

**Készítette:**

Fikó Róbert  
Neptun kód G55OFZ  
Csoport: 15-ös csoport  
E-mail: [g55ofz@inf.elte.hu](mailto:g55ofz@inf.elte.hu)  
Személyes e-mail: [fiko.robert+elte@gmail.com](mailto:fiko.robert+elte@gmail.com)

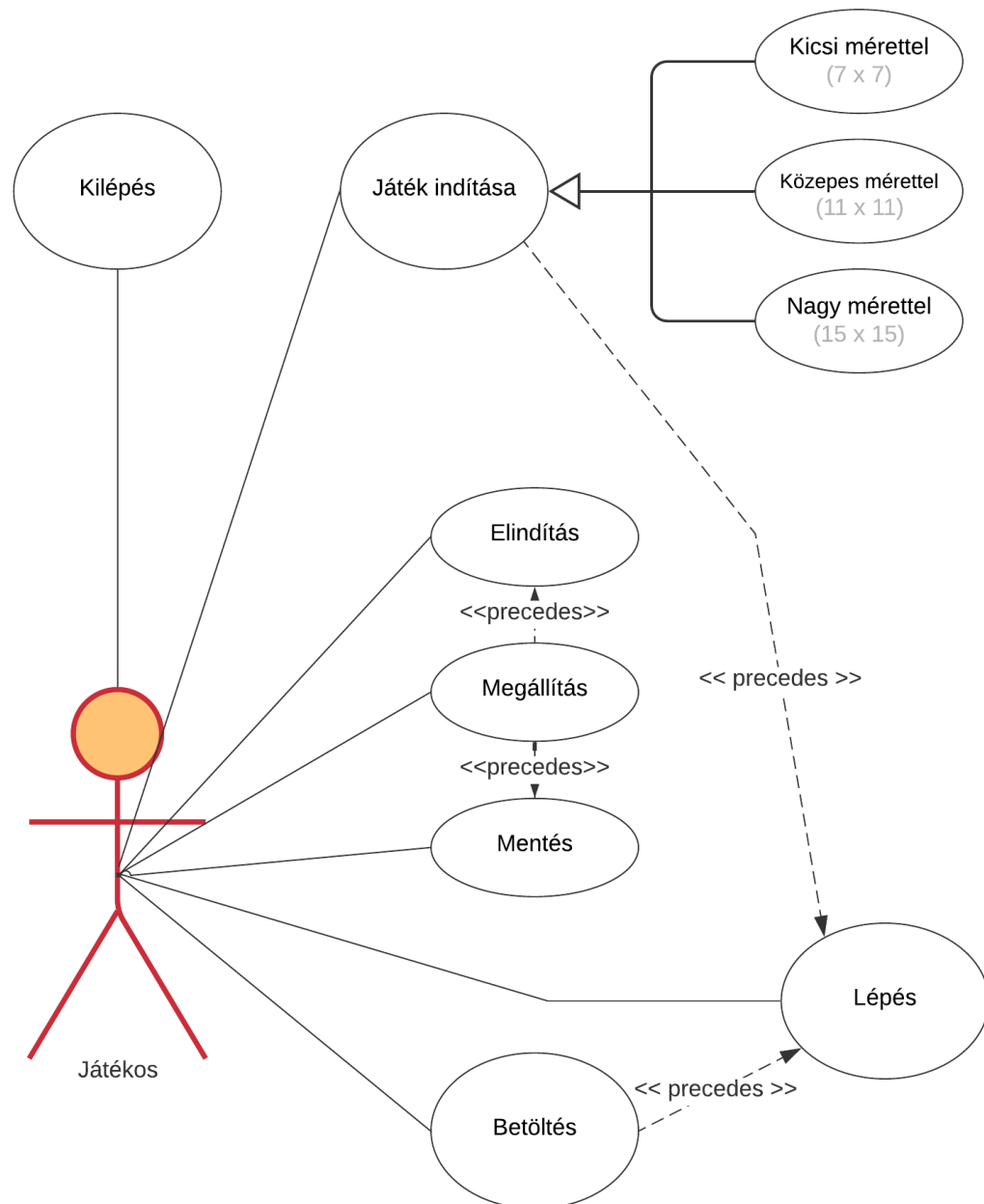
**Feladat leírása**

Készítsünk programot, amellyel a következő játékot játszhatjuk. Adott egy  $n \times n$  mezőből álló játékpálya, amelyben egy elszabadult robot bolyong, és a feladatunk az, hogy beteretljük a pálya közepén található mágnes alá, és így elkapjuk. A robot véletlenszerű pozícióban kezd, és adott időközönként lép egy mezőt (vízszintesen, vagy függőlegesen) úgy, hogy általában folyamatosan előre halad egészen addig, amíg falba nem ütközik. Ekkor véletlenszerűen választ egy új irányt, és arra halad tovább. Időnként még jobban megkergül, és akkor is irányt vált, amikor nem ütközik falba. A játékos a robot terelését úgy hajthatja végre, hogy egy mezőt kiválasztva falat emelhet rá. A felhúzott falak azonban nem túl strapabíróak. Ha a robot ütközik a fallal, akkor az utána eldől. A ledőlt falakat már nem lehet újra felhúzni, ott a robot később akadály nélkül áthaladhat. A program biztosítson lehetőséget új játék kezdésére a pályaméret megadásával ( $7 \times 7$ ,  $11 \times 11$ ,  $15 \times 15$ ), valamint játék szüneteltetésére (ekkor nem telik az idő, nem lép a robot, és nem lehet mezőt se kiválasztani). Ismerje fel, ha vége a játéknak, és jelenítse meg, hogy milyen idővel győzött a játékos. A program játék közben folyamatosan jelezze ki a játékidőt. Ezen felül szüneteltetés alatt legyen lehetőség a játék elmentésére, valamint betöltésére.

