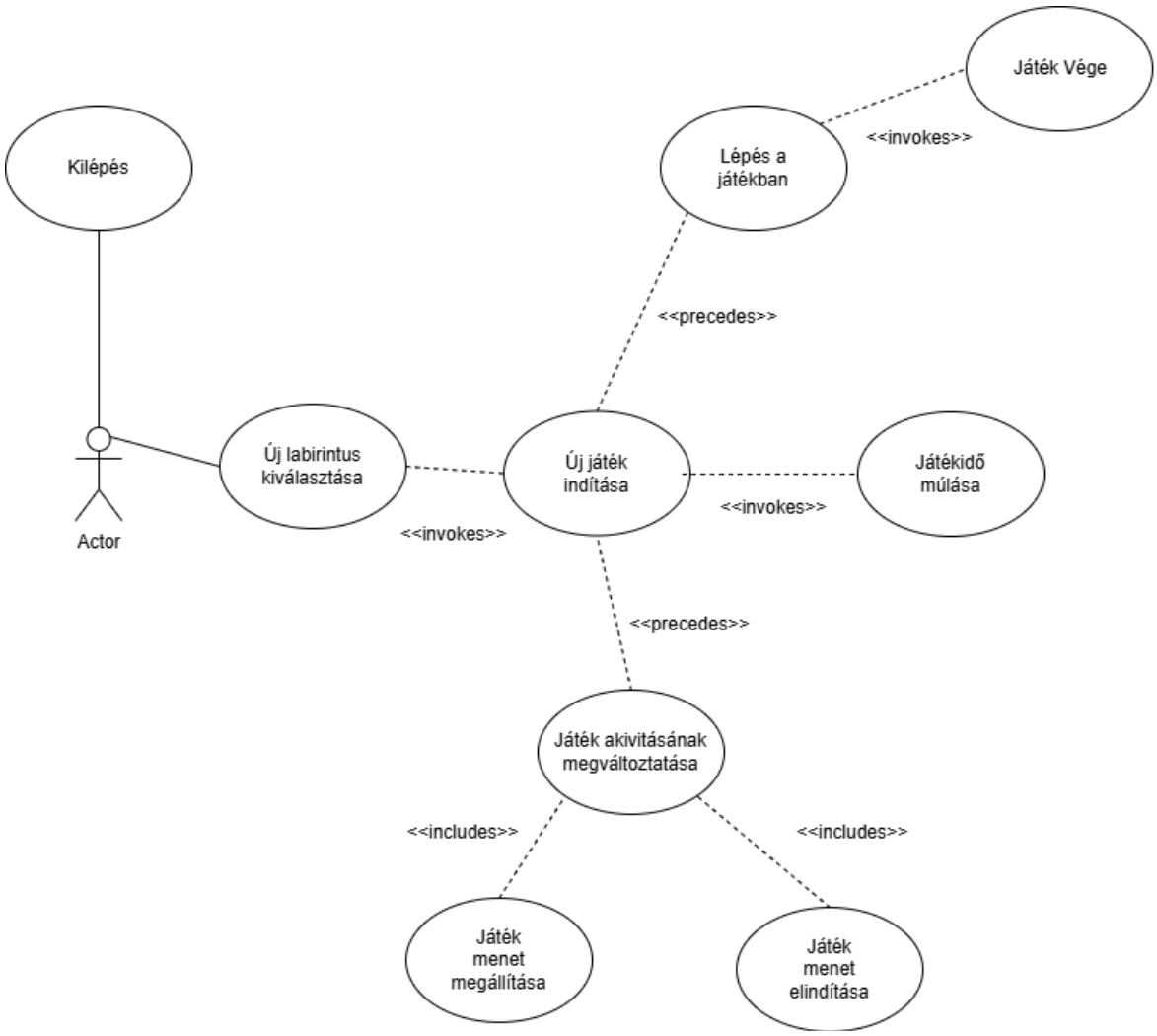
Feladat:

Készítsünk programot, amellyel a következő játékot játszhatjuk. Adott egy $n \times n$ elemből álló játékpálya, amely labirintusként épül fel, azaz fal, illetve padló mezők találhatóak benne, illetve egy kijárat a jobb felső sarokban. A játékos célja, hogy a bal alsó sarokból indulva minél előbb kijusson a labirintusból. A labirintusban nincs világítás, csak egy fáklyát visz a játékos, amely a 2 szomszédos mezőt világítja meg (azaz egy 5 × 5-ös négyzetet), de a falakon nem tud átvilágítani. A játékos figurája kezdetben a bal alsó sarokban helyezkedik el, és vízszintesen, illetve függőlegesen mozoghat (egyesével) a pályán. A pályák méretét, illetve felépítését (falak, padlók) tároljuk fájlban. A program legalább 3 különböző méretű pályát tartalmazzon. A program biztosítson lehetőséget új játék kezdésére a pálya kiválasztásával, valamint játék szüneteltetésére (ekkor nem telik az idő, és nem léphet a játékos), továbbá ismerje fel, ha vége a játéknak. A program játék közben folyamatosan jelezze ki a játékidőt.

Elemzés:

- A programnak legalább 3 labirintussal kell rendelkeznie, amiknek különböző méretűnek kell lenniük.
- Új játék indításához a felhasználó kiválaszt egy pályát, a labirintusban mindig bal alul kezd.
- A játék legyen megállítható és ekkor a stopper is álljon meg, és a játékos se tudjon lépni.
- A játékidő legyen jelezve a képernyőn.
- A játék végét felismeri a program és üzenetet küld a felhasználónak.
- A feladatot egyablakos asztali alkalmazásként Windows Forms grafikus felülettel valósítjuk meg.
- Az ablakban elhelyezünk egy menüsort, amiben lesz egy labirintus és egy játék menüpont. A labirintus menüpontban lesznek elérhetőek a különböző labirintus pályák, a játék menüpontban pedig a játék megállítása és folytatása lesz elérhető. Az ablak alján elhelyezkedő státuszsor pedig az eltelt időt jeleníti meg.
- Magát a labirintust egy NxN es mátrix reprezentálja, amit egy game panelen jelenítünk meg festett négyzetek segítségével. Magát a játékost a billentyűzet nyilaival lehet irányítani a labirintusban. A labirintusban falon nem tud átmenni a játékos vagy lemenni a pályáról se tud. A játékos lámpa fénye egy 5x5 térben világítja be a játékos körüli négyzeteket ügyelve arra, hogy a falakon ne világítson át. A kezdő négyzet piros míg a cél négyzet zöld színű.

- A játék automatikusan feldob egy dialógusablakot, amikor vége a játéknak, ezen megjeleníti a kijutási időt. Az ablak eltűnése után ugyanúgy indíthatunk új játékot.
- A felhasználói esetek az 1. ábrán láthatóak.



