|  |  |  |
| --- | --- | --- |
|  | **Eötvös Loránd Tudományegyetem**  Informatikai Kar  Algoritmusok és Alkalmazásaik Tanszék |  |

**Sci-fi kolónia szimulátor**

|  |  |
| --- | --- |
| *Témavezető:*  Kovácsné Pusztai Kinga Emese  Tanársegéd | *Szerző:*  Nagy Richárd Tibor  Programtervező Informatikus BSc |

Budapest, 2018

Tartalomjegyzék

[Bevezetés 5](#_Toc513324090)

[Felhasználói dokumentáció 6](#_Toc513324091)

[Rendszerkövetelmények 6](#_Toc513324092)

[Telepítés 7](#_Toc513324093)

[Indítás 7](#_Toc513324094)

[Célközönség 8](#_Toc513324095)

[Játékmenet 10](#_Toc513324096)

[A játék célja 10](#_Toc513324097)

[Kamera 11](#_Toc513324098)

[Robotok 11](#_Toc513324099)

[Épületek 12](#_Toc513324100)

[Ellenségek 15](#_Toc513324101)

[Munkák 16](#_Toc513324102)

[Fejlesztői dokumentáció 17](#_Toc513324103)

[Elemzés 17](#_Toc513324104)

[Fejlesztői környezet 17](#_Toc513324105)

[Felhasználói esetek diagramja 18](#_Toc513324106)

[Felhasználói esetek leírása 19](#_Toc513324107)

[A komponensek diagramja 20](#_Toc513324108)

[Modell 21](#_Toc513324109)

[Edge 21](#_Toc513324110)

[Node 21](#_Toc513324111)

[Graph 22](#_Toc513324112)

[PriorityQueue<T> 23](#_Toc513324113)

[Pathfinder 24](#_Toc513324114)

[IPrototypeable 25](#_Toc513324115)

[PrototypeManager<T> 26](#_Toc513324116)

[Prototypes 27](#_Toc513324117)

[Building 28](#_Toc513324118)

[Difficulty 30](#_Toc513324119)

[Enemy 31](#_Toc513324120)

[Job 33](#_Toc513324121)

[JobManager 35](#_Toc513324122)

[Projectile 37](#_Toc513324123)

[Robot 39](#_Toc513324124)

[Tile 42](#_Toc513324125)

[World 44](#_Toc513324126)

[Perzisztencia 48](#_Toc513324127)

[Nézet 49](#_Toc513324128)

[IDisplayable 49](#_Toc513324129)

[SpriteManager 50](#_Toc513324130)

[View<T> 51](#_Toc513324131)

[ViewManager 52](#_Toc513324132)

[BuildingView 52](#_Toc513324133)

[EnemyView 53](#_Toc513324134)

[JobView 54](#_Toc513324135)

[ProjectileView 54](#_Toc513324136)

[RobotView 55](#_Toc513324137)

[TileView 55](#_Toc513324138)

[WorldView 56](#_Toc513324139)

[CameraController 58](#_Toc513324140)

[MouseController 59](#_Toc513324141)

[Tesztelés 60](#_Toc513324142)

[Egységtesztek 60](#_Toc513324143)

[Végfelhasználói tesztek 60](#_Toc513324144)

[Skálázhatóság 61](#_Toc513324145)

[Egyéb fejlesztési lehetőségek 61](#_Toc513324146)

## Ajánlás

Ezt a dolgozatot szeretett szüleimnek dedikálom, akik egész életemben mellettem álltak és támogattak.

# Bevezetés

Szakdolgozatom során szerettem volna egy olyan projektet véghez vinni, mely során az egyetemi éveim alatt szerzett tudásom jelentős részének felhasználása mellett új és izgalmas eszközökkel és módszerekkel ismerkedhetek meg.

A lehetőségek átgondolása után egy komplex játékszoftverre esett a választásom. Ezen keresztül van alkalmam már tanult gráf algoritmusok alkalmazására és újak implementálására, elmerülhetek a valós idejű útvonalkeresés rejtelmeiben, belekóstolhatok a mesterséges intelligencia világába, foglalkozhatok véges állapotgépekkel, kipróbálhatom magam egy összetett program optimalizálásában, megismerkedhetek a C# által nyújtott legmodernebb nyelvi elemekkel, megtanulhatom az XML fájlformátum írásának és olvasásának módját, valamint bemutathatom a verziókövető rendszerekkel kapcsolatos tapasztalataimat.

# Felhasználói dokumentáció

## Rendszerkövetelmények

A játék a Unity motorra épült, így a futtatáshoz szükséges hardver követelmények ehhez igazodnak. A minimális igények a következők:

* Windows Vista Service Pack 1, vagy annál újabb Microsofttól származó operációs rendszer.
* Egy DirectX 10 (shader model 4.0) kompatibilis grafikus kártya (gyakorlatilag az összes, 2006 óta gyártott fogyasztói GPU rendelkezik ezzel a képességgel).
* Streaming SIMD Extensions 2 utasításkészlettel ellátott processzor. A 2001-ben kiadott Pentium 4 névre hallgató CPU már rendelkezett ezzel a technológiával.
* Legalább 50 MB szabad hely a háttértáron

Ugyan a fentiek elegek a program futtatásához, a sima játékélmény eléréséhez a következőket ajánlom:

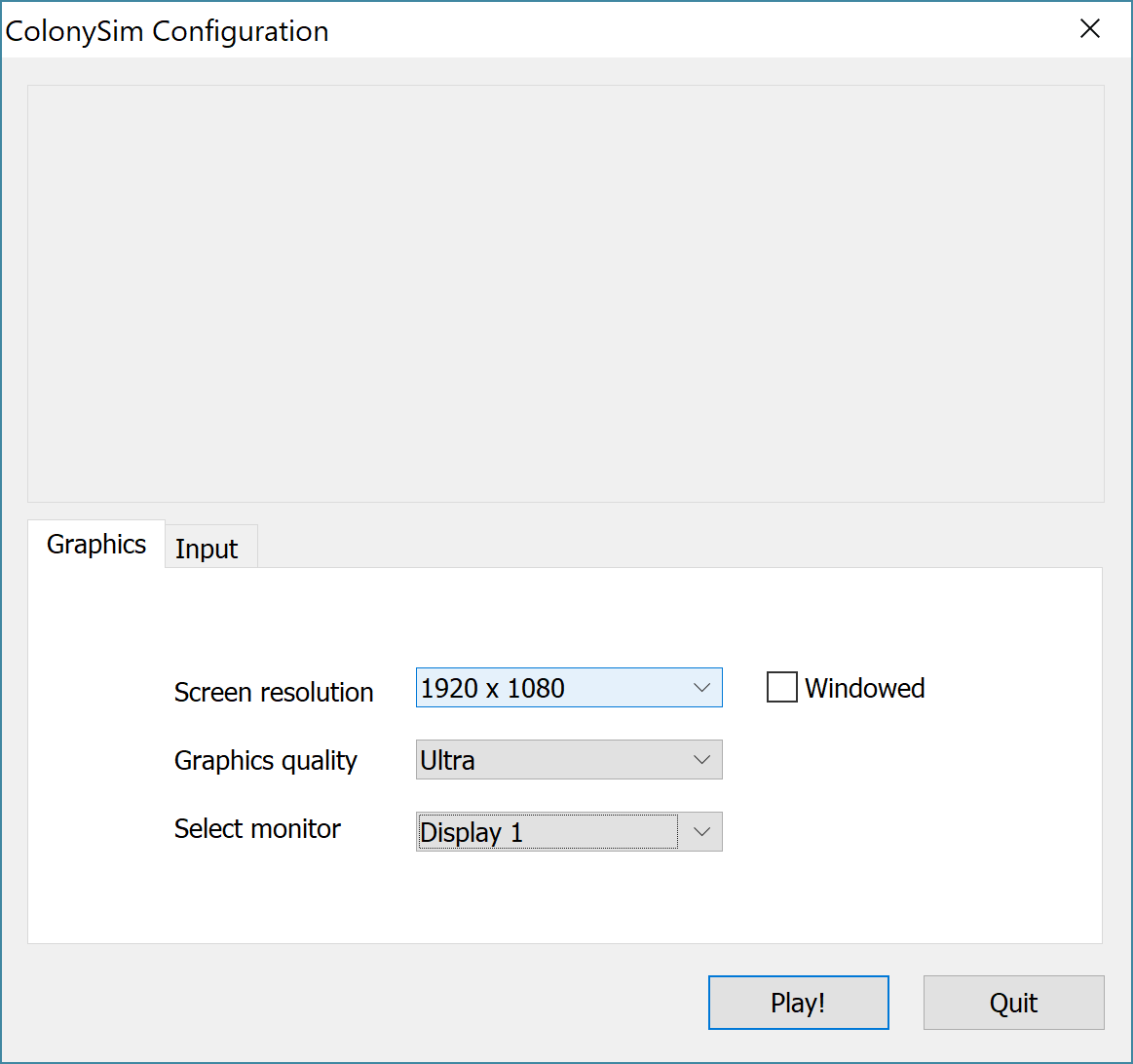
* 2 magos, legalább 2.0 GHz-es processzor
* 256 MB grafikus memóriával rendelkező videókártya
* 2 GB szabad rendszermemória

A felhasználó a saját számítógépe képességeihez mérten beállíthatja a játék felbontását és az egyéb grafikai opciókat a zökkenőmentes játékmenet érdekében.

## Telepítés

A program teljesen önálló, helyes működéséhez nem szükséges semmilyen más szoftvert telepíteni. A főkönyvtárban megtalálható a futtatható állomány, az erőforrásfájlokat tároló colonysim\_Data könyvtár, valamint mellékelve vannak a szükséges .dll kiterjesztésű fájlok.

## Indítás

A futtatható, colonysim.exe fájlra duplán kattintva a tetszőleges fájlkezelőben (a program egyébként parancssorból is indítható, paramétereket, kapcsolókat nem vár) az alábbi konfigurációs ablak jelenik meg:

Az egyes menüpontokkal a következők állíthatók:

* **Screen resolution**: A játék felbontása. Ajánlott a monitor natív felbontásával megegyező opciót választani. Amennyiben a játék nem a felhasználó igényeinek megfelelően fut, érdemes csökkenteni a felbontást.
* **Windowed**: Bepipálásával eldönthető, hogy a program teljes képernyős üzemmódban, vagy ablakosan fusson.
* **Graphics quality**: A Unity motor különböző grafikus utófeldolgozási szintjei között választhatunk. Minél nagyobbra állítjuk, a játék annál szebb, viszont a rendszer számára is egyre megterhelőbb.
* **Select monitor**: Kiválaszthatjuk, hogy a program melyik monitoron jelenjen meg.
* A beállítások befejezése után a játék a **Play** gomb megnyomásával indítható.
* Kilépésre is van lehetőség, a jobb felső sarokban lévő **X**, vagy a **Quit** gombra való kattintással.

## Célközönség

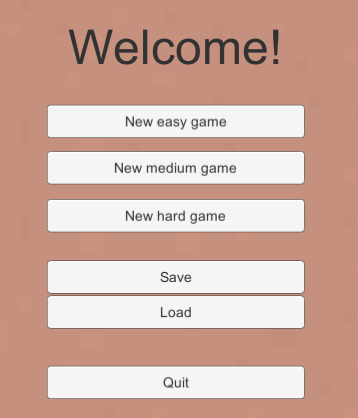
A játék tanulási görbéje egyáltalán nem meredek, a szabályok és funkciók mindenki számára könnyen és gyorsan megtanulhatók, így a programot az összes, számítógépet használni tudó embernek tudom ajánlani.

Az egyes játékszessziók nem igényelnek sok időt, a maximális, megszakítás nélküli menetidő 5 perc. Ez által nincs szükség semmiféle hosszú távú elkötelezettségre, a játék bármikor, tetszőlegesen kevés időre is igénybe vehető. Az előbbieket egészíti még ki a lehetőség a játékállapot elmentésére és betöltésére, így az bármikor abbahagyható és később ugyanonnan, a progresszió elvesztése nélkül folytatható.

Érdemes még megemlítenem, hogy a program kezelőfelülete angol nyelven készült el. Amennyiben a felhasználó nem tud angolul, ajánlom a felhasználói dokumentáció átolvasását, ahol minden funkció leírása megtalálható.