**Elemzés**

- A játékot három pályamérettel játszhatjuk:  $7 \times 7$ ,  $11 \times 11$ ,  $15 \times 15$ , a célunk minden esetben ugyanaz: a robotot becsalni a mágnes alá.
- A feladatot egyablakos asztali alkalmazásként Windows Presentation Foundation grafikus felülettel valósítom meg.
- Az ablak felső sorában elhelyezésre kerül egy menüsáv a következő opciókkal:
  - New game
    - Size:  $7 \times 7$  (Méret)
    - Size:  $11 \times 11$  (Méret)
    - Size:  $15 \times 15$  (Méret)
  - Game menu
    - Pause (Megállít)
    - Play (Folytatás)
    - Save (Mentés)
    - Load (Betöltés)
- Az ablak alján egy státusz sort jelenítünk meg:
  - Kezdő képernyő esetén közli, hogy nincsen éppen futó játék
  - Játék alatt kijelzi az eltelt időt
  - Megállított játék alatt közli a megállítást tényét

- A táblát tulajdonképpen egy nyomógomb ráccsal valósítjuk meg (7,11 vagy 15-ös mérettel). A nyomógombra kattintáskor, az általa reprezentált mezőre fal kerül. A mágnesre és a robotra való kattintás esetén semmi nem történik.
- A játék képes érzékelni mikor vége a játéknak, ekkor feldob egy dialógus ablakot, hogy nyertünk, s mennyi idővel.
- Mikor betöltünk vagy mentünk egy játékot a program ennek sikerességét, vagy sikertelenség esetén a hiba okát dialógusban jelzi.
- A felhasználói eseteket az ábra bemutatja:

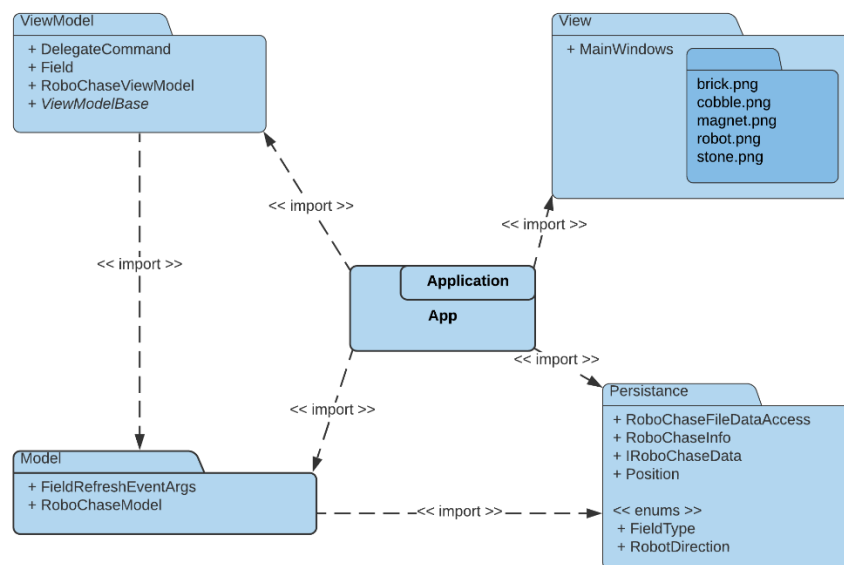


## Tervezés

### Programszerkezet

A programot MVVM architektúrában valósítjuk meg. A megjelenés a `RoboChase.View`, a megjelenés mögötti logika a `RoboChase.ViewModel`, a modell a `RoboChase.Model`, míg a perzisztencia a `RoboChase.Persistence` névtérben helyezkedik el. A program környezetét az `App` osztály biztosítja a program környezetét, példányosítja a modellt, a nézetmodellt, a nézetet, biztosítja a kommunikációt, valamint felügyeli az adatkezelést.

Az alábbi ábra illusztrálja



### Perzisztencia

- Az adatkezelés feladata a játéktáblával kapcsolatos információk tárolása, illetve a betöltés és a mentés megvalósítása, biztosítása
- A `RoboChaseInfo` osztály minden információt tartalmaz, amivel meg lehet konstruálni egy játékmodellt, majd egy nézetet
- A játék információs osztályon keresztül lehetőségünk van minden játék adatot manipulálni, beállítani adattagokon és ún. Propertyken keresztül
- A játék információs osztályon a módosítások elvégzésére a játékmodell osztály áll rendelkezésünkre.
- A játék kimentését (így a hosszabb távon való eltárolását az `IRoboChaseData` interfész biztosítja).
- Az interfészt szöveges (`*.crazy`) fájlok olvasását és mentését a `RoboChaseDataAccess` osztály valósítja meg, az ezen folyamatok közben fellépő hibákat hamis visszatérési értékkel jelezzük
- Ezen fájlokban tárolt adatokat itt a mintának megfelelően kell formáznunk (ábrát lásd a következő oldalon)

example.crazy

```

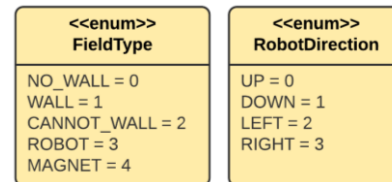
7          tábla mérete
2 4        robot pozíciója
1          robot iránya
0          mező típusa
20         eltelt idő
3          idő irányváltóztatásig
1 0 0 0 0 0 0  transzponált tábla mtx.
0 0 2 0 0 0 0
0 0 0 0 0 0 0
0 0 0 4 0 0 0
0 0 0 0 0 0 0
0 1 0 0 0 0 0
0 0 0 0 0 0 1