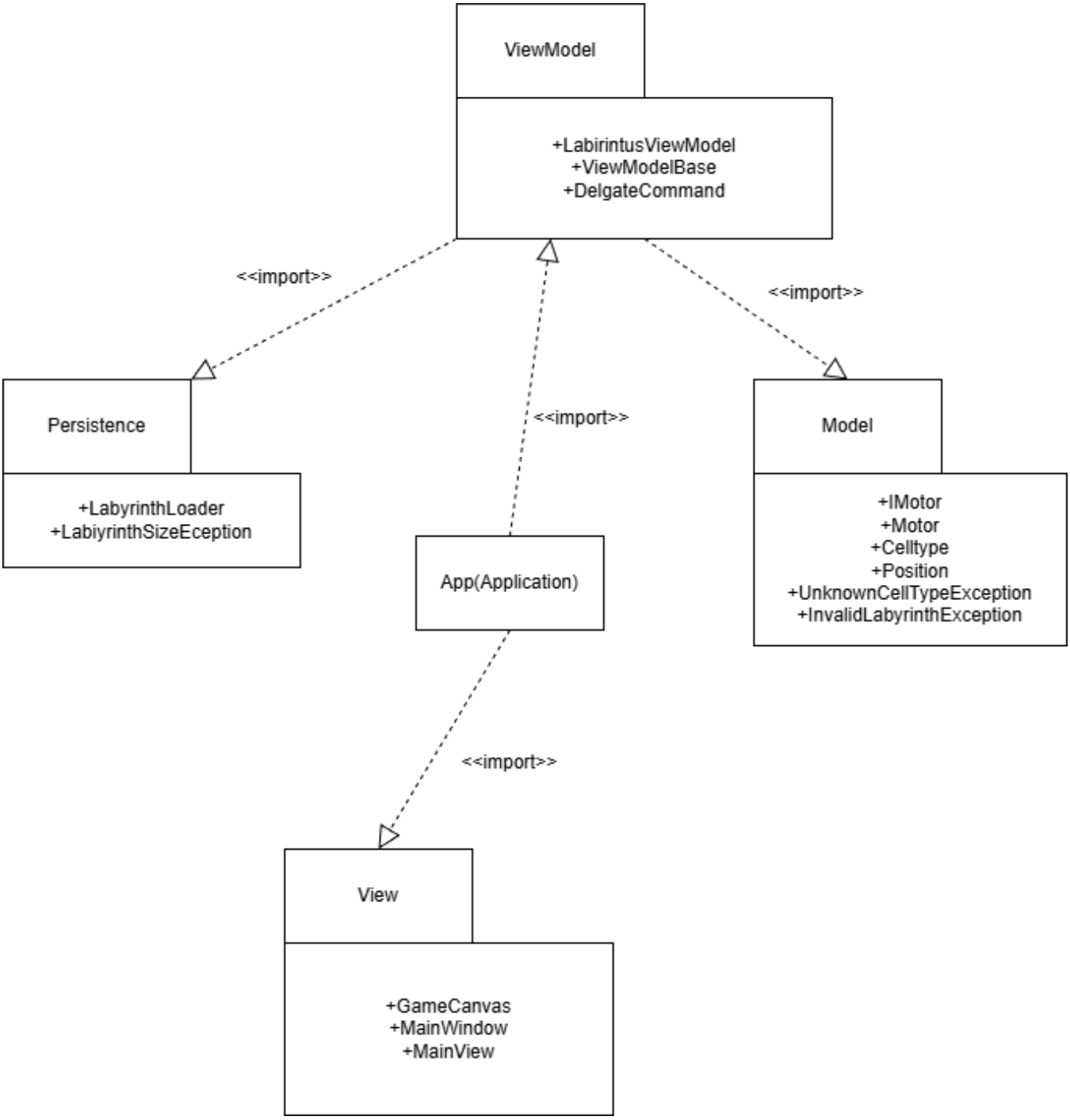
1. ábra: Felhasználói esetek diagramja

Tervezés

- **Programszerkezet**:

A szoftvert .NET Avalonia alkalmazásként valósítjuk meg, amely MVVM (Model-View-ViewModel) architektúrára épül. Ennek megfelelően a kódot logikailag négy rétegre bontjuk: a modellt (Model), a nézetmodellt (ViewModel), a nézetet (View) és a perzisztenciát (Persistence) tartalmazó névterekre.A program vezérlését az alkalmazás osztály (App) végzi, amely példányosítja a modellt, a nézetmodellt és a nézetet, valamint biztosítja a rétegek