Főmenü

A konfigurációs ablakon a **Play** gombra kattintva a játék elindul, és a következő menüvel fogadja a felhasználót:



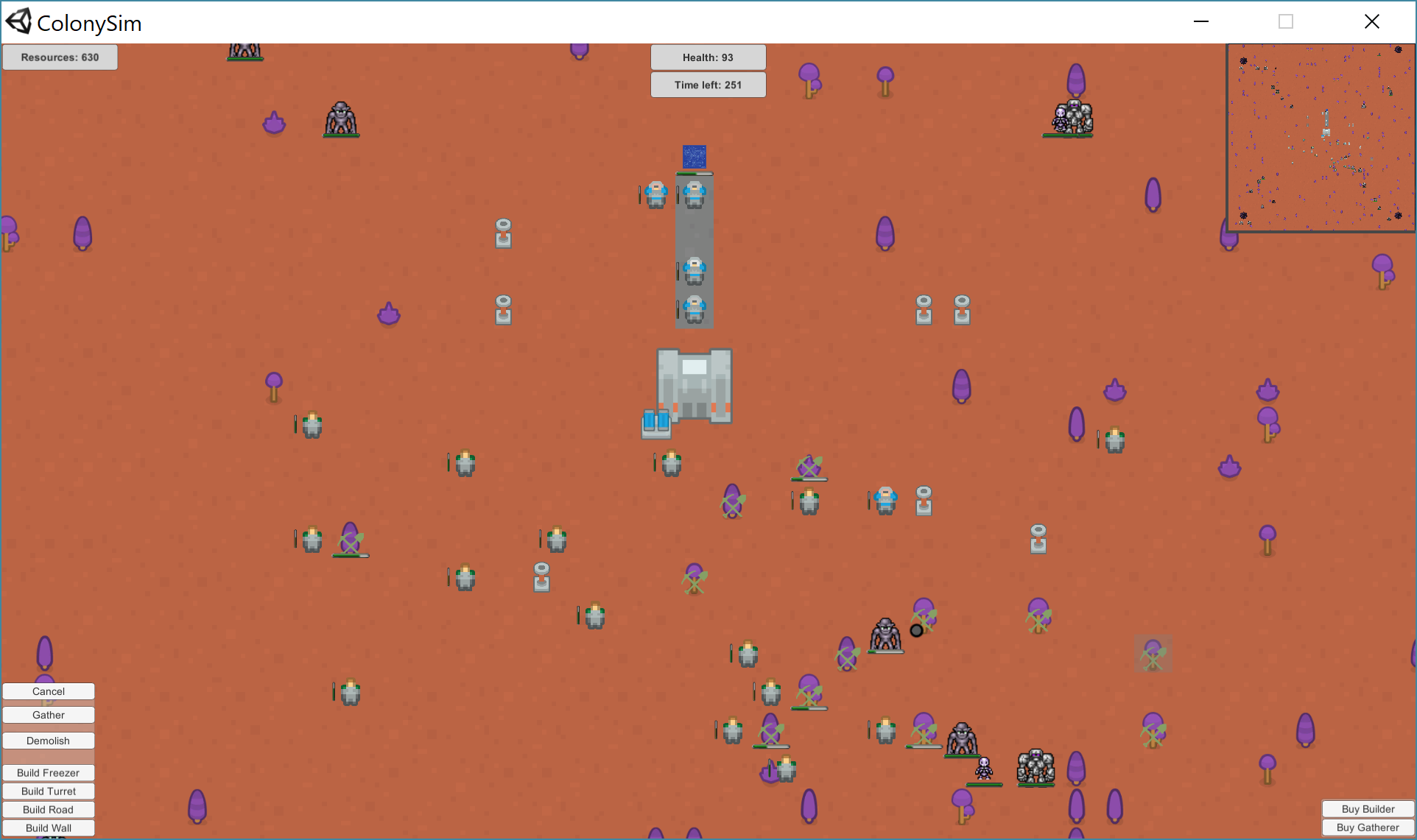
A felső három menüpont új játék kezdésére szolgál, rendre könnyű, közepes és nehéz fokozaton. A nehézségi szintek közötti különbségeket az alábbi táblázat mutatja be:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nehézség** | **Kezdeti erőforrások** | **Túléléshez szükséges idő** | | **A térképen szétszórt erőforrások** | **Ellenségek termelődési pontjai** |
| Könnyű | 700 | 3:00 | 7500 | | 1 |
| Közepes | 600 | 4:00 | 6250 | | 2 |
| Nehéz | 500 | 5:00 | 5000 | | 4 |

* A **Save** gombbal elmenthető az aktuális játékállás.
* A **Load** gombra való kattintással pedig egy korábban mentett játék tölthető be, és folytatható.
* A **Quit** gomb lehetővé teszi a szoftverből való kilépést.

A főmenü játék közben is bármikor elérhető az **Escape** billentyű lenyomásával. Az idő ilyenkor megáll, ezáltal a menü egyben szüneteltetési funkcióként is szolgál. Az Escape ismételt lenyomásával a menü eltűnik, a játék pedig folytatható onnan, ahol abbahagytuk.

## Játékmenet



### A játék célja

A játék célja a mezőkből álló pálya közepén található főhadiszállás megvédése a támadó ellenségektől a rendelkezésre álló eszközök segítésével. Ha egy ellenség eléri a bázist, saját hátralévő életerejének megfelelő kárt okoz benne. Amennyiben a főhadiszállás élete (a kezelőfelületen felül, középen látható az aktuális érték) nullára csökken, a játék véget ér. Ha ezt sikerül elkerülni a nehézségtől függően beállított ideig, a felhasználó nyer. A hátralévő idő szintén a felület felső részén, középen látható.

A játékosnak célja eléréséhez több eszközt is igénybe vehet. A felület bal felső részén figyelemmel követheti, hogy mennyi erőforrás áll jelenleg a rendelkezésére. Ezt több módon is felhasználhatja. A jobb alsó panelen vásárolhat robotokat, melyeknek a bal alsó panelen különféle feladatokat oszthat ki. Ezek, valamint az ellenségek és az épületek részletezésére külön szekcióban kerül sor.

Figyelni kell azonban arra, hogy az ellenségeknek mindig legyen szabad útja a főhadiszálláshoz, mert amennyiben nem találnak útvonalat, a játék vereséggel véget ér.

A játékost segíti még a jobb felső sarokban elhelyezett kistérkép, ami az egész pályát lefedi, és folyamatosan, hasznos információkkal látja el a felhasználót az aktuális történésekről.

### Kamera

A kamera irányítása roppant egyszerű: amennyiben az egér a képernyő széléhez ér, a kamera abba az irányba fog mozogni. Lehetőség van közelítésre és távolításra is, ezt a funkciót az egér görgőjével tudjuk kihasználni.

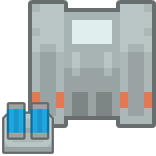
### Robotok

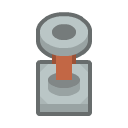
A bázis védelmének legfőbb eleme a munkák végrehajtásán dolgozó robotok. A játékosnak jelenleg két típusú robot áll rendelkezésére. Az egyik a gyűjtögető, aminek célja a beszerzésre kijelölt fák erőforrásainak kinyerése, valamint az építő, amely a különböző épületek építésével vagy esetleges lerombolásával foglalkozik. A gépek önállóan dolgoznak, közvetlenül nem irányíthatók. Elvállalják az általuk végezhető, legkorábban kiadott munkát, majd megpróbálják teljesíteni azt. Töltődési szinttel is rendelkeznek (az aktuális szint a robot mellett látható), emiatt időnként töltésre van szükségük. Ehhez visszatérnek a főhadiszállásra, majd miután készen állnak, folytatják tevékenységüket. Amennyiben nem tudnak feltöltődni, megsemmisülnek.

Az egyes robottípusok statisztikái a következők:

* A **konstrukciós** robotok 25 erőforrásba kerülnek, másodpercenként pedig 20 egységnyit tudnak mozogni.
* A **gyűjtögető** robotok olcsóbbak, 20 erőforrásért vásárolhatók meg, és valamivel gyorsabbak is, 30 egységet képesek haladni másodpercenként

### Épületek

A játék több, egyedi épületet tartalmaz. Ezek mind különböző funkciókkal, tulajdonossal, mérettel, árral és viselkedéssel rendelkeznek. Az épületfajták és azok tulajdonságai az alábbiak:

* A **főhadiszállás** a játék legfontosabb épülete. A felhasználó célja a megvédése, az ellenségeké pedig az elérése és elpusztítása. Az újonnan megvásárolt robotok innen kerülnek ki, és a töltődéshez is ide térnek vissza. 3x3 területet foglal el, 100 életerő ponttal rendelkezik, átmászni pedig nem lehet rajta. Újra nem építhető és le sem rombolható, a játékos a pálya kezdetén egyet kap. Mindig a térkép közepén helyezkedik el.
* ****A **falak** a védelemben nagy szerepet töltenek be. Megakadályozzák az ellenségek áthatolását, azonban a robotoknak is ki kell őket kerülni. A konstrukciós robotok tudják felépíteni. 50 erőforrásba kerülnek, csak üres mezőre építhetők és 3 másodpercig tart az elkészítésük. Előnyös tulajdonságuk, hogy a szomszédaikkal összekapcsolódva egy összefüggő épületet alkotnak.
* A fenti képen látható, és az ahhoz hasonló **fák** szolgáltatják a játékos erőforrásait. A felhasználó által kijelölhetők gyűjtésre, ami után a gyűjtögető robotok elvégzik a kitermelést. Ez 3 másodpercbe telik, és 25 erőforrást nyújt a játékosnak. A játék kezdetén a nehézségi beállítástól függő mennyiségű fa lesz elhelyezve, és ugyan a kitermelés során néhány újra nő, előbb utóbb az összes elfogy.
* Az **utak** kétélű kardként viselkednek, hiszen mind a robotok, mind az ellenségek mozgását meggyorsítják 25%-kal. Emiatt érdemes olyan helyre építeni őket, ahol csak a robotok járnak, és az ellenségek nem tudják felhasználni őket az útjuk során a bázishoz. Az építő robotok tudják őket létrehozni, ami 25 erőforrásba és 3 másodpercbe kerül. Ha olyan mezőre épül, aminek van úttal már rendelkező szomszédja, akkor azzal összekapcsolódva egybefüggő épületként jön létre.
* A **spawner** az ellenségeket folyamatosan létrehozó épület. Innen kezdik meg útjukat a főhadiszállás felé. 3x3 mezőt foglalnak el, nem lehet rajtuk áthaladni, és nem is lehet őket lerombolni. A játék nehézségi beállításától függően rendre 1, 2, vagy 4 darab jön létre a pálya egyes sarkaiban. Ahogy csökken a játék megnyeréséig hátralevő idő, annál kisebb időközönként hoznak létre új ellenségeket, ezáltal dinamikusan, egyre jobban nő a játék nehézsége.
* A **tornyok** alkotják a védelem gerincét. Egy mezőt foglalnak el, áthatolhatatlanok, 75 erőforrásba kerülnek és a robotok 5 másodperc alatt építik fel őket. A legfontosabb tulajdonságuk viszont az, hogy másodpercenként lőnek egy, maximum 4 mező távolságra levő ellenségre, ezáltal 3 életerőnyi sebzést okozva neki.
* Néhány jól elhelyezett **fagyasztó** rengeteget javíthat a védelmen. A toronnyal szemben 8 másodpercig tart megépíteni őket és 100 erőforrásba kerülnek, azonban megtérül az áruk, ugyanis messzebbre, 5 mezőnyire lőnek, és az ellenségek élete helyett a sebességüket sebzik, méghozzá egészen addig, amíg az le nem csökken 1-re.

### Ellenségek

A játék három ellenséget tartalmaz, mindhárom egyedi tulajdonságokkal rendelkezik. A spawnerek véletlenszerűen, de egyenlő eséllyel választják ki a következő ellenség típusát. Ha az életük eléri a 0-t, elpusztulnak. Amennyiben elérik a főhadiszállást, szintén elpusztulnak, viszont az addigi hátralevő életüknek megfelelő mennyiségű sebzést okoznak a bázisnak, ezzel közelebb hozva a felhasználót a játék elveszítéséhez.

Az egyes fajták adatai a következők:

* A leggyorsabb ellenség, 15 egységet halad másodpercenként, azonban csak 5 életereje van.
* Minden tekintetben a középső, a sebessége 10, élete pedig 15.
* Ez az ellenség rendelkezik a legtöbb élettel, méghozzá 50-nel. Cserébe a leglassabb, csupán 5 egységnyit tud haladni másodpercenként.

### Munkák

A felhasználó a program használata során a győzelem érdekében különféle feladatokat oszthat ki a robotok számára, melyeket azok legjobb képességük szerint megpróbálnak teljesíteni. Az egyes munkák a kezelőfelület bal alsó sarkában lévő panelén választhatók ki, majd a pálya mezőire kattintva, vagy esetleg több mezőt kijelölve adhatók ki. Egy mezőhöz egyszerre csak egy feladat rendelhető. A munkák a kiadás sorrendjében kerülnek elvégzésre, a még be nem fejezettek vissza is vonhatók. A feladatok három fő típusba sorolhatók be. A különböző típusok különböző jelölésekkel látják el a kiválasztott mezőket:

* A fákat tartalmazó, **kitermelésre** kijelölt mezők az alábbi, szerszámokat tartalmazó jellel vannak ellátva:
* Az épületek (fal, út, torony, fagyasztó) **felépítésére** kijelölt mezők ezt, a tervrajzot ábrázoló jelet hordozzák:
* A **lebontásra** szánt épületek pedig az alábbi módon vannak megjelölve:

# Fejlesztői dokumentáció

## Elemzés

A dolgozat célja egy összetett játékszoftver elkészítése Unity keretrendszerben, C# nyelven. Fontos, hogy a legújabb nyelvi lehetőségéket, programtervezési mintákat felhasználva, az objektum-orientált szemléletet megtartva készüljön el a program. Ezek mellett szükséges még a tiszta, átlátható és karbantartható kódolás, mely nélkül minden projekten hamar eluralkodik a káosz.

A játéknak könnyen megtanulhatónak, a felhasználó számára átláthatónak és nem utolsó sorban szórakoztatónak kell lennie. A rohanó világra való tekintettel ügyelni kell a program egy-egy futásának idejére, hogy az is használhassa, akinek csak 5 szabad perce van, valamint az is, aki több órát is el tud tölteni.