```

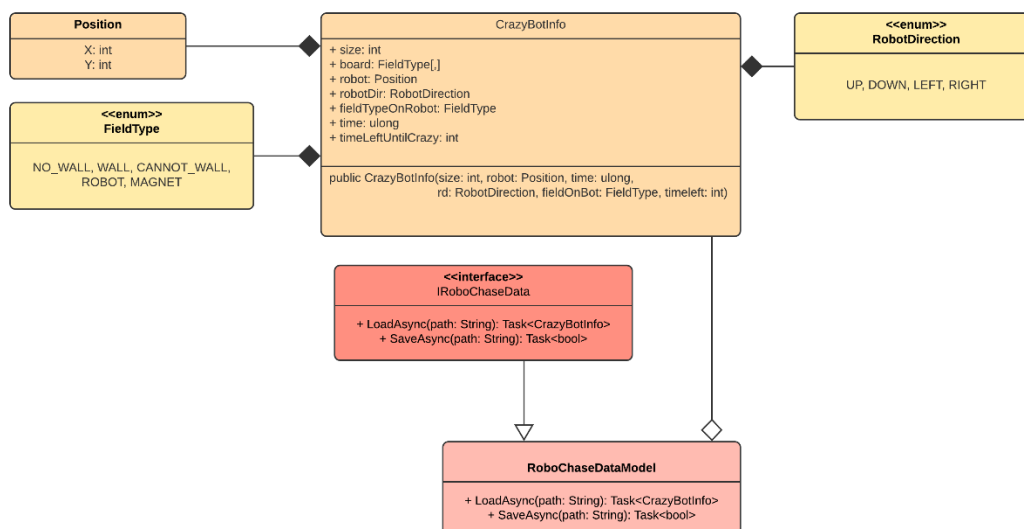
Az első sor megadja a tábla méretét, majd következik a robot pozíciója, amit a robot iránya követ.

Majd a mező típusa, amin a robot áll. Ezek után megadjuk az eltelt időt, majd a véletlenszerű irányváltóztatásig hátralévő időt.

Végül a táblát leíró mátrix következik transzponálva.