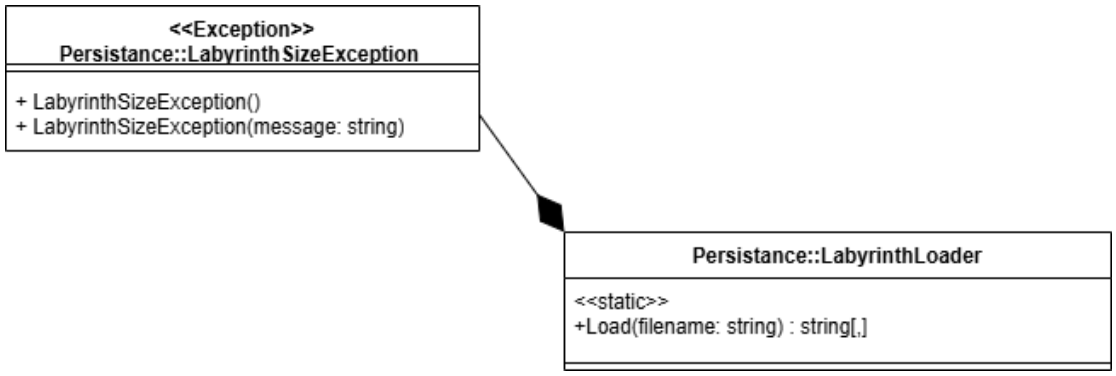
közötti kommunikációt eseményfeliratkozások segítségével.



2. ábra: Az alkalmazás csomagdiagramja

- **Perzisztencia**:

- Az adatkezelés feladata a labirintus pályáinak betöltése külső fájlból, amely tartalmazza a pálya szerkezetét (falak, utak, kezdőpont, kijárat). A réteg célja, hogy a játék logikai modellje számára biztosítsa a pályát reprezentáló adatokat.
- A LabyrinthLoader osztály felelős a pályaadatok beolvasásáért. A Load(string filename) metódus megnyitja a megadott szöveges fájlt, beolvassa annak tartalmát soronként, majd azt kétdimenziós sztringtömbbé (*string[,]*) alakítja, amely a pálya celláit írja le.
- A fájl formátuma egyszerű, minden sor egy pályasornak felel meg, és minden karakter a pálya egy-egy mezőjét jelöli. A betöltő biztosítja, hogy a fájl ne legyen üres amennyiben a fájl üres, a metódus egy üres (*0×0*) tömböt ad vissza, így a program nem fut hibára.
- A pálya méretét a fájl sorainak száma, illetve az első sor hossza határozza meg. A LabyrinthLoader ellenőrzi, hogy minden sor azonos hosszúságú legyen. Ha a feltétel nem teljesül, LabyrinthSizeException kivételt dob, ezzel garantálva, hogy a pálya mindig négyzetes vagy téglalap alakú mátrixként kezelhető.
- A pályát alkotó karakterek jelentése a következő (CellType típusok):
  - # → fal (*Wall*)
  - . → út (*Road*)
  - S → kezdőpont (*Start*, járható mezőként kezelve)
  - E → kijárat (*Exit*)
- A fájlban található ismeretlen vagy hibás karakterek kezeléséért a játék modellje (nem lehet tudni előre hogy ha másik modell lesz miket használ fel) felel, ugyanakkor a betöltő biztosítja, hogy minden sor helyesen feldolgozható legyen.
- A pályaadatok forrása hagyományos **.**txt kiterjesztésű szövegfájl, amely bármikor betölthető a játék indításakor. A fájl sorainak száma a pálya magasságát, a sorok karaktereinek száma pedig a szélességét adja meg.
- A LabyrinthSizeException osztály egyedi kivételtípust biztosít az olyan hibák kezelésére, amikor a fájl formátuma nem felel meg az elvárásoknak (például különböző hosszúságú sorok esetén).