Mivel a felhasználók mind különböző erősségű gépekkel rendelkeznek, lehetővé kell tenni számukra a grafikai beállítások személyre szabhatóságát, hogy a saját ízléseiknek megfelelően állíthassák be a felbontás, a grafikai részletesség és a sima játékmenet közti egyensúlyt.

A szoftver Modell-Nézet-Perzisztencia architektúrában készül el, ezzel elkülönítve az egyes önálló, de egymásra épülő programrészeket, beleértve azok adatait és viselkedésüket.

## Fejlesztői környezet

A szoftver fejlesztése során az alábbi programokat használtam fel:

* Unity 2017.3.0f1
* Visual Studio Enterprise 2017 15.6.6
* Photoshop CC 2015
* GitKraken

## Felhasználói esetek diagramja



## Felhasználói esetek leírása

|  |  |  |
| --- | --- | --- |
| **Felhasználói eset** | **Leírás** | |
| Indítás | Given: | A fájlkezelőben van | |
| When: | Rákattint a futtatható állományra | |
| Then: | Elindul a játék | |
| Kilépés | Given: | Fut a játék | |
| When: | Rákattint a kilépés gombra | |
| Then: | A játék leáll | |
| Új játék | Given: | A főmenüben van | |
| When: | Rákattint valamelyik új játék gombra | |
| Then: | Új játék indul | |
| Szüneteltetés | Given: | Fut a játék | |
| When: | Megnyomja az escape gombot | |
| Then: | A játék szünetel | |
| Folytatás | Given: | A játék szünetel | |
| When: | Megnyomja az escape gombot | |
| Then: | A játék folytatódik | |
| Mentés | Given: | A játék szünetel | |
| When: | Megnyomja a mentés gombot | |
| Then: | A játék elmentődik | |
| Betöltés | Given: | A játék szünetel | |
| When: | Megnyomja a betöltés gombot | |
| Then: | A mentett játékállás betöltődik | |
| Vereség | Given: | A játék fut | |
| When: | A bázis élete nullára csökken | |
| Then: | Elveszti a játékot | |
| Nyerés | Given: | A játék fut | |
| When: | Lejár a nyeréshez szükséges idő | |
| Then: | Megnyeri a játékot | |

|  |  |  |
| --- | --- | --- |
| **Felhasználói eset** | **Leírás** | |
| Kamera irányítás | Given: | A játék fut | |
| When: | Az egér a képernyő széléhez ér | |
| Then: | A kamera mozog | |
| Robot vásárlás | Given: | Van elég erőforrása | |
| When: | Valamelyik robot vásárló gombra kattint | |
| Then: | A megfelelő típusú robot megjelenik | |
| Munka kiadás | Given: | A mezőre kiadható az aktuálisan kijelölt munka | |
| When: | A mezőre kattint | |
| Then: | A munka megjelenik a mezőn | |

## A komponensek diagramja

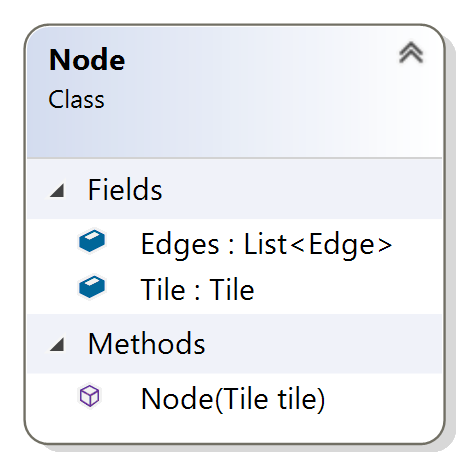


## Modell

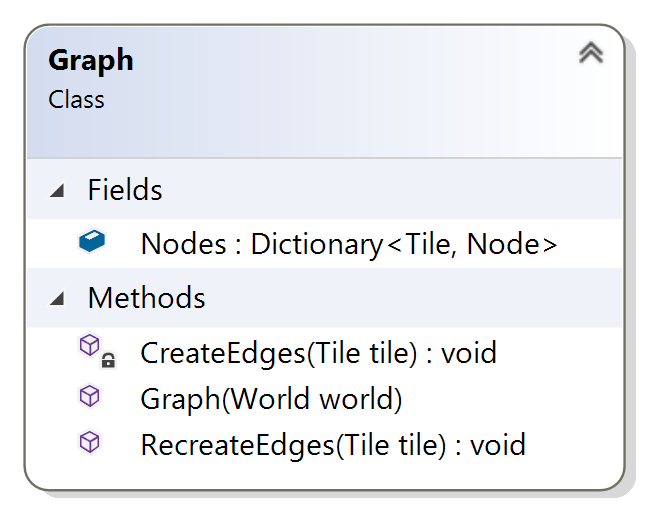
### Edge

* Az **Edge** osztály egy gráfon belüli él reprezentálására szolgál.
* Szerepét az útkereső algoritmusokban tölti be.
* **MovementCost**: lebegőpontos szám típusú mező, mely az él súlyát jelöli.
* **Node**: Referencia arra a csúcsra, amelyikre az él mutat.
* **Konstruktora** paraméterként egy csúcsot és egy valós számot kap, melyek alapján beállítja a megfelelő mezőket.

### Node

* A **Node** osztály egy gráfon belüli csúcsot jelöl.
* Az útkeresés szempontjából fontos.
* A **Tile** mezőben tárolja, hogy melyik, a valós játékban létező mező útkeresési adatait tárolja.
* Az **Edges** field egy lista, amely az összes, a csúcsból kimenő Edge típusú élt tartalmazza.
* **Konstruktora** egy mezőt vár paraméterként, melyet eltárol a Tile adattagjában.

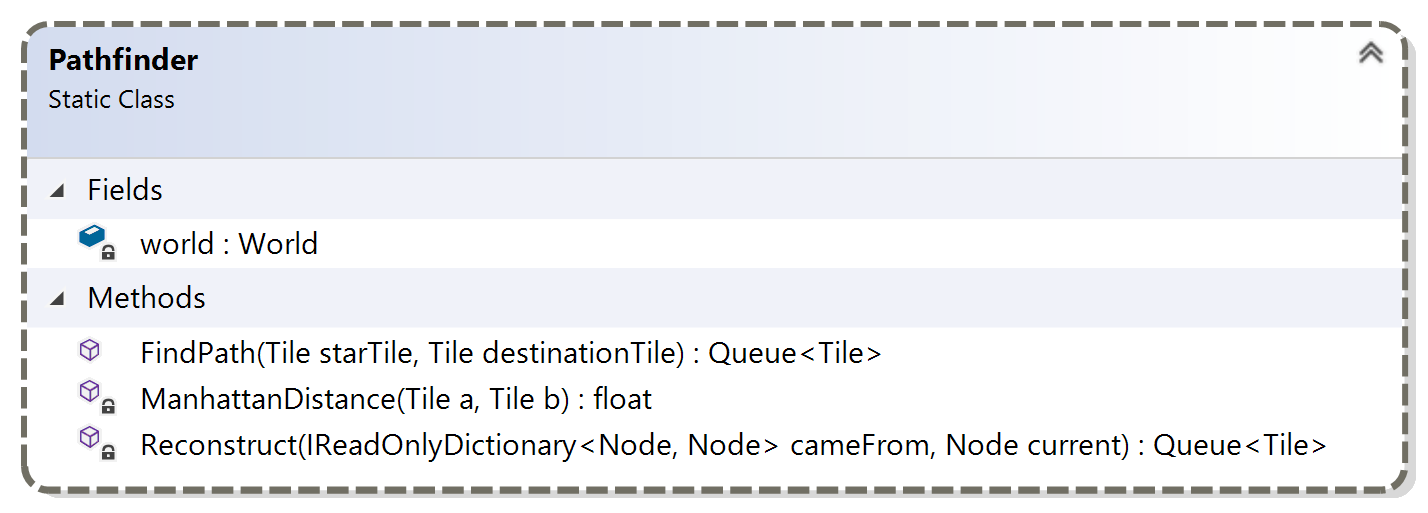
### Graph

* A **Graph** osztály alkotja az útkeresési rendszer alapját.
* A **Nodes** adattag egy Dictionary, mely játékbeli mező kulcsokkal tárol gráfbeli csúcsokat.
* A **CreateEdges** metódus egy mezőt vár paraméterül, mely alapján létrehozza, vagy ha már léteznek, frissíti a mezőhöz tartozó csúcs kimenő éleit. Az élek súlyát a szomszédos mezők MovementCost attribútumából állítja elő, ha pedig az 0, tehát a két mező között ebben az irányban nincs átjárás, az él létre sem jön.
* A **konstruktor** egy World típusú paramétert vár. Ez alapján épül fel a gráf, hiszen ez tartalmazza a játék összes mezőjét. A Dictionary felépítése után az élek is létrejönnek.
* A **RecreateEdges** metódust az élek újra tervezésére lehet használni. A bemenő adatként átadott mező kimenő élei lesznek újra kalkulálva. Akkor hívódik meg, ha a játék valamelyik mezőjében, útkeresési szempontból változás történik. Automatikusan frissíti a szomszédos csúcsok éleit is.

### PriorityQueue<T>

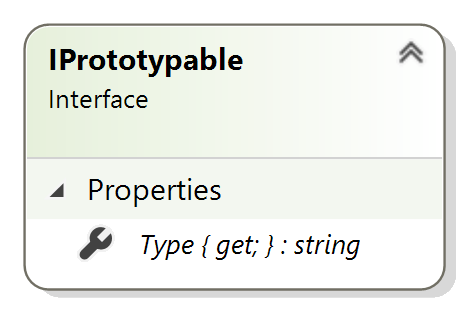
* Mivel a C# nyelvi elemként nem tartalmaz prioritásos sort, szükség volt elkészíteni egy saját implementációt az útkeresés helyes implementálásának érdekében.
* Az osztály egy szabadon elérhető, public domain heap implementációt csomagol be, és használatához szolgáltat egy egyszerű, könnyen használható interfészt.
* Generikus osztály paraméterként megkapja a tárolni kívánt csúcsok típusát
* A **Count** property tárolja az aktuálisan tárolt, adott típusú adattagokat.
* A **Contains** metódussal megtudható, hogy egy, paraméterként adott csúcs a kupac része-e.
* A **Dequeue** függvény visszaadja a legnagyobb prioritású mezőt, majd kiveszi azt a sorból.
* Az **Enqueue** metódus egy paraméterként adott csúcsot illeszt be a kupacba, az ugyancsak bemenő adatként megadott prioritással.
* Az **UpdatePriority** használható egy, már a heapben lévő csúcs prioritásának frissítésére.
* Az **EnqueueOrUpdate** alprogram az előző két függvény valamelyikét hívja meg aszerint, hogy a kupac tartalmazza-e a megadott mezőt.
* A konstruktor felállítja a paraméterként megadott kezdeti nagysággal rendelkező heapet. A hatékonyság érdekében alapértelmezetten egy a nagyság 10.

### Pathfinder



* A **Pathfinder** egy statikus osztály, amely az útvonalkeresést végzi.
* A **world** nevű mezőben mindig az aktuális játékvilágot tárolja, így biztosított, hogy az utak keresését a megfelelő gráfon hajtja végre.
* Az útkeresés az **A\*** algoritmussal működik. Ez a Dijkstra algoritmus egy kibővítése azzal, hogy nem minden irányban keres, hanem egy heurisztikát használ az általános irány eldöntésére. Ezáltal az esetek többségében jelentősen jobb teljesítményt nyújt, mint a heurisztika nélküli Dijkstra. Hátránya azonban, hogy ugyan elég valószínű, nem garantált a legoptimálisabb út megtalálása. Egy olyan valós idejű alkalmazásban, mint például ez a játék, úgy vélem az előnyök nagyban meghaladják a hátrányokat, így erre a bevált és elterjedt algoritmusra esett a választásom.
* Az algoritmus részletes leírása, története és összehasonlítása más megoldásokkal az alábbi linken érhető el: <https://bit.ly/2j7ELut> (utolsó megtekintés: 2018.04.20)
* A **FindPath** metódus egy kezdeti, és egy cél mezőt vár paraméterként. Eredményül egy csúcsokból álló sort ad vissza, mely tartalmazza az útvonalkeresés eredményét. Ha a két csúcs között nincs út, a visszaadott eredmény null lesz. Futása során felhasználja a világ által tárolt útvonalkeresési gráfot.
* A **ManhattanDistance** adja a keresés heurisztikáját. Két csúcs Manhattan távolságát számolja ki, és adja eredményül. Az útvonal keresése így mindig a cél csúcs irányába indul el, és próbál arra tartani.
* A **Reconstruct** metódus állítja elő a keresés végeredményeként kapott sorozatot. A FindPath alprogram hívja meg, átadva neki a kiértékelt csúcsokat, valamint mindhez azt a csúcsot, ahonnan oda elértünk (szülő). Ezek alapján állítja fel a kezdőcsúcsból a célba vezető utat.

### IPrototypeable

* Az **IPrototypeable** interfész adja a prototípus programtervezési minta alapját.
* Lényege, hogy a program futásának elején létrejött objektum prototípusok soha nem változnak, de ha szükséges egy új objektum létrehozása, akkor lemásolhatják az egyik prototípust, amik tárolják a szükséges adatokat, viselkedéseket. Ezáltal elkerülhető a rengeteg leszármazott osztály csapdájába esés, a kód sokkal karbantarthatóbb.
* Minden prototípusokkal rendelkező osztálynak tárolnia kell a szöveges típusát.