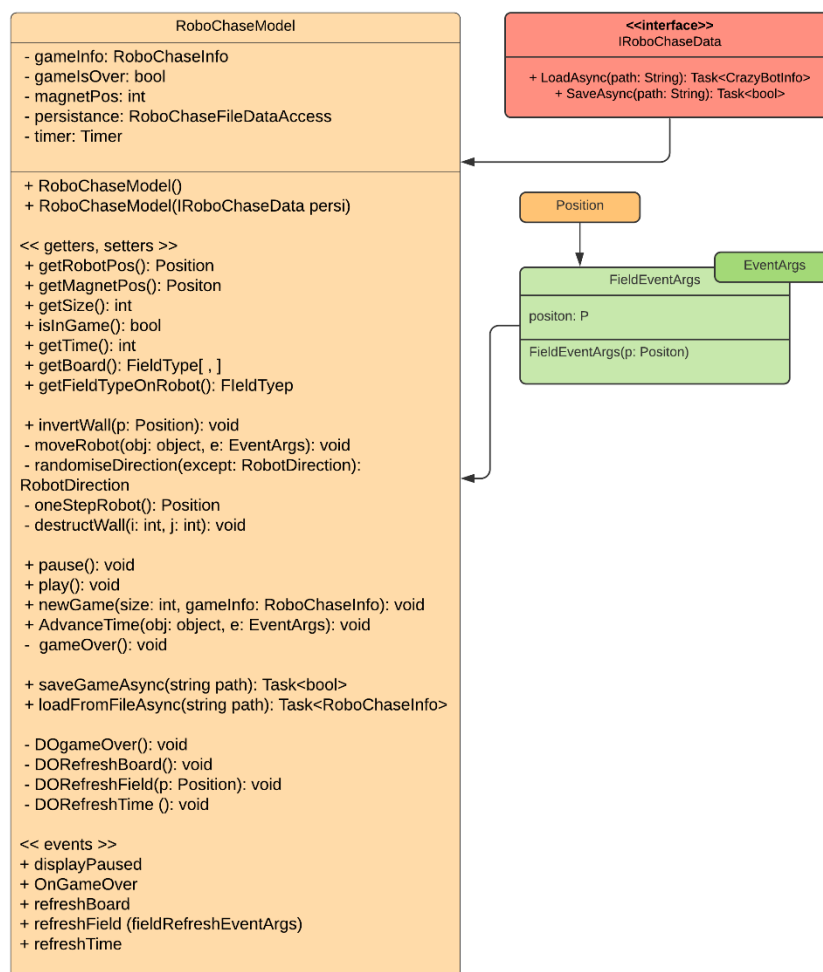


A fejlesztés során, úgynevezett enum-okat, felsorolókat vezettem be, hogy a kód olvashatóbb legyen, de a kimentés során ezen felsorolási értékeket az általuk reprezentált egész szám voltában iratjuk ki.