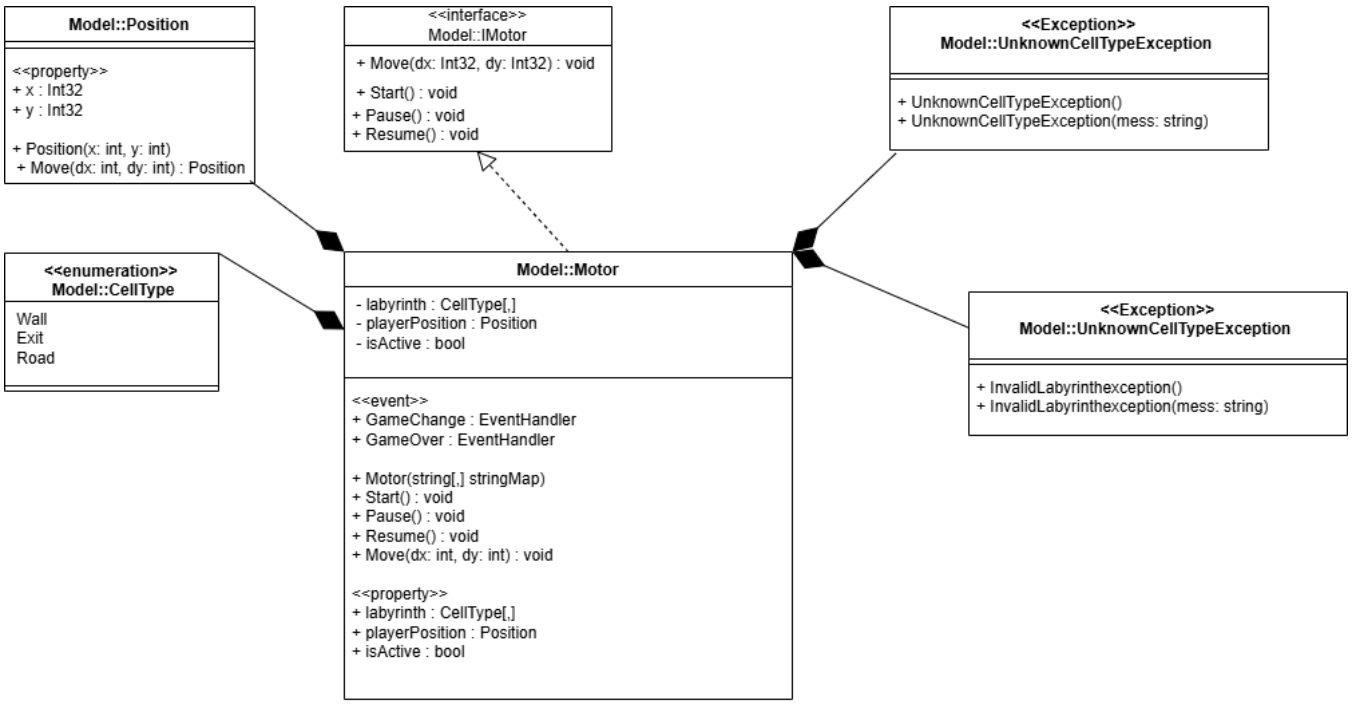


- **Modell**:
  - A játék működését a Motor osztály valósítja meg, amely irányítja a játékos mozgását a labirintusban, kezeli a játék állapotát és az eseményeket. Az

osztály tárolja a játéktér leírását (*labyrinth*), a játékos aktuális pozícióját (*playerPosition*), valamint a játék futásának állapotát (*isActive*).