### PrototypeManager<T>

* A **PrototypeManager** egy olyat generikus osztály, amely paraméterként átadott T típus prototípusait kezeli.
* A típusparaméternek kötelezően implementálnia kell az **IProtoypeable** interfészt.
* A prototípusok a \_**prototypes** nevű privát adattagban tárolódnak, ami egy stringhez T típusú objektumokat rendelő Dictionary. A kulcsok megegyeznek a prototípus típusokkal.
* Az **Add** metódussal egy új prototípus vehető fel, amennyiben az még nem létezik a dictionary-ben.
* A **Get** alprogram paraméterként egy szöveget kap. Ha tárol ilyen típusú objektumot, akkor visszaad egy arra mutató referenciát. Amennyiben nem talál ilyet, a generikus paraméter alapértelmezett értékét adja vissza (ez általában null).
* A **konstruktor** elvégzi a dictionary kezdeti létrehozását, természetesen üresen.
* Az összes, a játék kezdetén vagy a során létrejöhető objektum ilyen kollekciókban van tárolva.

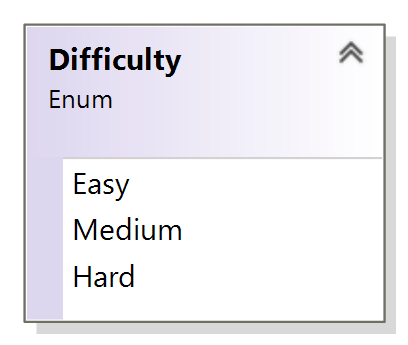
### Prototypes

* A **Prototypes** egy statikus osztály, mely létrehozza, tárolja, és hozzáférést nyújt a játék során létrejövő dinamikus objektumok prototípusaihoz.
* A **Buildings** egy PrototypeManager, ami generikus paraméterként a Building osztályt kapta meg. Tárolja a játék összes épületének prototípusát.
* Az **Enemies** adattag a lehetséges ellenségek kollekciója.
* A **Jobs** mező tárolja a kiadható munkákat.
* A **Projectiles** rejti az egyes védelmi épületek által létrehozott lövedékek prototípusait.
* A **Robots** pedig a két, játékos által megvásárolható és indirekt irányítható robot prototípusát tárolja.
* A **CreateBuildingPrototypes** visszatérési érték nélküli metódus, mely az épület tervrajzok létrehozásáért felel, beleértve azok adatait.
* A **CreateEnemyPrototypes** hozza létre a játékos ellenségeinek őseit.
* A **CreateJobPrototypes** alprogramban jönnek létre a kiadható feladatok, másolásra készen.
* A **CreateProjectilePrototypes** hozza létre a lövedékek alaprajztát, amikből a tornyok és fagyasztók létrehozzák lövéseiket.
* A **CreateRobotPrototypes** hozza létre a robotok prototípusát, tulajdonságaikkal, típusaikkal együtt.
* A **CreatePrototypes** metódus sorban meghívja a fenti 5 függvényt, ezáltal biztosítva a játék objektumainak létrejöttét.
* A statikus **konstruktor** felel a kollekciók inicializálásáért, és a CreatePrototypes függvény meghívásáért. A statikus konstruktorok C#-ban azonnal lefutnak, amint valami hivatkozik az osztályra, így garantált a prototípusok megléte.

### Building

* A **Building** osztály reprezentálja a pályán elhelyezkedő épületeket.
* Implementálja az IDisplayable és IPrototypeable interfészeket
* A **Tile** mezőben tárolják a központi mezőt, amit elfoglalnak. Létrejöttük után ennek értéke már nem változhat.
* Rendelkeznek **X** és **Y** koordinátákkal. Ezek read only propertykként vannak implementálva, az általuk elfoglalt mező megfelelő koordinátáit adják vissza.
* A **Conjoined** boolean változó tárolja, hogy azonos típusú szomszédaival összekapcsolódik-e (ez a falaknál, valamint az utaknál jellemző). Ez a megjelenítés szempontjából fontos.
* A **MovementModifier** mező egy float, befolyásolja az útvonalkereséskor az épület mezőihez rendelt csúcsokba menő élek súlyát, vagy, ha a az épületen nem lehet áthaladni, megakadályozza az élek létrejöttét.
* A **Size**, int típusú mezőben tárolódik az épület kiterjedése. A játékban van példa mind 1x1-es, mind 3x3-mas épületekre.
* A **Type** adattag tárolja az épület típusát. A Prototípusok lekérésénél, valamint több egyéb helyen van használva.
* A **timer** field tárolja a következő frissítésig hátralevő időt másodpercekben. Amennyiben 0, vagy az alá csökken, az Update metódus meghívja az OnUpdate alprogramot.
* Egyes épületek rendelkezhetnek viselkedéssel is. Amennyiben rendelkeznek, az az **OnUpdate** változóban van tárolva. A típusa Action<Building>. Az C#-ban az akciók hasonlítanak a c++-ból ismert függvény pointerekre, azonban nagy előnyük, hogy típushelyesek, ezáltal használatuk sokkal kézenfekvőbb és biztonságosabb.
* Az **UpdateInterval** tárolja, hogy egy-egy esetleges frissítés után mennyire kell visszaállítani a timer mezőt. Gyakorlatilag a frissítések között eltelt időt reprezentálja. Úgynevezett nullable adattag, tehát float típusa létére mégsem kötelező, hogy legyen értéke.
* Az **Update** metódus az utolsó hívása óta eltelt időt kapja paraméterül másodpercekben. Ennyivel csökkenti a timer mezőt, majd, ha szükséges, invoke-olja az OnUpdate akciót. Amennyiben az akció, vagy az UpdateInterval mező értéke null, nem csinál semmit.
* Amennyiben változás történik az épületen, meghívódik az **OnChange** metódus. Ez megvizsgálja, hogy a **Changed** eventjére feliratkozott-e egy nézet. Amennyiben igen, kiváltódik az esemény, és a nézet, aminek célpontja ez az épület, értesül a változásról és végrehajtja a szükséges módosításokat.
* Két **konstruktorral** is rendelkezik. Az egyik a prototípusok létrehozására használt, paraméterként megkapja az összes szükséges adatot. Néhány paraméter alapértelmezett értékkel is rendelkezik az olvashatóság javítása érdekében. A másik a prototípusok klónozására használható, paraméterként ugyanis egy másik épületet vár, aminek adattagjait lemásolva hoz létre egy új épületet, ezzel elérve a prototípus programtervezési minta célját.

### Difficulty

* A **Difficulty** egy enumerációs osztály, mely a játék nehézségi szintjeit reprezentálja.
* Jelenleg három nehézségi mód van, az egyes szintek közötti részletes eltérések a felhasználói dokumentációban olvashatók.
* Röviden, ahogy nő a nehézség, annál kevesebb erőforrás áll a játékos rendelkezésére, és annál több ellenség támadja, több irányból.

### Enemy

* Az **Enemy** az ellenségek osztálya.
* Mind az IPrototypeable, mind az IDisplayable interfészt implementálja, hiszen prototípusok másolásával jönnek létre, valamint megjeleníthetők a nézet által.
* A **Path** mezőben tárolják az általuk bejárni kívánt utat, az aktuális mezőjüktől egészen a főhadiszállásig.
* A **Destination** egy Tile típusú mező, méghozzá az, amelyiket a bázis foglal el.
* Élettel is rendelkeznek, az aktuális érték a **Health**, egész típusú adattagban tárolódik. Amennyiben ez nullára csökken, az ellenség elpusztul.
* A maximum, kezdeti életerejük a **MaxHealth** mezőben van, a prototípus klónozásakor erre állítódik be az aktuális életük.
* A **MovementProgress** egy float típusú változó, azt reprezentálja, hogy az ellenség hol jár a következő mezőre való lépésben. A nézet ezt használja fel a pontos megjelenítéshez.
* A **NextTile** mező tárolja az enemy útján a következő mezőt. Ide fog lépni, amennyiben a MovementProgress eléri az 1-et.
* A **Speed** mutatja meg, hogy mekkora sebességgel halad az ellenség. A játékos által ez csökkenthető, azonban soha nem csökkenhet 1 alá.
* A **Tile** változó tárolja az aktuálisan elfoglalt mezőt.
* Az ellenség saját **típusa** meghatározásához egy string típusú referenciát tárol.
* Mivel implementálja az IDisplayable interfészt, lekérdezhető aktuális **X**, valamint **Y** koordinátája.
* A **Clone** metódus egy már meglévő ellenséget vár paraméterül, és a másoló konstruktort felhasználva visszaad egy új objektumot, ami megegyezik a paraméterben kapottal. A prototípusok lemásolására alkalmas.
* Ugyancsak két **konstruktorral** rendelkezik. Az egyikkel létrehozhatók, a másikkal lemásolhatok a prototípusok.
* A **FindPathToHeadQuarter** nevéből adódóan megkeresi az útvonalat az aktuális mezőtől a főhadiszállásig. Beállítja a Destination változót a bázis aktuális mezőjére. Visszatérési értékként megadja, hogy létezik-e út. Amennyiben nem, a felhasználó lezárt minden útvonalat, és a játék emiatt véget ér.
* Az **OnChange** alprogram az ellenséggel történt változások esetén hívódik meg, eseményt váltva ki, jelezve ezzel a hozzá rendelt nézetnek, hogy frissítenie kell a megjelenítést.
* Az **Update** függvény felel a viselkedésért. Amennyiben az ellenség élete eléri a nullát, megsemmisíti azt. Ha még nincs útvonala, vagy időközben az járhatatlanná vált, újat keres. Különben végrehajtja a sebességnek megfelelő mozgást, és ha elérte célját, csökkenti a bázis életerejét.

### Job

* A **Job** a kiosztható munkák osztálya.
* Rendelkezik egy Func<Tile,bool> típusú mezővel, amelynek a neve **CanCreate**. A Func egy visszatérési értékkel rendelkező Delegate, ami az Action-nél már említett típushelyes függvény pointer. Egy mezőt kapva eldönti, hogy arra ki lehet-e adni az aktuális feladatot.
* A munka az **AmountDone** valós értékű mezőben tárolja, hogy hol tart az elvégzése.
* Árral is rendelkeznek, ez a kiadásukkor kerül levonásra a játékos erőforrásaiból és a **Cost** mezőben vannak tárolva.
* A **IsComplete** egy read only property, azt határozza meg az AmountDone és a TimeToComplete változókból, hogy a munka el van-e már végezve.
* Az **OnComplete** egy Action, akkor hívódik meg, ha a feladat elkészül.
* A **Progress** mező által lekérdezhető a munka százalékos végzettségi szintje, ez a megjelenítés számára hasznos.
* A **Robot** mezőben tárolva van, ha egy robot elvállalta és dolgozik az aktuális munkán.
* A **RobotType** egy szöveges mező, megmutatja, hogy milyen robottípusok alkalmasak a feladat elvégzésére.
* A **Tile** az a mező, amelyre a munka ki lett osztva.
* A hatékonyság és gördülékenység érdekében, ha egy robot lemond egy munkáról, akkor az egy rövid időre nem vállalható el senki más által. Ezzel elkerülhető az elérhetetlen feladatok által okozott lassulás.
* A **TimeToComplete** egy float típusú változó, azt tárolja, hogy hány másodpercig tart elvégezni egy munkát.
* A feladatok típussal is rendelkeznek, mivel implementálják az IPrototypable interfészt.
* **X** és **Y** koordinátájuk lekérhető, ez megegyezik a mező koordinátáival, amin elhelyezkednek.
* Létezik copy **konstruktor**, amivel a másik konstruktorral létrehozott prototípusok klónozhatók.
* Az **OnChange** változás esetén meghívja a **Changed** eventre feliratkozott metódusokat.
* A **Work** metódust a robotok hívják meg, amikor dolgoznak a feladaton. Növelik az AmountDone változót, valamint, ha a Progress eléri az 1-et, meghívódik az OnComplete függvény, ami elvégzi a befejezéskor esedékes műveleteket.

