

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAI

TANSZÉK

Tervezési minták a Játékfejlesztésben

Témavezető:

Kovácsné Puszta Kinga
tanársegéd, MSc

Szerző:

Hajdu Marcell Ferenc
programtervező informatikus BSc

Budapest, 2021

Témabejelentő

A szakdolgozatomban a modern játékfejlesztésben leggyakrabban használt tervezési mintákat mutatom be egy 2D-s játék fejlesztése során. Ennek az a célja, hogy megmutassam, hogy a játékfejlesztésben, hogyan és pontosan milyen tervezési mintákat érdemes alkalmazni. Mindezzel létrehozok, jól átlátható, könnyen bővíthető kódot. Ezen tényezők fontosságát azért tartom kimagaslónak, mivel a mai játékfejlesztésnek és abba történő belépésnek a legnagyobb problémája a projektek be nem fejezése, ami legtöbbször az előre meg nem tervezett, gyakran nem átlátható, nehézkesen bővíthető kód. Maga a 2D-s játék célja, hogy egy akadályokkal és ellenfelekkel teli pályán végig navigálunk és elérjük a végcélt. Ehhez az utóbbi években nagy népszerűségnek örvendő Unity játékfejlesztői motort használnom, amiben komponens alapú szkripteket lehet létrehozni C# nyelvben.

Tartalomjegyzék

1. Bevezetés	5
2. Felhasználói dokumentáció	6
2.1. A játék rövid leírása	6
2.1.1. Célcsoportok	6
2.2. Rendszerkövetelmények	7
2.2.1. Játék indítása	7
2.3. Funkciók ismertetése	8
2.3.1. Grafikus Felhasználói Interfész (GUI)	8
2.3.2. Pályák	16
2.3.3. Irányítás	18
2.3.4. Tárgyak	19
2.3.5. Varázslatok	20
2.3.6. Státuszeffektusok	20
2.3.7. Ellenfelek	21
3. Fejlesztői dokumentáció	23
3.1. Tervezés	23
3.1.1. Kezdeti tervezés	23
3.1.2. Unity	25
3.1.3. S.O.L.I.D.	28
3.1.4. Tervminták	29
3.2. Megvalósítás	34
3.2.1. Felhasználói esetdiagram	35
3.2.2. Státusz rendszer	36
3.2.3. Varázslás rendszer	37

TARTALOMJEGYZÉK

3.2.4. Játékos rendszer	38
3.2.5. Ellenség rendszer	42
3.2.6. Adattárolás	43
3.2.7. CI/CD pipeline	45
3.3. Tesztelés	46
3.3.1. Egységtesztek	46
3.3.2. Kézi Tesztelés	47
3.3.3. Tesztelési konklúzió	49
4. Összegzés	50
4.1. Köszönetnyilvánítás	50
Irodalomjegyzék	51

1. fejezet

Bevezetés

A játékfejlesztésben, mint általánosságban a szoftverfejlesztésben nagy hangsúlyt kell fektetnünk a megfelelő tervezésbe hogy jól skálázható, egyszerűen átlátható kódot adjunk ki kezeink közül. Azonban a modern törekvések mint például a vizuális kódolás és a könnyen átlátható, valamint beletanulható keretrendszerök, és fejlesztői környezetek elterjedése miatt gyakran olyan emberek látnak hozzá nagy rendszerek, mi esetünkben játékok készítéséhez, akiknek nincs előzetes ismerete ilyenek megvalósításához. A bőségesen elérhető egyszerű eszközökkel hamar el lehet készíteni egyszerűbb azonban nehézkesen (néhány esetben semmilyen módon sem) továbbfejleszthető játékokat. Ez azonban csak a jéghegy csúcsa. Rengeteg kezdő játékfejlesztő szimplán a játéka befejezéséig sem jut el. Legtöbb esetben ez azért történik, mert a kód átláthatatlan módosítások benne nehezen észrevehető mellékhatásokat okozhatnak, miknek javítása sok energiát követel. Egyszerűbb előlről kezdeni, más projektnek nekiállni, vagy egyszerűen feladni.