- A játék elindítása a Start() metódussal történik, amely aktiválja a motort és jelzi a megjelenítőnek az állapotváltozást a GameChange esemény segítségével.
- A játék szüneteltethető (Pause()) és újraindítható (Resume()) a megfelelő metódusok meghívásával, amelyek az *isActive* állapotot módosítják.
- A játékos mozgását a Move(dx, dy) metódus kezeli:
  • ellenőrzi, hogy a játék aktív-e,
  • kiszámítja az új pozíciót a Position.Move() metódus segítségével,
  • meggátolja, hogy a játékos a pályán kívülre vagy falra lépjen,
  • érvényes lépés esetén frissíti a játékos pozícióját és GameChange eseményt vált ki,
  • amennyiben a játékos kijárathoz érkezik (*CellType.Exit*), a GameOver esemény jelzi a játék végét.
- A labirintus szerkezetét egy karaktertömbből (*stringMap*) hozza létre a Motor konstruktora, amely a következő szimbólumokat értelmezi:
  • „#” → *CellType.Wall* (fal),
  • „.” → *CellType.Road* (járható út),
  • „S” → *CellType.Road* és egyben a játékos kezdőpozíciója,
  • „E” → *CellType.Exit* (kijárat).
- A Position osztály a játékos koordinátáit tárolja (*x*, *y*), és lehetőséget ad az új pozíció meghatározására az Move(dx, dy) metóduson keresztül, amely új *Position* objektumot ad vissza az elmozdulásnak megfelelően.
- A CellType felsorolás a pálya mezőinek típusait írja le (*Wall*, *Road*, *Exit*).
- Minden egyes alkalommal, ha a játkos lépett az UpdateVisibleCells() metódus meg lesz hívva és legenerálja a játékos által látható mezőket, ezt a View rétegben fogja majd kirajzolni a program, de a számítás a modellben történik meg.
- A Motor osztály eseményeket (*GameChange*, *GameOver*) biztosít a felhasználói felület vagy más komponensek számára, így azok valós időben reagálhatnak a játék állapotváltozásaira.

- A korábban definiált IMotor interfész meghatározza a motor alapvető műveleteit (lépés, indítás, szüneteltetés, folytatás, konvertálás), ezzel lehetővé téve a motor egységes kezelését és cserélhetőséget.