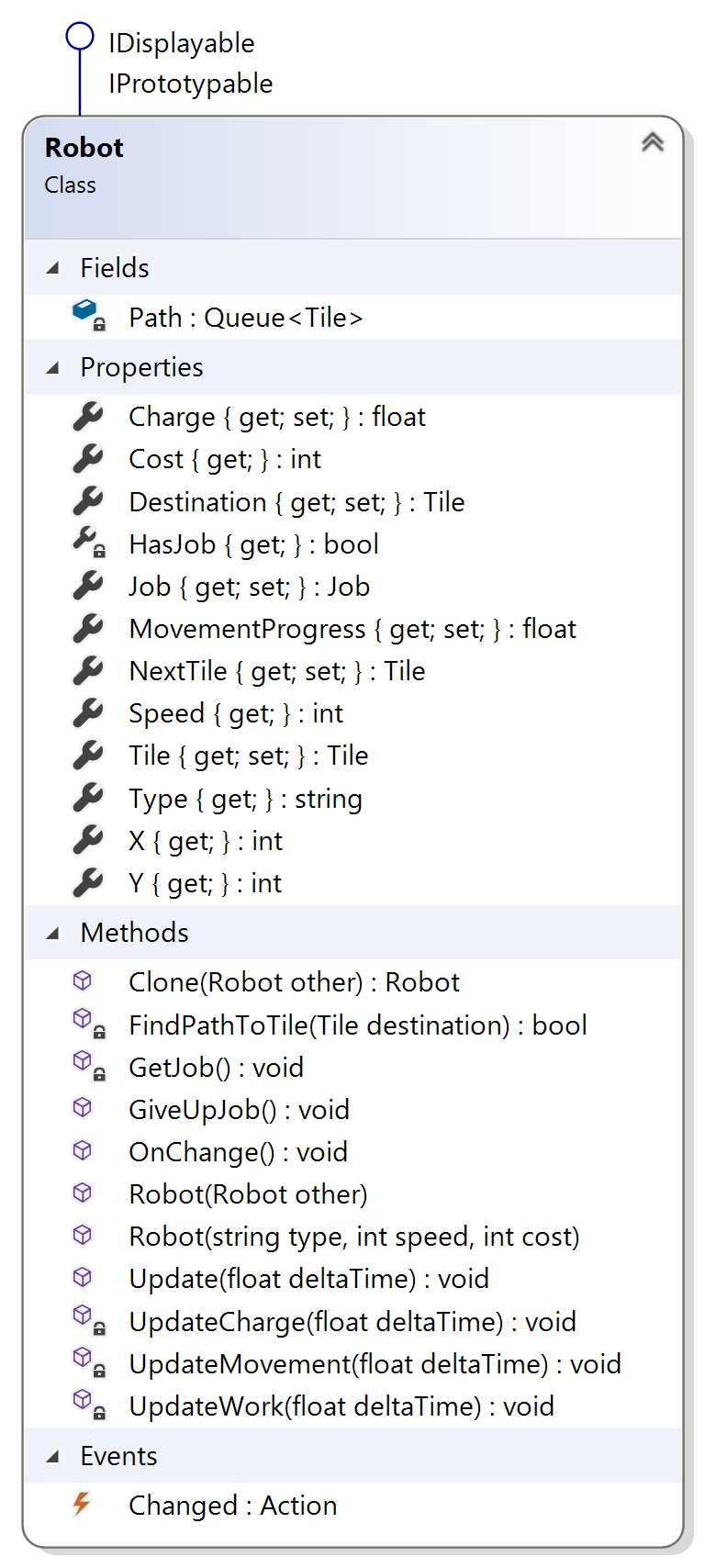
### JobManager

* A **JobManager** osztály felel a feladatok kiosztásáért, a kiosztott és a kiosztásra váró munkák számontartásáért, újak felvételéért, régiek visszavonásáért, valamint az időtúllépések nyilvántartásáért.
* A beágyazott **TimeOut** osztály reprezentál egy időtúllépést. Számon tartja a hátralévő időt, a feladatot, valamint a hozzá tartozó robotot. Konstruktora értelemszerűen az adattagokhoz rendeli a paraméterül kapott értékeket.
* Ezen a **timeout**-ok egy Listában vannak elhelyezve, a timeOuts mezőben.
* **Konstansként** tároljuk az időtúllépések lejártának idejét.
* Tárolunk az aktuális **világra** egy referenciát, a könnyű elérés érdekében.
* Az **AvailableJobs** listában vannak azon feladatok, amelyek elérhetők és még egy robot sem vállalta el őket.
* A **TakenJobs** is List<Job> típusú, azon munkákat tartalmazza, amelyek végrehajtásán egy robot aktívan dolgozik.
* A **Jobs** egy csak olvasható property, konkatenálva visszaadja az AvailableJobs és a TakenJobs listákat, ezáltal lekérdezhető az összes munka, egyetlen, felsorolható típusként.
* A **CancelJob** egy feladatot vár paraméterként, amit visszavon és megsemmisít. Elvállalhatatlanná teszi az összes robot számára, visszaadja az elköltött erőforrásokat, feladatja a munkát azzal a robottal, aki már elvállalta (ha van ilyen), majd törli a listákból. Végül értesíti a nézetet a változásról.
* A **CreateJob** egy új munka létrehozására szolgál. Ellenőrzi a feladat CanCreate metódusával, hogy létrehozható-e egyáltalán, és ha igen, a paraméterben kapott prototípus munkát leklónozza a szintén bemenő adatként kapott mezőre, az AmountDone változót pedig nullára állítja. Le is vonja a szükséges erőforrásokat a játékostól.
* A **TakeJob** alprogramot a robotok hívják meg, amikor egy új munkát szeretnének elvállalni. Visszaadják az elvállalható feladat objektumot, vagy nullt, amennyiben nincs a robot típusának megfelelő munka. A kiosztott feladatot átteszi a TakenJobs listába az AvailabeJobs listából.
* Az **Update** metódus felel az időtúllépések idejének csökkentéséért, valamint az összes, már elvégzett munka kivételéért a listákból.

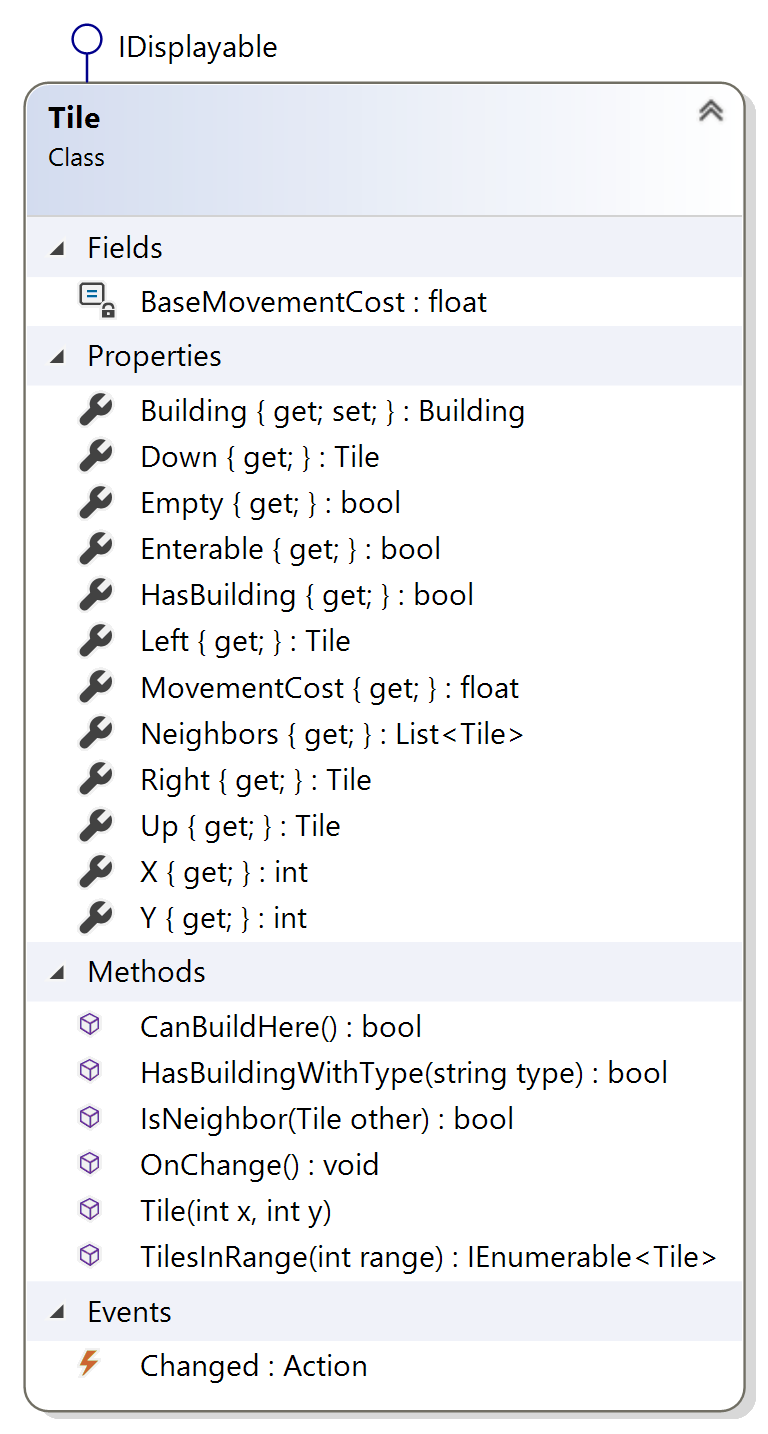
### Projectile

* A **Projectile** osztály valósítja meg az egyes védelmek által tüzelt lövedékek típusát.
* Az implementált interfészeiből kiderül, hogy megjeleníthető és lehetnek prototípusai.
* Az **OnHit** akció hívódik meg, amikor ütközik a célpontként kijelölt ellenséggel. A lövedéktől függően ez más-más módon befolyásolhatja az ellenfeleket.
* A **Position** mezőben tárolódik a lövedék aktuális pozíciója. Ez Vector2 típusú, ami a Unity motor egy beépített osztálya. X és Y koordinátákkal rendelkezik és vannak hozzá különböző kiegészítő metódusok.
* A **Speed** reprezentálja a lövedék sebességét, amivel az ellenség felé halad. Inicializálás után nem változik.
* A **Target** egy ellenségre mutató referencia, a projectile célpontját jelöli.
* Szöveges típussal is rendelkeznek, melyet a **Type** mező tárol.
* **X** és **Y** koordinátái lekérdezhetők, ezt a Position vektorból nyerik ki a read only propertyk.
* Az **OnChanged** metódus hívódik meg, ha valamilyen olyan változás történik, amelyről érdemes értesíteni a nézetet. Jelen esetben ezen változások közé tartozik a helyzet megváltozása, a lövedék létrejötte, valamint a cél elérése. Kiváltja a Changed eseményt.
* Már megszokott módon rendelkezik egy copy, valamint egy sima, adattagokkal paraméterezett **konstruktorral**.
* A **SetPosition** két, egész típusú koordinátát vár, amikből létrehoz egy Vector2 típusú változót, majd erre állítja a Position értékét. Az új lövedékek létrehozásakor fontos, hiszen a mezők koordinátái egészként vannak tárolva.
* Az **Update** alprogram kezeli a frissítéseket. Ha a célpont időközben megsemmisült, elpusztítja a lövedéket is. Ha nem, megvizsgálja a célpont és a lövedék közötti távolságot, és amennyiben elég közel került, regisztrálja a találatot. Egyéb esetben közelebb mozgatja az objektumot a kijelölt ellenség felé, és értesíti a nézetet a változásokról.

### Robot

* A **Robot** a játékban fontos szerepet betöltő robotok típusa.
* Lehetnek prototípusaik, valamint a nézet is kapcsolódhat hozzájuk, mert implementálják a megfelelő interfészeket.
* A **Path** változó egy mezőkből álló sorozat, mely mindig az aktuális tile és a cél közötti utat tárolja.
* Rendelkeznek **Charge** adattaggal is. Ez valós típusú, és az aktuális töltődési szintet tárolja.
* A **Cost** egész típusú, azt mondja meg, hogy mennyi erőforrásba kerül egy-egy adott típusú robot megvásárlása.
* A **Destination** egy Tile típusú adattag. Itt tárolódik a cél, ahova a robot el akar jutni. Állapotától függően ez általában vagy egy munka helyszíne, vagy pedig a főhadiszállás, ahova visszatérnek töltődni.
* A **HasJob** egy read only property, megadja, hogy jelenleg rendelkezik-e a robot feladattal. Ehhez az alábbi mezőt vizsgálja.
* A **Job** változóban tárolódik az aktuálisan elvállalt feladat, amit a robot próbál végrehajtani.
* A **MovementProgress**, csak úgy, mint az ellenségeknél, azt reprezentálja, hogy hol jár a sorban a következő mezőre lépés folyamatában. Amennyiben ez eléri az 1-et, megtörténik a mező váltás.
* A **NextTile** mindig a következő tile a útvonalon. Ha ez időközben elérhetetlenné válik, az útvonal újra kiértékelődik.
* A **Speed** mutatja meg, hogy milyen gyorsan képes a robot haladni az útja során.
* A **Tile** mező tárolja az entitás aktuálisan elfoglalt mezőjét.
* **Típussal** is rendelkezik, egy stringként van tárolva, ezáltal eldönthető, hogy milyen típusú feladatokat tud elvégezni, valamint a klónozásnál is nagy szerepet tölt be.
* **Koordinátái** megegyeznek az aktuális mezője koordinátáival, ezekre a propertykre a nézet interfészének implementálásához van szükség.
* A **Clone** metódusnak egy másik robot (általában egy prototípus) megadásával lehetőség van egy új robotot létrehozni.
* A **FindPathToTile** függvény a Pathfinder statikus osztályt felhasználva próbál útvonalat keresni az entitás aktuális pozíciója és a paraméterül megadott mező között. Visszatérési értékként megadja, hogy létezik-e út. Amennyiben igen, beállítja a Path, a Destination és a NextTile változókat, valamint lenullázza az MovementProgress értékét.
* A **GetJob** a JobManager osztálytól próbál kérni egy számára elvégezhető feladatot.
* A **GiveUpJob** akkor hajtódik végre, ha a robot nem tudja elvégezni a számára aktuálisan kiosztott feladatot. Ez általában akkor történik meg, amikor nincs elérhető útvonal a kettő között.
* Az **IDisplayable** interfészt implementálva változások alkalmával az OnChange alprogram hívódik meg, ami értesíti a nézetet a módosításról.
* Van copy, valamint adattagokkal paraméterezett **konstruktor** is, mint minden más, a játéban előforduló osztálynál.
* Az **Update** metódus a tagoltság növelése érdekében jött létre, sorban meghívja az alábbi három alprogramot, átadva az utolsó hívás óta eltelt időt, majd értesíti a nézetet, hogy az elvégezze a szükséges frissítéseket.
* Az **UpdateCharge** felel a töltődési szint szimulálásáért. Folyamatosan csökkenti azt, majd, ha elért egy bizonyos szintet, kényszeríti a robotot a töltődési állapotba való átállásra. Ilyenkor az entitás feladja az aktuális munkáját, és új feladatot kap: töltődjön fel. Ezt mindig a bázis mellett tudja végrehajtani. A feltöltés után visszatér a normál állapotába, és megpróbál elvállalni egy új feladatot. Ha a töltődési szint valamilyen oknál fogva lecsökken nullára, a robot megsemmisül.
* Az **UpdateMovement** metódus végzi a robot mozgatását. Mindig az aktuális cél felé tereli a saját sebességével, amennyiben az elérhető. Ha nem, megpróbál egy másik útvonalat keresni. Ha ez sem sikerül, feladja célját, hogy esetleg más robot is megpróbálhassa azt. Amennyiben a MovementProgress értéke 1 fölé nő, végrehajtja az átmenetet a mezők között, és kiveszi a következő mezőt a Path változóból.
* Amennyiben a robot a célja mellé ért, elkezdhet dolgozni a feladatán. Ezt az **UpdateWork** alprogram hajtja végre, a robot meghívja a Job Work metódusát, ezzel szimulálva a munkát. Ha végzett, új feladatot keres.