Szerintem azonban ezen problémák könnyedén kiküszöbölni lehetők!

Tervezési minták régóta léteznek és fejlődnek a tradicionális szoftverfejlesztésben és a nagyiparos játékfejlesztésben is, azonban a független/kezdő játékfejlesztésben nem régiekben kezd csak elterjedni, pedig meglátásom szerint néhány alapelvevel (mint például a S.O.L.I.D) összekapcsolva a feljebb kifejtett problémára tökéletes megoldást nyújt. Pontosan, hogy és miért az, amit ez a Szakdolgozat fog megválaszolni, egy rövid 2D-s játékon keresztül.

2. fejezet

Felhasználói dokumentáció

2.1. A játék rövid leírása

A játék Hack-and-Slash stílusú. A célunk, hogy haladjunk előre egy pályán, ahol különböző nehézségű és számosságú ellenfelek jönnek velünk szembe. A pályán találhatunk felvehető tárgyakat, amik szituációtól függően erősítenek bennünket. Halál esetén újraéledünk egy előre meghatározott ellenőrző ponton, ahonnan folytathatjuk a játékot. A pálya végén értékel minket a játék, és ha más játékosokhoz képest (mint egy árkád játékban, local high score) jobban teljesítettünk, vagy még nem telt be az eredménytábla, akkor az előbb említett táblára felkerülünk. A játék 3 pályát/szintet tartalmaz, amik egyre nehezebbek, ezen felül állásunkat el tudjuk mentei, valamint betölteni. Mindezzel egy kohézív gyors, kihívásokkal teli játékélményt kínálunk.

2.1.1. Célközönség

A játék azon eberek számára lehet érdekes, akik szeretik a gyors kihívással teli árkát stílusú Hack-and-Slash játékokat. 15 életévet minimum betöltött embernek ajánljuk, mivel a játék vért és erőszakot (ellenfelek legyőzése) tartalmaz.

2.2. Rendszerkövetelmények

Adatok	Minimum követelménye	Ajánlott követelmény
CPU	Intel i5-8250U	Intel i5-8600K
GPU	Nvidia GeForce MX150	Nvidia GTX 1050Ti
RAM	8Gb	16Gb
OS		Windows, Linux
Disc		300Mb

2.1. táblázat. rendszerkövetelmények

2.2.1. Játék indítása

A játék nem igényel telepítést. Futtatásához indítsa el a 'The Quest for the Thesis.exe'-t (kattintson rá bal egérgombbal kétszer Windows-on).

2.3. Funkciók ismertetése

A következő fejezetekben a játék funkcióinak részletesebb ismertetése lesz kifejtve. Hogyan is néz ki a játék, és mivel találkozhatunk miután elindítottuk.

2.3.1. Grafikus Felhasználói Interfész (GUI)

Kettőféle Grafikus Felhasználói Interfész különböztethetünk meg a játék során. Az egyik, ami tisztán információt közöl velünk, a másik amivel interaktálhatunk is.

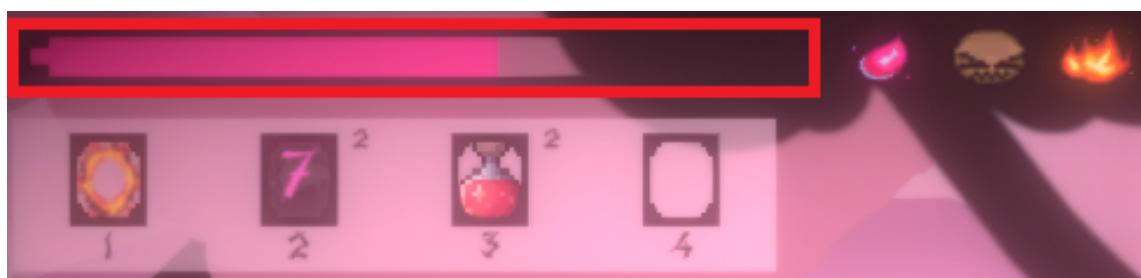
Tisztán információ közlő GUI-k

Játékon belüli indikátorok azok a grafikus elemek, amik visszajelzik nekünk a játékos aktuális állapotát.