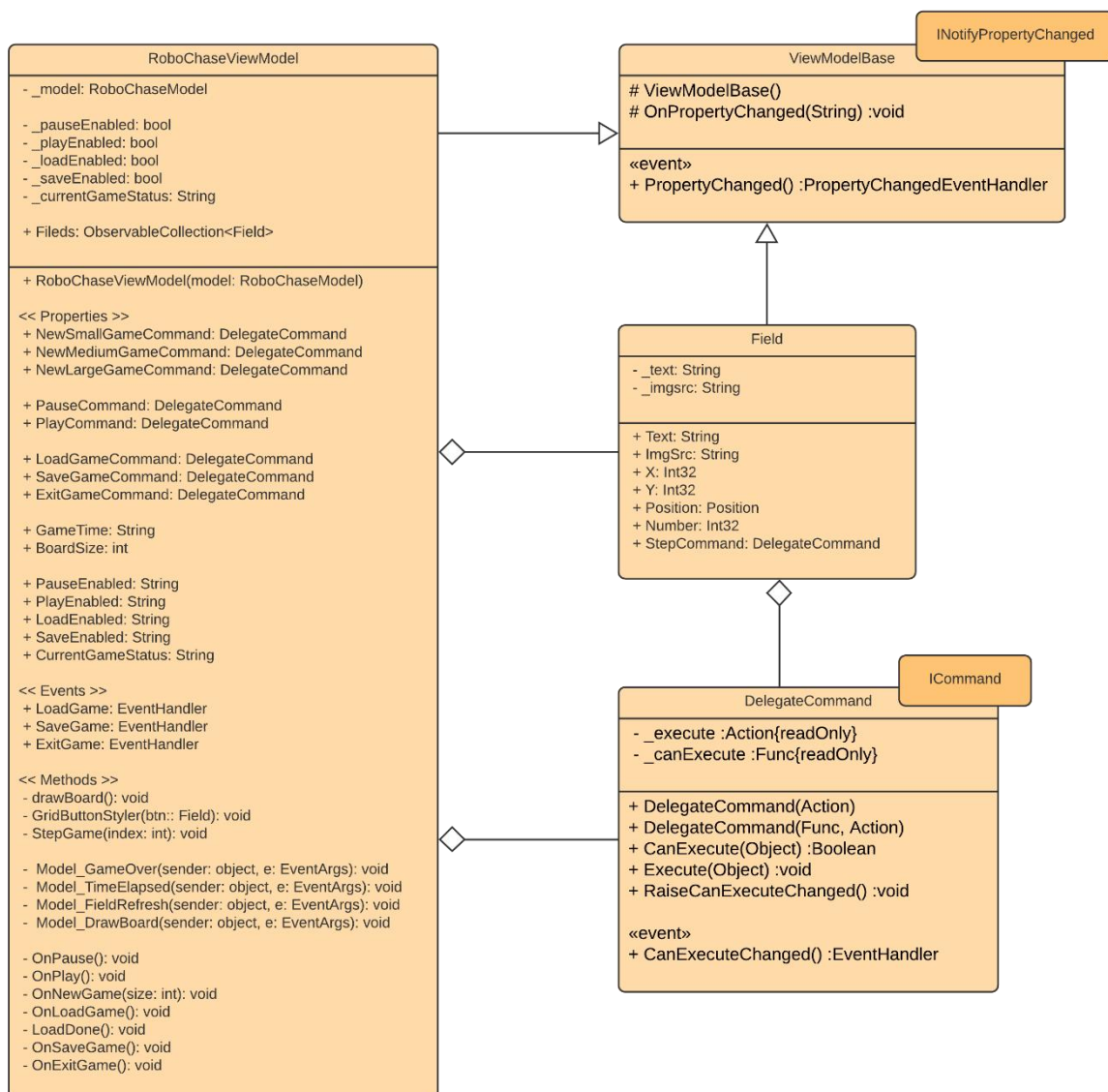
## Modell

- A modell egy igen jelentős részét a `RoboChaseModel` osztály valósítja meg, amely reagál a játéktáblán történt eseményekre. Ezen kívül szabályozza a játék egyéb paramétereit, mint az eltelt időt (`time`). Az osztály lehetőséget ad új játék kezdésére a `newGame` metódusa segítségével, valamint fal letételére (`invertWall`). Az időléptetése, az `AdvanceTime` metódussal történik, aminek függvényében a robot is lépni fog.
- A modell a nézet felé tudja jelezni, hogy szükséges a *teljes* tábla frissítése (`refreshBoard`), ezt megtudja tenni csupán egy mezővel is, ami amennyiben csak néhány mező változott, egy hatékonyabb kivitel, ezt a `refreshField` eseményen keresztül tudja megtenni. Ehhez az eseményhez tartozik egy saját event argumentum is, a `fieldRefreshEventArgs`, ami tartalmaz egy `Position`-t ami pedig információval látja el a nézetet, hogy melyik mezőt is kellene újrarajzolni
- Amikor a modell `newGame` metódusát meghívjuk paraméterül átadhatunk neki, egy `gameInfo` osztályt (ami persze opcionális), ebben az esetben a játékot a paraméterül megadott játék állással fogja inicializálni.
- A játék időbeli kezelését egy időzítő végzi, amelyet mindig aktiválunk játék során, illetve inaktíválunk.








## Nézetmodell

- A nézetmodell megvalósításához felhasználtunk egy általános utasítás (**DelegateCommand**), valamint egy ős változásjelző (**ViewModelBase**) osztályt.
- A nézetmodell feladatait a **RoboChaseViewModel** osztály látja el, amely parancsokat biztosít az új játék kezdéséhez, játék betöltéséhez, mentéséhez, valamint a kilépéshez. A parancsokhoz eseményeket kötünk, amelyek a parancs lefutását jelzik a vezérlőnek. A nézetmodell tárolja a modell egy hivatkozását (**\_model**), de csupán információkat kér le tőle, illetve a játéknehézséget szabályozza. Direkt nem avatkozik a játék futtatásába, leszámítva, mikor a játkos megnyom egy gombot, s ezt közvetíti a modellnek.
- A játéklemező számára egy külön mezőt biztosítunk (**Field**), amely eltárolja a pozíciót, valamint a lépés parancsát (**StepCommand**). A mezőket egy felügyelt gyűjteménybe helyezzük a nézetmodellbe (**Fields**).