### Tile

* A **Tile** osztály valósítja meg a mezők típusát.
* Alkalmas a nézet által megjelenítésre, mert implementálja az IDisplayable interfészt.
* Rendelkezik egy konstans változóval, ami meghatározza a mezőkön való áthaladás alapárát, útvonalkeresés szempontjából. Ez a **BaseMovementCost**.
* A **Building** adattagban tartjuk az esetleges épületre való referenciát, amely ezt a mezőt foglalja el.
* Az **Up** read only property megadja a mező felső szomszédját.
* A **Down** read only property megadja a mező alsó szomszédját.
* A **Left** read only property megadja a mező bal oldali szomszédját.
* A **Right** read only property megadja a mező jobb oldali szomszédját.
* Az **Empty** propertyvel lekérdezhető, hogy a mező üres-e. Egy mező üres, ha nincs rajta épület, és nincs rá feladat kiosztva.
* Az **Enterable** adattag által kideríthető, hogy útvonalkeresés során be lehet-e lépni a mezőre. Ehhez az kell, hogy az áthaladás ára ne legyen nulla.
* A **HasBuilding** property megadja, hogy van-e épület a mezőn.
* A **MovementCost** egy számolt property, megadja, hogy mennyi a mezőn való áthaladás nehézsége. Ez az alap, BaseMovementCost-ból és az esetleges, mezőn való épülettől függ.
* A **Neighbors** egy olyan property, amellyel lekérdezhetők a mező szomszédai, a pálya szélének figyelembe vételével. A szomszédok egy listában térnek vissza.
* Rendelkeznek **X** és **Y** koordinátákkal, a rajtuk elhelyezkedő objektumok ezt veszik alapul.
* A **CanBuildHere** metódus elárulja, hogy lehetséges-e a mezőre építés. Akkor lehetséges, ha még nincs rajta épület.
* A **HasBuildingWithType** alprogrammal megtudható, hogy létezik-e adott típusú épület a mezőn. Egy boolean értékkel tér vissza.
* Az **IsNeighbor** függvény paraméterként egy másik mezőt vár, és megadja, hogy a kettő szomszédos-e.
* Az esetleges változásokról a nézetet a **Changed** event kiváltásával értesíti, amit az **OnChange** metódus hív meg.
* A **TilesInRange** függvény egy egész számot vár paraméterként, és egy mezőket tartalmazó felsorolható típust ad vissza, ami tartalmazza az adott hatókörön belüli más mezőket.

### World

* A **World** a szoftver központi osztálya. Ez kapcsolja össze a modell összes többi elemét.
* Kapcsolódik hozzá egy nézet, amit az **OnChange** meghívásával tud értesíteni a változásokról a **Changed** eseményen keresztül, mert implementálja az IDisplayable interfészt.
* A **StartingTime** nevű konstans adattag tárolja az alapértelmezetten túlélni szükséges időt, amit az egyes nehézségi szintek tovább befolyásolhatnak.
* A privát, \_**size** field a pálya méretét reprezentálja, vagyis azt, hogy hányszor hány mező van jelen a játékban. Alapértelmezetten ez 50x50-es pályákat jelent.
* A **random** egy System névtéren belüli Random típusú változó, amely véletlen szám generálásra használható.
* A World osztály megvalósítja az egyke programtervezési mintát, hiszen a **Current** property-ben mindig tárolja az aktuális példányra mutató referenciát, amin keresztül az összes többi osztály is hivatkozik rá. Ha a konstruktor újra hívódik, például új játék kezdetekor vagy egy korábbi állás betöltésekor, a Current változó értéke is megváltozik, ezáltal a régi világra semmi nem fog referenciát tárolni, így a C# szemétgyűjtő algoritmusa ki tudja törölni azt a memóriából. Emiatt biztosított, hogy a World objektumból mindig csak egy lesz példányosítva.
* A **Buildings** egy lista, amely az aktuális épületeket tárolja.
* Az **Enemies** az ellenségeket tartalmazó lista.
* A **Graph** adattag a korábban ismertetett gráf típusra tart egy referenciát, ami az útvonalkereséshez szükséges adatokat tartalmazza. A konstruktor hívásakor mindig felépül, később csak a változtatásoknak megfelelően módosul.
* A főhadiszállásra külön, Building típusú referenciát tárol a **HeadQuarters** nevű adattagban.
* A bázishoz élet is tartozik, ez a **Health**, egész típusú mezőben van számon tartva. Ha nullára csökken, a játék véget ér.
* A **JobManager** a korábban leírt munkakezelő osztály, mely a munkákkal kapcsolatos feladatokat látja el.
* Amennyiben nem létezik útvonal a főhadiszálláshoz, az ellenségek ezt a **NoPathToHeadQuarter** logikai változón keresztül tudják jelezni. Ilyenkor a játék szintén véget ér.
* A szüneteltetésre is van lehetőség, ennek ténye a **Paused** adattagban van tárolva. Ha ez igaz, nem telik az idő, nem mozog semmi, nem futnak a frissítések.
* A **Projectiles** egy egyszerű lista, ami a jelenleg repülő lövedékeket tartja számon.
* A **RemainingTime** reprezentálja a pálya megnyeréséig hátralevő időt, másodpercekben.
* A **Resources** mező tárolja a jelenleg a játékos rendelkezésére álló erőforrásokat.
* A pályán dolgozó vagy töltődő összes robot a **Robots** elnevezésű listában van elhelyezve.
* Az egyes mezők koordináta szerinti lekérdezésére lett létrehozva egy **indexer**, mellyel a világra mutató referenciától egyszerűen lehet mezőket visszakapni, ugyanolyan szintaxissal, mintha egy 2 dimenziós tömb elemét szeretnénk megtudni.
* A **Tiles** adattag tárolja a referenciákat a pálya mezőire.
* A **CreateBuilding** metódus paraméterül egy épület prototípusát és az építési helyként kijelölt mezőt kapja. Amennyiben a tile már be van építve, nem tesz semmit. Egyébként létrehozza a mérettől függően a beépítésre szánt mezők listáját, és azoknak is ellenőrzi ürességét. Ezek után klónozza a prototípust, és a visszakapott, új épületet elhelyezi a térképen, ügyelve arra, hogy az kapcsolódik-e szomszédaihoz. A változott mezőkhöz tartozó, gráfon belüli csúcsokból induló és azokba vezető éleit is frissíti az új adatok alapján. Ezek után értesíti mind a saját, mind az új épület nézetét a változásokról.
* A **CreateEnemy** függvény egy új ellenség elhelyezésére használható. Kap egy prototípust, amit klónoz, és egy mezőt. Ezek ismeretében elvégzi a szükséges műveleteket, majd értesíti a nézetet.
* A **CreateProjectile** hasonlóan működik a lövedékekre, azonban az előbbieken felül paraméterként megkapja a célpont ellenséget is, amit be is állít a megfelelő pozícióval együtt.
* A **CreateRobot** a fentiekhez hasonló művelet a robotok vásárlására, azzal a kiegészítéssel, hogy levonja az entitás árát a felhasználó erőforrásaiból.
* A **DemolishBuilding** egy épület lebontására alkalmazható. Bemenő adatként egy mezőt kap, amin a lebontandó épület elhelyezkedik. Felállít egy listát a building által elfoglalt mezőkről, majd végrehajtja a lebontást, valamint újra kiértékeli az érintett éleket az útvonalkeresési gráfban. A nézet értesítése sem marad el.
* A **DestroyEnemy** metódussal lehet valamilyen oknál fogva ellenségeket elpusztítani, legyen az akár a védelmek miatt, vagy mert az enemy sikeresen elérte a főhadiszállást.
* A **DestroyProjectile** a paraméterül kapott lövedék eltüntetésére használatos, ha például az célba ért, vagy a kiszemelt ellensége megszűnt létezni.
* A **GenerateEnvironment** alprogram állítja fel a pályát a mérettől és a nehézségtől függően. Kiszámolja az elhelyezendő fák számát és véletlenszerűen, a random adattag segítségével szétszórja őket a mezőkön. Ugyancsak itt helyeződik el az 1, 2 vagy 4, ellenségeket gyártó spawner a pálya sarkain.
* Az **OnChange** metódus a megszokott módon akkor hívódik, ha a nézetet érintő változás történik az objektumban. Ekkor a **Changed** event kiváltódik, így értesül a megjelenítő és el tudja végezni a szükséges frissítéseket.
* Az **Update** alprogram hajtja meg a játékot. Amennyiben a program szüneteltetve van, nem tesz semmit. Egyébként meghívja a lent ismertetett, robotokat, ellenségeket, lövedékeket és épületeket meghajtó függvényeket, csökkenti a nyerésig hátralevő időt, valamint frissíti a JobManagert is.
* Az **UpdateRobots**, **UpdateEnemies**, **UpdateBuildings** és az **UpdateProjectiles** metódusok végzik sorban a robotok, az ellenségek, az épületek, valamint a lövedékek előrevitelét a megadott idővel. Mindegyik egy LINQ ciklussal halad végig a megfelelő kollekciókon, és meghívja az egyes objektumok Update metódusát.
* A **konstruktor** a paramétereknek megfelelően létrehozza a mezőket, inicializálja az adattagokat, beállítja a nehézségnek megfelelő hátralevő időt és a rendelkezésre álló erőforrásokat, valamint meghívja a pálya generálására alkalmas metódust.

## Perzisztencia

A játékok elmentése és betöltése XML formátumban történik, a Unity motor által szolgáltatott PlayerPrefs könyvtár felhasználásával. Ez lehetővé teszi az összes, Unity által támogatott platformon az adatok tárolását és megőrzését a játék szessziók között.

Windows platformon ez a registryben történik, de Linuxon például a ~/.config/unity3d/

könyvtárban. A játék egyszeri menetének hossza nem igényli, hogy egynél több mentési hely álljon rendelkezésre, úgyhogy az új mentések mindig felülírják az előzőt, betöltéskor pedig mindig a legutolsó állás töltődik be.

Ezen utasításokat a felhasználó a menüben adhatja ki, melyről a nézet értesíti a modellt, ami pedig végrehajtja a szükséges műveleteket.

A World osztály implementálja az IXmlSerializable interfészt, így a megírt implementáció alapján XML szöveges formátummá alakítható.

A WriteXml metódus végzi az átalakítást. Menti a méretet, a hátralevő időt, a rendelkezésre álló erőforrásokat, valamint a főhadiszállás életerejét. Ezután jönnek a játék objektumai, sorban az épületek, robotok, ellenségek és robotok. Itt alkalmazva van a pehelysúlyú programtervezési minta, ugyanis nincs minden attribútumuk tárolva, csupán a típusuk és azok, melyek a futás során változhatnak, például a pozíciójuk, robotoknál a töltődési állapotuk, feladatoknál a készenléti szintjük stb. Azon változók, melyek a létrejöttükkor beállítódnak, viszont az élettartamuk során nem változnak, a prototípusokból vannak kinyerve töltődés során, ezzel nagyban csökkentve a mentett adatok méretét.

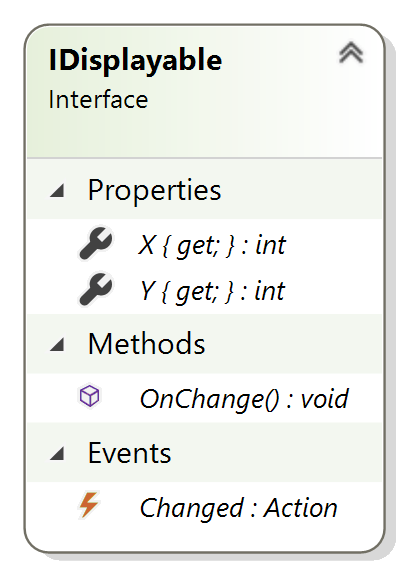
A **ReadXml** végzi az Xml szövegek beolvasását. Alaphelyzetbe állítja a világot, majd betölti annak attribútumait. Ez után következnek az egyes objektum szekciók, melyekhez meghívja a **ReadXmlBuildings**, **ReadXmlJobs**, **ReadXmlEnemies** és **ReaXmlRobots** alprogramokat, melyek elvégzik a hozzájuk rendelt entitások beolvasását és a prototípusokból való példányosítását.