2.1. ábra. Játékon belüli indikátorok

Itt látható a játékos jelenlegi élete.



2.2. ábra. Játékos életereje

A játékos képességei és tárgyai, valamint az idő addig amíg újra lehet használni a tárgyat/képességet.



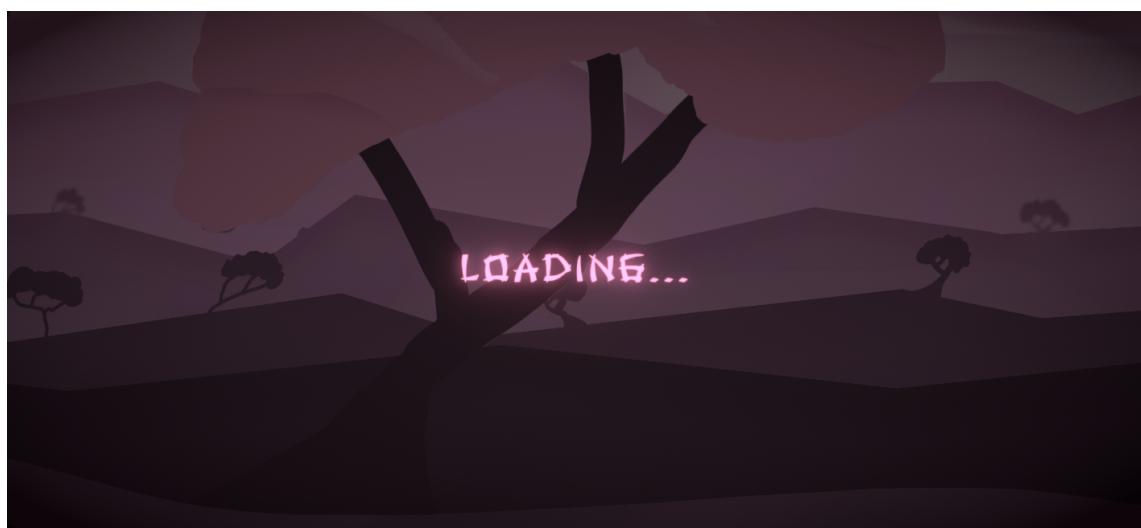
2.3. ábra. Játékos képességei és tárgyai

Valamint a játékost éppen érintő státuszeffekteket.



2.4. ábra. Játékost érintő státuszeffekteket

Ezeken felül ebbe a kategóriába tartozik még a töltőképernyő, ami a szintek között jelenhet meg.



2.5. ábra. Töltőképernyő

Valamint a halálképernyő, ami akkor jelenik meg, ha a játékos meghalt.



2.6. ábra. Halálképernyő

Interaktív GUI-k

Ezeken a képernyőkön vannak gombok, lenyíló listák, vagy csúszkák, amikkel interaktálhatunk a játékkal.

1. **Főmenü** az első GUI elem amit látunk a játék indításakor. Öt gomb látható ezen a felületen:

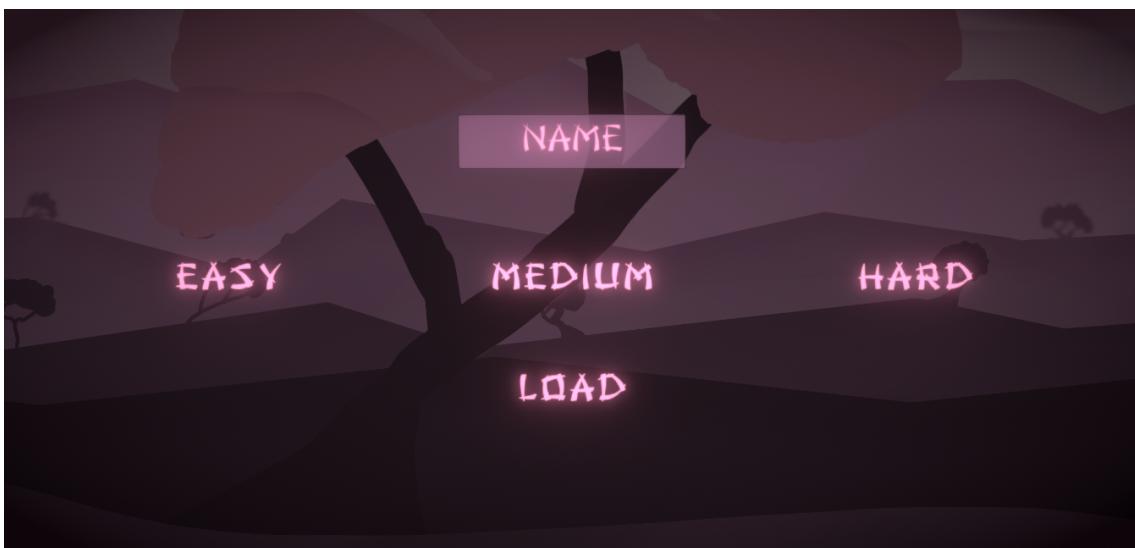
- *Start* ha erre a gombra kattintunk, akkor tovább lépünk a *Pályaválasztóra*.
- *Settings* erre kattintva megnyitjuk a *Beállítások 6* menüt.
- *Score Board*-ra kattintva eljutunk az *Eredménytáblára 4.*
- *Credit* gombra kattintva eljutunk a *Köszönhetnyilvánító 5* menüre.
- *Exit* gomb lenyomásával kilépünk a játékból.



2.7. ábra. A Kezdőképernyő

2. Pályaválasztó menün adhatjuk meg a nevünket ami később az Eredménytáblán 4 látható.

- Az *Easy*, *Medium* és *Hard* gombok lenyomásával átléphetünk az *Utasítások 3* menübe, és a játék betölti a megfelelő pályát.
- *Load* gomb lenyomását követően átlépünk a *Mentés és Betöltés 7* menübe.



2.8. ábra. A Pályaválasztó menü

3. Utasítások menüben láthatók a játék bemeneteire végrehajtott akciók.

- *Play* gomb lenyomását követően elkezdődik a játék.



2.9. ábra. Utasítások menü

4. Eredménytábla ahol a szinteken elért eredmények láthatók.

- Back gomb lenyomását követően visszatérünk az előző menübe.



2.10. ábra. Eredménytábla

5. Köszönetnyilvánító menün található a játék készítéséhez felhasznált asset-ek készítői.

- Back gomb lenyomását követően visszatérünk az előző menübe.



2.11. ábra. Köszönetnyilvánító menü

6. Beállítások menüben változtathatjuk meg az alapértelmezett beállításainkat.

- Bal felül választhatjuk ki egy lenyíló listából a kívánt felbontást.
- Jobb felül választhatjuk ki egy lenyíló listából a játékkablak üzemmódját.
- *Graphics Quality* menüpontban egy lenyíló listából kiválaszthatjuk a játék minőségi beállításait.
- *Main Volume* csúszkán a fő hangerőt tudjuk állítani.
- *Music Volume* csúszkán a zene hangerejét tudjuk állítani.
- *SFX Volume* csúszkán az effektek hangerejét tudjuk állítani.
- *Back* gomb lenyomását követően visszatérünk az előző menübe.



2.12. ábra. Beállítások

7. **Mentés és Betöltés** menün található a 4 mentési hely. Ahol ha már volt mentésünk, akkor a mentésben szereplő adatok jelennek meg, ha nem akkor az *Empty Slot* felirat.