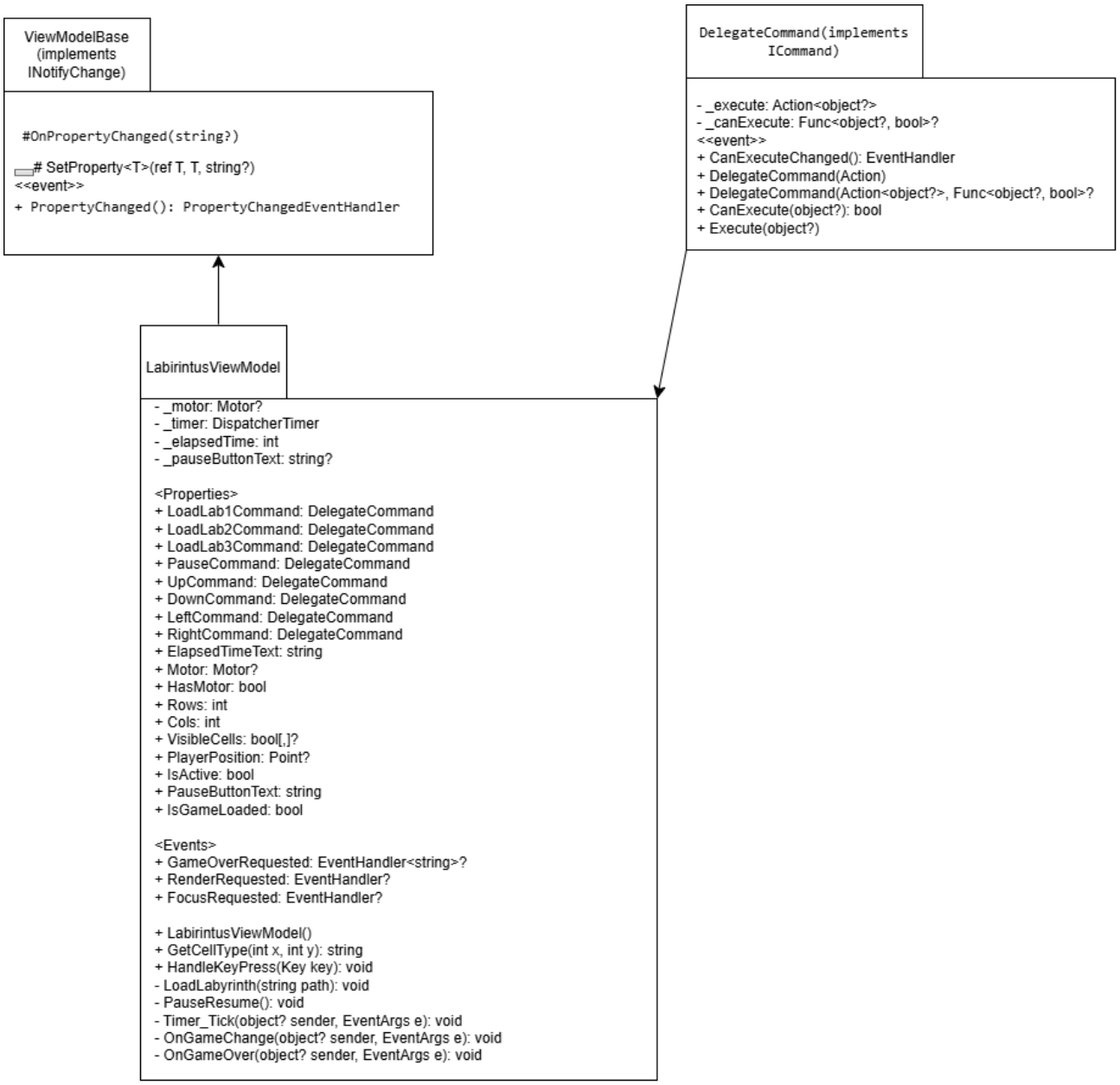
ViewModel

A nézetmodell réteg feladata a modell és a nézet közötti kapcsolat biztosítása, valamint az állapotok és parancsok kezelése MVVM (Model-View-ViewModel) mintának megfelelően. A megvalósítás alapja a ViewModelBase osztály, amely az INotifyPropertyChanged interfész implementálásával és a [CallerMemberName] attribútum használatával gondoskodik az adatkötések hatékony frissítéséről. A felhasználói interakciókat a DelegateCommand osztály általános parancsai kezelik, amelyek lehetővé teszik a műveletek logikájának és engedélyezési feltételeinek elválasztását a felülettől.

A központi logikát a LabirintusViewModel osztály fogja össze. Ez tárolja a modell (Motor) referenciáját, és „átmenő” tulajdonságokon keresztül (Rows, Cols, PlayerPosition, VisibleCells) teszi elérhetővé a játékállapotot a nézet számára anélkül, hogy azokat duplikálná. A játékidő méréséért egy DispatcherTimer felel, amely másodpercenként frissíti a felhasználó felé megjelenített időt (ElapsedTimeText). A nézetmodell eseményfeliratkozásokon keresztül figyeli a modell állapotváltozásait (OnGameChange), és ezeket a nézet felé saját eseményekkel (RenderRequested, GameOverRequested) továbbítja, így biztosítva a grafikus felület újrarajzolását vagy a játék végét jelző üzenet megjelenítését.

View

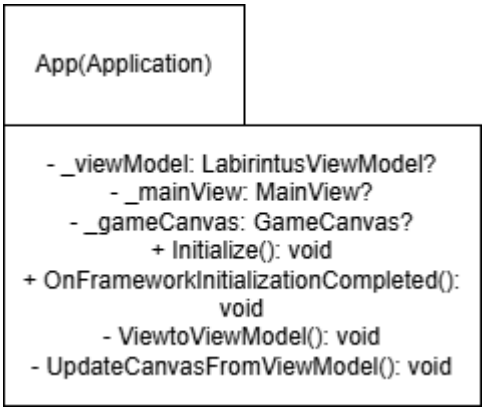
A felhasználói felületet a MainView (UserControl) és a MainWindow ablak alkotja, amelyek elrendezését egy rács (Grid) strukturálja: felül a pályaválasztó menü, alul a vezérlőgombok és a státuszsor, középen pedig a játéktér helyezkedik el. A nézet és a nézetmodell összekötése nem a XAML-ben, hanem az alkalmazás indulásakor (App.xaml.cs) történik, biztosítva a tiszta függőségkezelést.
A játéktér megjelenítését egy speciális, teljesítményoptimalizált vezérlő, a GameCanvas végzi. A hagyományos gomb-alapú rácsok helyett ez az osztály a Control ősosztályból származik, és a Render metódus felüldefiniálásával közvetlenül a rajzolási kontextusra (DrawingContext) rajzolja ki a labirintus falait, a padlót és a játékost. Ez a megoldás lehetővé teszi a gyors és villódzásmentes megjelenítést. A vezérlés érdekében a GameCanvas elfogja a billentyűzeteseményeket (OnKeyDown), amelyeket az App osztály segítségével továbbít a nézetmodell HandleKeyPress metódusának feldolgozásra.