## Nézet

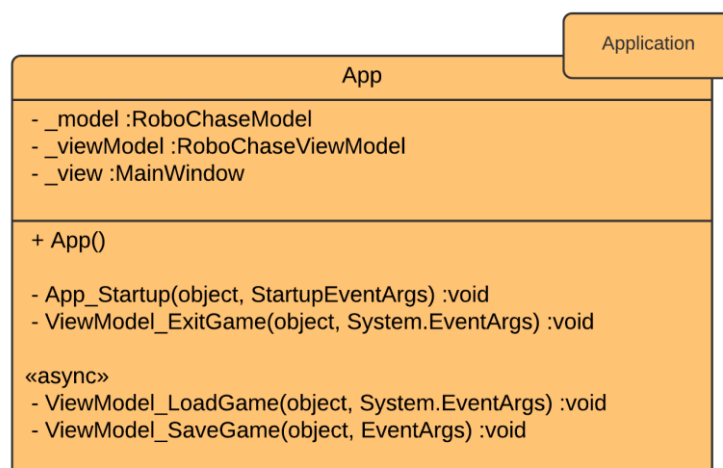
- A nézet csak egy képernyőt tartalmaz a `MainWindow` osztályt.
- A fájlnev bekérését betöltéskor és mentéskor, valamint a figyelmeztető üzenetek megjelenését beépített dialógusablakok segítségével végezzük.
- A játéktáblát egy dinamikusan megvalósított nyomógomb rács reprezentálja. A felület tetején menüsor helyezkedik el, ami megegyezik a tervezésben leírtakkal, az alján pedig egy státuszsor, ami szintén hasonló paraméterezéssel rendelkezik.
- A nézet a megfelelő enumerációs értékekhez az alábbi textúrákat rendeli hozzá:

FieldType	NO_WALL 0	WALL 1	CANNOT_WALL 2	ROBOT 3	MAGNET 4
Texture name	noWallTexture	WallTexture	cannotWallTexture	robotTexture	magnetTexture
Texture					

- *Megjegyzés: a robotot a szoftver maszkolja rá a CANNOT\_WALL és a NO\_WALL textúrákra. A mágnes esetében ugyan ez a helyzet, bár a mágnes nem lehet már lerombolt (CANNOT\_WALL) helyen, mivel a mágnesre nem tehetünk falat, ahogy a robotra sem, viszont a robot áthaladhat lerombolt falon.*

## Környezet

- Az `App` osztály feladata az egyes rétegek példányosítása (`App_Startup`), összekötése, a nézetmodell, valamint a modell eseményeinek lekezelése, és ezáltal a játék, az adatkezelés, valamint a nézetek szabályozása



## Tesztelés

A tesztelés a CrazyBotTest osztály segítségével, MS Test rendszer Unit, azaz egység tesztei segítségével lett megvalósítva, melyet az alábbi táblázat összegez:

### LoadCheck

Ellenőrzi, hogy a játék megfelelően áll e fel: megfelelő mérettel, megfelelő információs (tábla) osztállyal jön e létre a model, **amikor fájlból** töltünk be, **mockolt** perzisztencia réteggel.

### ConstructorCheck

Ellenőrzi, hogy a játék megfelelően áll e fel: megfelelő mérettel, megfelelő információs (tábla) osztállyal jön e létre a model, s az idő valóban telik.

### MoveRobot

A robot mozgását teszteli mind a négy lehetséges irányba, az időzítő tick-elését szimulálva.

### PlaceWalls

Teszteli a falak letételét helyes (NO\_WALL) helyre, és helytelen (CANNOT\_WALL, MAGNET, ROBOT), illetve már letett falak felvételének megpróbálása.

### HitAndPlaceWall

Falak letétele a pályára és robot nekivezetése annak, mind a négy irányból, a fal ledőlésének ellenőrzése, és a robot irányváltztatásának ellenőrzése

### HitEdge

A robot a pálya szélének vezetése, és visszapattanás ellenőrzése.

### WalkOnCannotWall

Annak ellenőrzése, hogy a robot valóban áttud-e menni korábban már ledöntött falakon.

### RobotGotMagnet

A robot mágnes pozíciójába vezetése, ezáltal a játék végének kiváltása.

### RobotRandomChange

A robot véletlen időközönként irányt vált. Ennek ellenőrzésére szolgál ez a teszt eset.