- *Load* gombot megnyomva a kijelölt mentés fog betölteni.
- *Save* gombra kattintva a kijelölt mentési helyre elmentődik a jelenlegi állás (csak játék közben látható).
- *Back* gomb lenyomását követően visszatérünk az előző menübe.



2.13. ábra. Mentés és Betöltés menü

8. **Szünet** menüt a játék közben az **ESC** gomb lenyomásával hívhatjuk elő.

- *Resume* gomb lenyomását követően a játék folytatódik.
- *Save / Load* gombot megnyomva továbblépünk a *Mentés és Betöltés 7* menüre.
- *Exit* gomb lenyomását követően a játék bezáródik.



2.14. ábra. Szünet menü

9. **Vég** menü egy pálya végén jelenik meg.

- Legfelül látható a pálya elvégzését követő eredményünk.
- A szerzett pontunk alatt láthatóak a többi pályákat reprezentáló gombok, amikre kattintva a megfelelő pálya betöltődik (különböző pályákról érkezve a Vég menübe más pályák jelennek meg itt).
- *Load* gombot megnyomva továbblépünk a *Mentés és Betöltés 7* menüre.
- *Score Board*-ra kattintva eljutunk az *Eredménytáblára 4*.
- *Credit* gombra kattintva eljutunk a *Köszönetnyilvánító 5* menüre.
- *Exit* gomb lenyomását követően a játék bezáródik.



2.15. ábra. Vég menü

2.3.2. Pályák

A játékban három pályát különítünk el: Easy, Medium, Hard

- **Easy** pálya a névből eredően a legkönnyebb. A tárgyak és az ellenfelek megismerésére tökéletesen alkalmas.



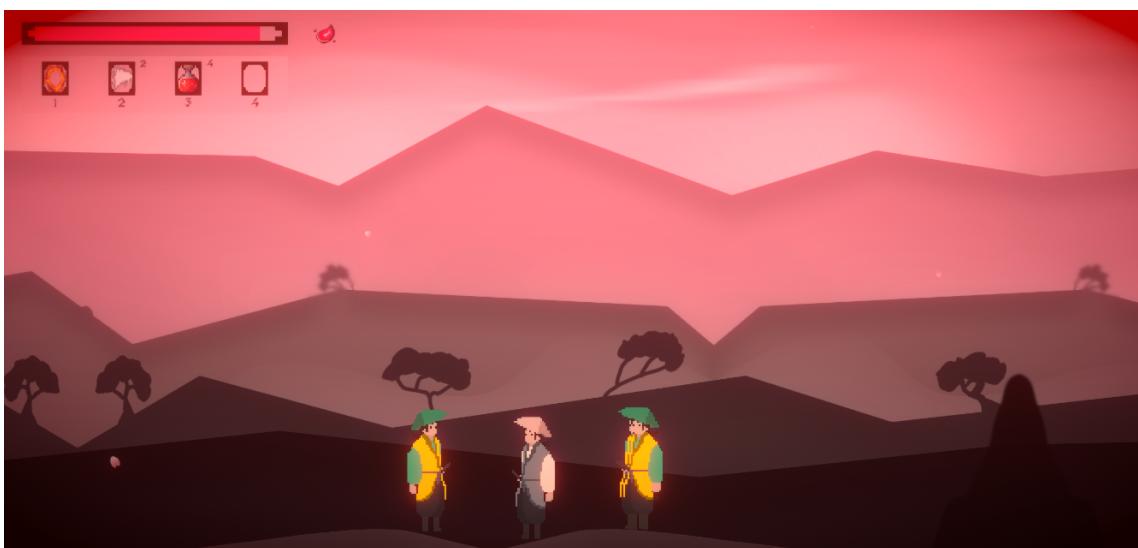
2.16. ábra. Easy pálya

- **Medium** pálya közepes nehézségű. Több ellenség érkezik felénk és a felvehető tárgyak ritkábbak.



2.17. ábra. Medium pálya

- **Hard** pálya a legnehezebb. A tárgyak kevesek, az ellenségek számosak és az erősebb fajtából valók.