A betöltés után a játék folytatható a korábbi állapotából.

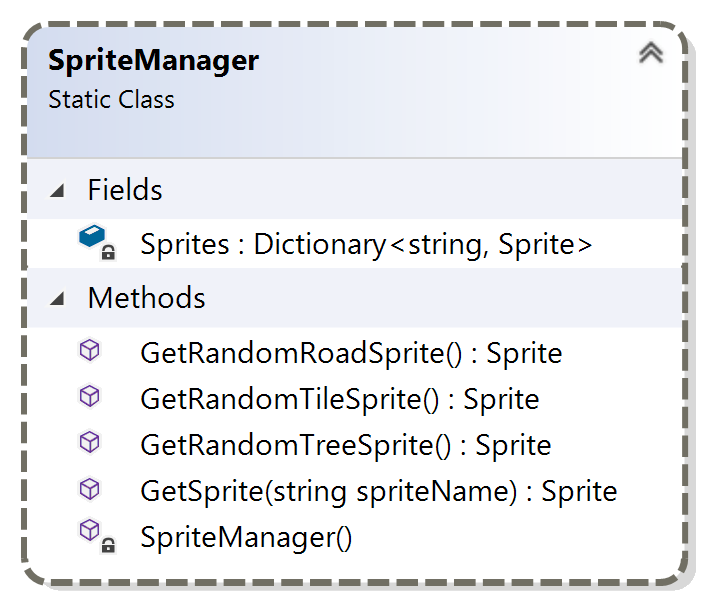
## Nézet

A nézet felel a felhasználóval való interakciók lebonyolításáért, valamint a játék egészének megjelenítéséért. Az egyes nézetek Unity prefabként vannak tárolva, melyek GameObject-ként hozhatók létre a játékban. Minden GameObject-hez van kötve egy nézet osztály, ami biztosítja annak vezérlését és a célpontjának számon tartását.

### IDisplayable

* Az **IDisplayable** interfészt minden olyan osztály implementálja, amit vizuálisan meg lehet jeleníteni a felhasználó számára.
* Megköveteli az **X** és **Y** koordináták lekérdezhetőségét.
* Szükséges rendelkezniük egy **OnChange** metódussal, valamint egy **Changed** eventtel.

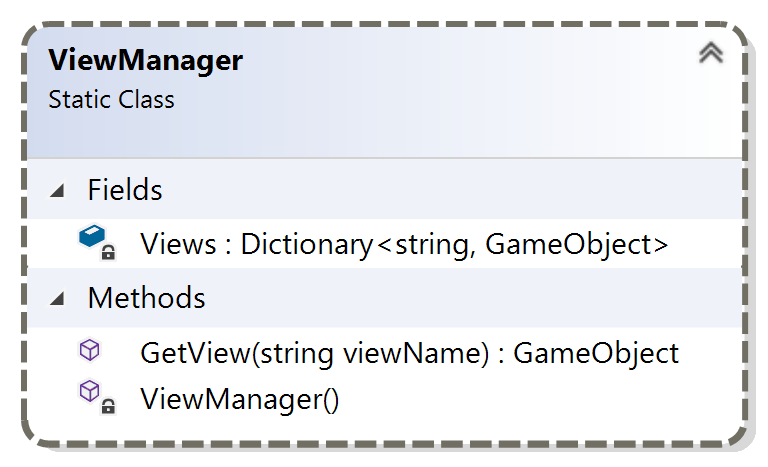
### SpriteManager

* A **SpriteManager** statikus osztály, mely a sprite-ok betöltéséért felel, valamint biztosítja az azokhoz való hozzáférést.
* A **Sprites** egy olyan Dictionary, amelyből a sprite-ok nevével lehet lekérdezni azokat.
* Az előbbi lekérdezést a **GetSprite** metódus végzi el, paraméterként egy nevet vár, visszatérési értéke pedig egy Sprite típusú objektum, amennyiben az létezik.
* Néhány játékelem többféle kinézettel is rendelkezik a játékélmény fokozásáért. Ezek közül véletlenszerűen kapnak egyet amikor létrejönnek. Ilyen elemek a fák, az utak és a mezők. A random sprite lekérdezésére rendre a **GetRandomTreeSprite**, a **GetRandomRoadSprite**, valamint a **GetRandomTileSprite** ad lehetőséget.

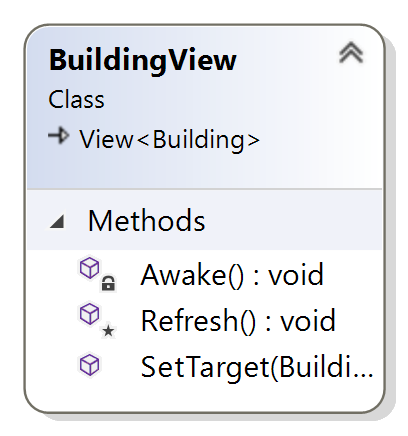
### View<T>

* A **View** egy generikus, absztrakt osztály, ami a MonoBehaviour ősosztályból származik, így leszármazottai példányosíthatók Unity GameObjectként.
* Minden nézet osztálynak ez az őse.
* A **T** generikus paraméternek kötelezően implementálnia kell az IDisplayable interfészt.
* Tárol egy **Spriterenderer** típusú referenciát. Ez a Unity motor által szolgáltatott, sprite-ok megjelenítésére használható osztály.
* A megjelenítésre szánt T típusú játékelemet a **Target** nevű változóban tartja.
* Az **Awake** függvény az objektum létrejötte után fut le, beállítja a megfelelő SpriteRenderert, valamint, hogy a nézet melyik megjelenítési rétegen látszódjon.
* A **Refresh** egy absztrakt metódus, a leszármazottak itt implementálják a nézetek frissítését a célpontok változásai alapján.
* A **SetTarget** állítja be a megjeleníteni kívánt objektumot célpontként.
* Az **UpdatePosition** függvény felel a játékelem helyes pozíciójának megjelenítéséért a pályán.

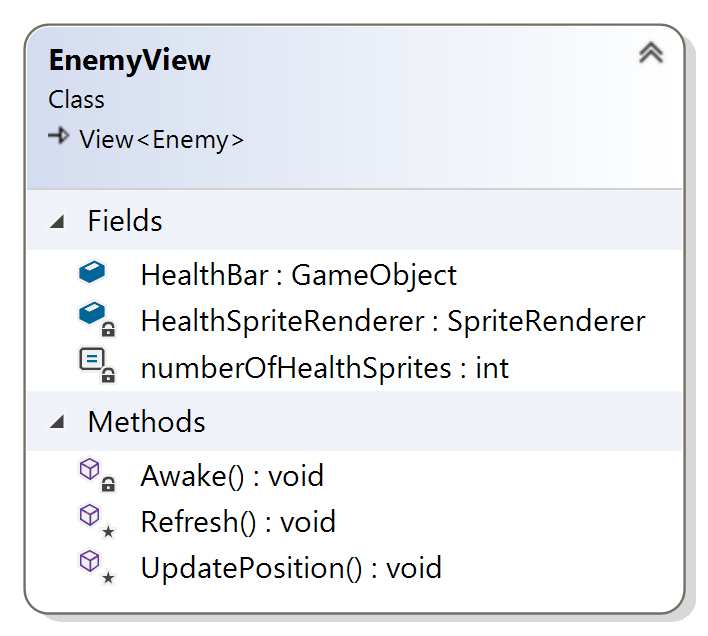
### ViewManager

* A **ViewManager** egy statikus osztály.
* A **View** kollekcióban tárolja az összes, megjeleníthető nézet prefab-et.
* A konstruktor lefutása során betölti a prefab-eket, és eltárolja őket a nevük alapján.
* Ezek később a **GetView** metódussal le is kérdezhetők, amennyiben új elem megjelenítésére van szükség.

### BuildingView

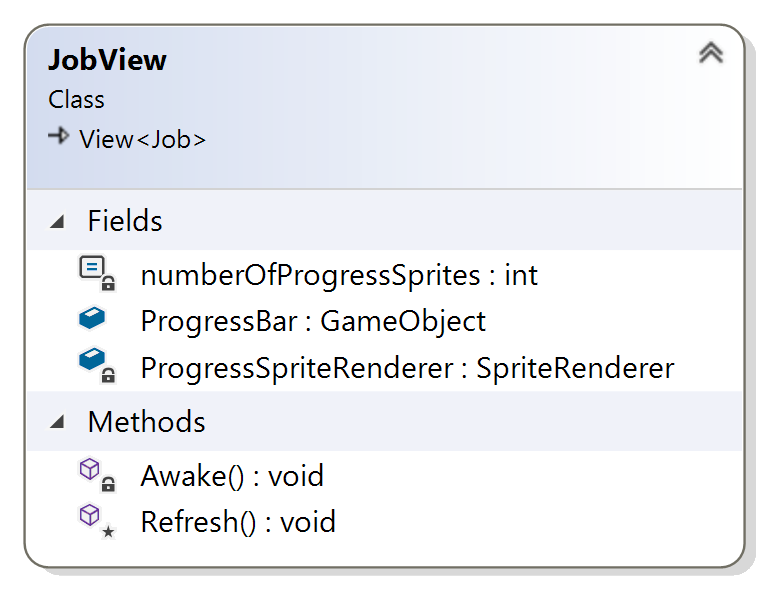
* A **BuildingView** egy épület megjelenítéséért felel.
* A **View** leszármazottja, generikus paraméterként a Building osztályt adja át.
* Ügyel arra, hogy egyes épületek összekapcsolódnak szomszédaikkal, valamint, hogy néhány épület megjelenése variálható.

### EnemyView

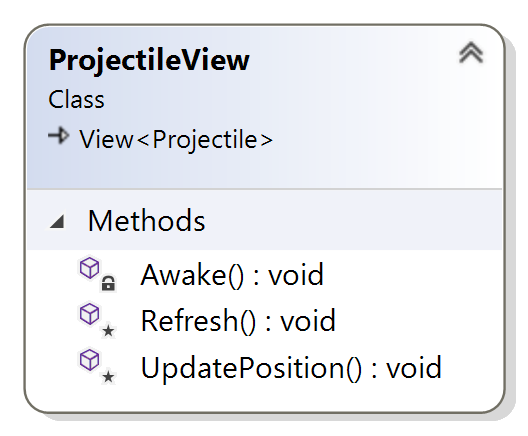


* Az **EnemyView** ellenségek megjelenítésére alkalmas. Ugyancsak a View osztály leszármazottja, generikus paraméterként az Enemy osztályt átadva.
* Rendelkezik egy hátralevő életet jelölő csíkkal, a **HealthBar**-ral. A játékos így nyomon tudja követni az egyes ellenségek életerejét.
* A csíkhoz tartozik egy sprite megjelenítő, a **HealthSpriteRenderer**.
* A **numberofHealthSprites** egy konstans mező, azt jelzi, hogy hány féle megjelenítési fázisa lehet az életerőt jelző csíknak.
* A **Refresh** metódus itt ügyel a két mező közötti átmeneti állapotra is az ellenség mozgása közben.

### JobView

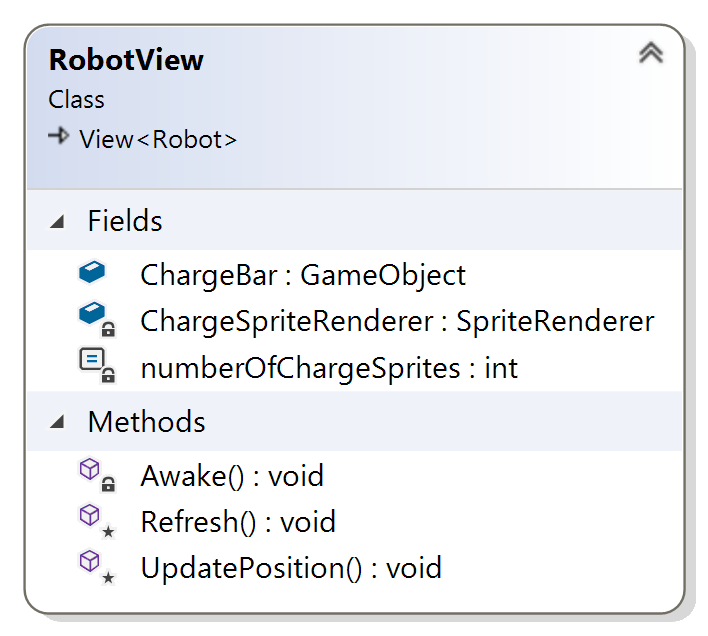
* A **JobView** felel az egyes feladataik nézetének kezeléséért.
* Rendelkezik a **ProgressBar** adattaggal, ami megjeleníti a feladat aktuális előrehaladását.
* Ehhez tartozik a **ProgressSpriteRenderer**, ami a sprite megjelenítését végzi.
* A progressziót mutató sprite-ok száma a **numberOfProgressSprites** konstans változóban tárolódik.

### ProjectileView

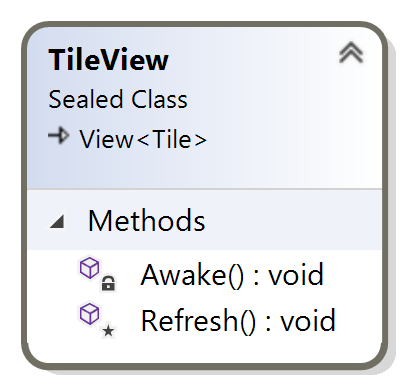


* A **ProjectileView** a lövedékek nézete.