App

- Az alkalmazás indításáért és a futtató környezettel való kommunikációért az App osztály (App.xaml.cs) felel. A program inicializálása során a ViewModel-First (Nézetmodell-alapú) megközelítést alkalmazzuk, amely biztosítja a logika és a megjelenítés tiszta szétválasztását már a belépési pontnál.
- Az alkalmazás indításakor a felület automatikus betöltése helyett – amelyet hagyományosan az App.xaml-ben definiált StartupUri tulajdonság végezne – a vezérlést az App.xaml.cs-ben felülírt indítási eseménykezelő (Avalonia esetén OnFrameworkInitializationCompleted) veszi át. A korábbi StartupUri beállítás figyelmen kívül marad.

- A metódus futása során a rétegek példányosítása szigorúan meghatározott sorrendben történik:
- Első lépésként a LabirintusViewModel (Nézetmodell) példányosítására kerül sor. Mivel a Nézetmodell felelős a Modell és a Perzisztencia rétegek létrehozásáért és kezeléséért, ezzel a lépéssel a teljes üzleti logika és adatkezelés inicializálódik.
- Ezt követően hozzuk létre a felhasználói felületet biztosító MainWindow (Nézet) példányt.
- Végül a két réteget manuálisan kapcsoljuk össze: a Nézet DataContext tulajdonságának értékül adjuk a korábban létrehozott Nézetmodellt (_view.DataContext = _viewModel;), majd megjelenítjük a főablakot.
- Ez a szerkezet biztosítja, hogy a Nézet már a megjelenés pillanatában rendelkezzen a működéshez szükséges érvényes adatokkal és logikával



Tesztelés:

A modell funkcionalitása egységtesztek segítségével lett ellenőrizve a MotorTest osztályban. A teszt során egy kis méretű labirintus térkép szolgál a mozgás, állapotkezelés és események vizsgálatára. A kezdőpozíció és a kijárat ismerten kerül beállításra, így jól ellenőrizhetőek a játékmenet szabályai.

Az alábbi tesztesetek kerültek megvalósításra:

- Motor_Constructor_InitializesPropertiesCorrectly: ellenőrzi, hogy a konstruktor helyesen inicializálja a labirintust, a játékos pozícióját, valamint az aktív állapotot.
- Start_ActivatesGameAndRaisesGameChangeEvent: vizsgálja, hogy a játék elindítása aktiválja-e a modellt, valamint kiváltja-e a megfelelő eseményt.
- PauseAndResume_TogglesIsActiveState: a szüneteltetés és folytatás helyes működését ellenőrzi, vagyis hogy a játék aktív állapota megfelelően váltakozik.
- Move_WhenActive_ChangesPlayerPositionAndRaisesEvent: teszteli, hogy aktív játék közben a mozgás helyesen frissíti a játékos pozícióját, és eseményt generál.
- Move_WhenPaused_DoesNotChangePosition: szüneteltetett állapotban a játékos pozíciója nem változik, a mozgás tiltott.

- Move_IntoWall_DoesNotChangePosition: fal irányába történő mozgás esetén a játékos pozíciója változatlan marad.
- Move_OutOfBounds_DoesNotChangePosition: a pálya határain kívülre nem engedi a rendszer a játékos kilépését.
- Move_ToExit_RaisesGameOverEvent: ellenőrzi, hogy a kijárat elérésével a játék véget ér, és a megfelelő esemény kiváltásra kerül.