### RobotView

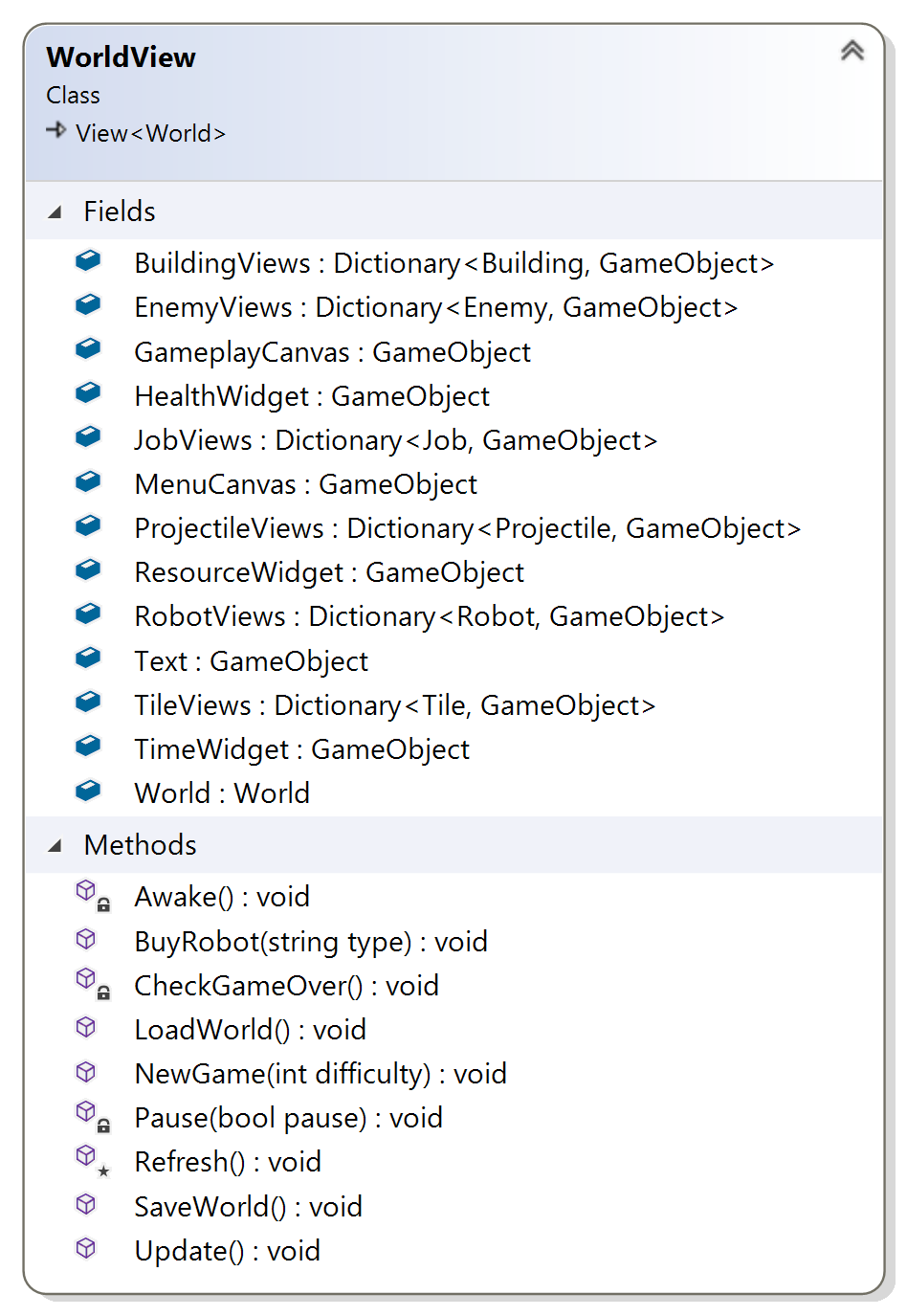
* A robotokat a **RobotView** osztállyal lehet megjeleníteni.
* Az aktuális töltődési állapotuk is látható, ezt a **ChargeBar** GameObject és a **ChargeSpriteRenderer** sprite megjelenítő végzi.
* A töltődés kijelzése lépcsőzetesen működik, a lépcsőfokok számát a **numberofChargeSprites** tartalmazza.

### TileView



* A **TileView** egy egyszerű, mezők megjelenítésére szolgáló osztály.

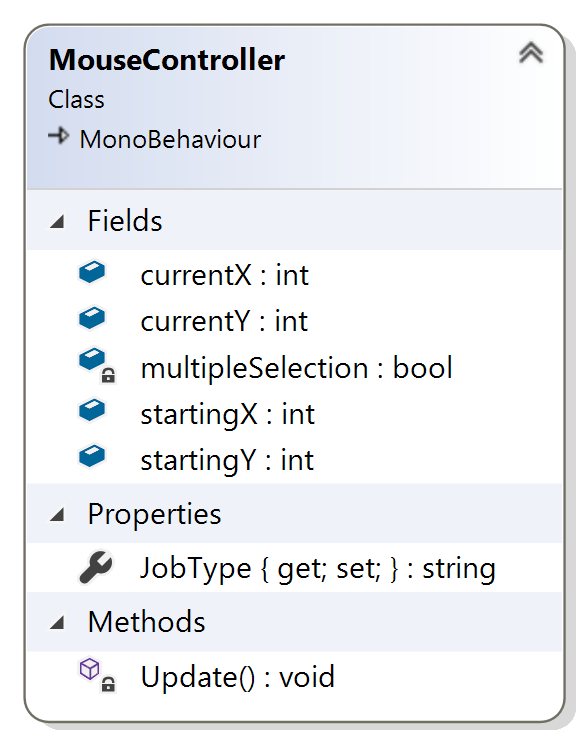
### WorldView

* A **WorldView** a legfontosabb nézet, a világ megjelenítése a feladata, emiatt tárolja az összes többi nézetet.
* A **BuildingViews** dictionary épületekhez rendeli a nézeteiket.
* Az **EnemyViews** az ellenségek megjelenítőit tárolja.
* A **GameplayCanvas** a játék futása közben elérhető menüelemeket tartja nyilván, például a gombokat és a kistérképet.
* A **HealthWidget** a főhadiszállás aktuális életét jelzi ki.
* A **JobView** tárolja a feladatok nézeteit.
* A **MenuCanvas**-on helyezkednek el a játék szüneteltetésekor megjelenő menüpontok, mint például az új játék kezdése gombok, a kilépés gomb, valamint a mentésért és betöltésért felelős gombok.
* A **ProjectileViews** tartja számot az aktuálisan játékban lévő lövedékek nézeteit.
* A **ResourceWidget** jeleníti meg a játékos által felhasználható erőforrások számát.
* A **RobotViews** kollekcióban vannak a robotok megjelenítői.
* A **Text** mező tárolja a játékosnak szánt aktuális üzenetet. Ez lehet a játék vége üzenet, gratuláció a nyeréshez stb.
* A **TileViews** dictionary tárolja a mezők nézeteit.
* A **TimeWidget** megjelenítő mutatja a túléléshez szükséges hátralevő időt.
* A világra való referenciát a **World** nevű változóban tartja.
* Az egyes robotok vásárlására használható gombok a **BuyRobot** metódust hívják meg, átadva a robot típusát.
* A **SaveWorld** és a **LoadWorld** függvényeket hívják meg a mentő és betöltő gombok. Ezek értesítik a modellt a szükséges műveletekről.
* A **NewGame** metódus hívódik meg, ha a felhasználó valamelyik új játék gombra kattint. A nehézség is itt adódik át, paraméterként.
* A escape billentyű megnyomására a **Pause** függvény hívódik meg, az aktuális állapottól függő paraméterekkel.
* Mivel a nézetek a MonoBehavior osztályból öröklődnek, a Unity motor automatikusan, folyamatosan meghívja az Update metódusukat, átadva az előző hívás óta eltelt időt. Ezt az időt adja tovább az aktuális világ objektumnak, ami elvégzi a szükséges frissítéseket. A nézet ezen felül még időszerűsíti a felhasználható erőforrásokat, a hátralevő életet és idő megjelenítő kijelzőket.

### CameraController

* A **CameraController** osztály felel a kamera mozgatásáért.
* Tárol egy referenciát a játék aktuális kamerájára a **Camera** változóban.
* A **PanningBorder** változó határozza meg, hogy milyen közel kell lennie a kurzornak a képernyő széléhez, hogy a kamera megmozduljon.
* A **PanningLowerLimists** és **PanningUpperLimits** változók tárolják, hogy milyen messzire lehet maximálisan mozgatni a kamerát.
* A **PanningSpeed** változó a kamera mozgási sebességét tartalmazza.
* A **ZoomingSpeed** pedig a kamera közelítési és távolítási sebességét tárolja.

### MouseController

* A **MouseController** osztály felel az egér kezeléséért, az azzal való interakciók lebonyolításáért.
* A **currentX** és **currentY** változókban tárolódik a kurzor aktuális pozíciója, egészre kerekítve, így pontosan meghatározható, hogy az egér melyik mező fölött áll.
* A **multipleSelection** logikai változó igaz értéket vesz fel, amennyiben a játékos az egér bal gombját nyomva tartva több mezőt is kijelöl.
* Ilyenkor a kijelölés kezdetének pozícióját a **startingX** és a **startingY** adattagok tartalmazzák.
* A **JobType** változóban pedig az aktuális, kiadásra szánt feladat típusa van tárolva.

## Tesztelés

### Egységtesztek

A szoftver modellje már a fejlesztés alatt is alapos tesztelésen esett át. A projekt végére összesen 118 egységteszt gyűlt össze. Ezek megtekinthetők a forráskódban, Visual Studio környezetben pedig le is futtathatók. Ügyeltem a modell összes osztályának, azon belül pedig a legtöbb függvény lefedésére.

Néhány fontosabb, gyakrabban előforduló teszteset:

* Konstruktorok által beállított mezők helyessége
* ArgumentNullException-ök kivédése, amennyiben egy függvény egyik paramétere null
* NullReferenceException-ök lekezelése
* Boolean értéket visszaadó műveletek tesztelése a lehetséges paraméter kombinációkkal
* Objektumok helyes létrejöttének tesztelése
* Esetleges InvalidOperationException-ök okozóinak kiderítése és megoldása
* Túlcsordulások során dobott OverFlowException-ök megelőzése
* KeyNotFoundException lekezelése, amennyiben a kollekció nem tartalmazza a megadott kulcsú elemet.

### Végfelhasználói tesztek

Az egyes funkciók a beépítésükkel párhuzamosan lettek végfelhasználói szempontból tesztelve. Emiatt gyakran befolyásolták a fejlesztés menetét és az implementáció módját. Néhány példa ezekre:

* A JobManager osztály kizárólag a tesztelésekből elvont következtetések miatt jött létre. Ez előtt előfordulhatott, hogy ha egy robot sem tudott elérni egy feladatot, a processzor nagyon leterhelődött, ugyanis a robotok sorban elvállalták a munkát, próbáltak hozzá útvonalat keresni (ez a legmegterhelőbb folyamat az egész szoftverben), ez nem sikerült, majd lemondtak róla, így egyből próbálkozott a következő. Ez igényelte azt a megoldást, hogy egy robot egy feladatot újra csak pár másodperc elteltével vállalhat el újra. Ezzel a probléma megoldódott, és amint a feladat elérhető lett akár egy robot számára is, volt is lehetőség azt végrehajtani.
* A kamera mozgatásának bekorlátozására is a tesztelés során derült ki az igény. Ez előtt lehetőség volt végtelen sokáig mozgatni a kamerát, messze elhagyva ezzel a pályát.
* A több feladat kiosztására használt Drag&Drop módszer eredetileg nem volt tervben, azonban a tesztelés során kiderült, hogy az azonos típusú feladatok egyesével való kiadása több mezőre unalmas, és nem elég gyors a játékmenet többi eleméhez képest.
* A játék elvesztésének tesztelése során merült fel az ötlet, hogy az ellenségek a bázisba érkezésükkor ne konstans sebzést okozzanak annak, hanem a hátralevő életüknek megfelelőt. Így a játékos akkor is jutalmazva van, ha ugyan nem pusztított el egy ellenséget, de mégis jelentős sebzést okozott neki.
* A pálya nagysága és az azon való események szétszórtsága indokolttá tette egy kis térkép bevezetését, ami nagyban segíti a játék átláthatóságát.

### Skálázhatóság

A program elsősorban otthoni, személyi számítógépekre lett fejlesztve, így nem igényel sok erőforrást. A legnagyobb terhelés, amit elő tudtam idézni a nehéz fokozatú játék során történt, nem sokkal a pálya megnyerése előtt. Ekkor megközelítőleg 150 ellenség tartott a főhadiszállás felé, 40 robot dolgozott párhuzamosan, és 50 védelmi épület akadályozta az ellenséget. Észrevehető lassulást nem tapasztaltam.

### Egyéb fejlesztési lehetőségek

A játékszoftver nem rendelkezik hanggal. Túl időigényes feladat lett volna, szakmai tudásomat pedig aránytalanul kis mértékben növelte volna.

A szoftver felépíthető ugyan Linux, Mac, sőt Android környezetben is, ezek tesztelése és támogatása ugyancsak túl sok időt vett volna igénybe.