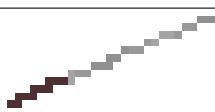
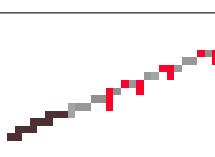
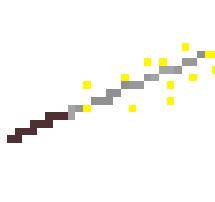


2.18. ábra. Hard pálya

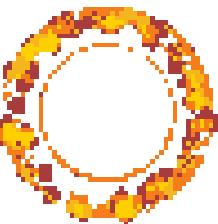
2.3.3. Irányítás

Akció	Gomb
Bal irányú mozgás	A
Jobb irányú mozgás	D
Ugrás	SPACE
Gyors hirtelen mozgás	Baloldali ALT
Sima Támadás	Baloldali egérgomb
1, 2, 3, 4 gombok	megfelelő helyen lévő tárgy használata
Tárgyak felvétele	E
Játék megállítása	ESC

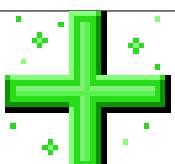
2.3.4. Tárgyak

Név	Kép	Leírás	
Gyógyító ital		Megszünteti az összes státuszeffektust.	2.3.4.2
Életerő ital		Gyógyítja a játékost és a <i>Gyógyul</i> 2.3.6.6 státuszeffektet rakja rá.	2.3.4.3
Tűz varázsfókusz		Lehetővé teszi a <i>Tűzkarika</i> 2.3.5.2 varázslat használatára a játékost.	2.3.4.4
Jég varázsfókusz		Lehetővé teszi a <i>Jéglándzsa</i> 2.3.5.3 varázslat használatára a játékost.	2.3.4.5
Katana		Egy egyszerű kard amivel sebzed az ellenségeket.	2.3.4.6
Masamune		Gyengén sebzi az ellenségeket, azonban <i>Vérzés</i> 2.3.6.2 státuszeffektet rak rájuk.	2.3.4.7
Muramasa		Erősen megsebzi az ellenségeket, és <i>Kábult</i> 2.3.6.5 státuszeffektet rak rájuk, azonban alacsony a támadási sebessége.	2.3.4.8

2.3.5. Varázslatok

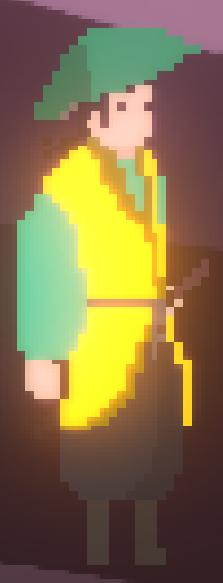
Név	Kép	Leírás	
Tűzkarika		Rengeteg sebzést okoz az összes közeli ellenségnek, és az <i>Égés</i> 2.3.6.3 státuszeffektet rakja rájuk, nagy az újrahasználási ideje.	2.3.5.2
Jéglándzsa		Megsebzi az ellenségeket, és <i>Fagyás</i> 2.3.6.4 státuszeffektet rak rájuk.	2.3.5.3

2.3.6. Státuszeffektusok

Név	Kép	Leírás	
Vérzés		Időközönként megsebzi kicsit a célpontot.	2.3.6.2
Égés		Időközönként megsebzi a célpontot.	2.3.6.3
Fagyás		Időközönként megsebzi és lelassítja a célpontot.	2.3.6.4
Kábult		Rendkívül megsebzi a célpontot és lezárja a mozgási lehetőségeit egy időre.	2.3.6.5
Gyógyul		Időközönként gyógyítja a célpontot.	2.3.6.6

2.3.7. Ellenfelek

Minden ellenségnak kettő típusa van: járőr és őr. Az őr egy helybe áll és vár a játékosra. A járőr kettő pont között mozog.

Név	Kép	Leírás	
Alap Ellenség		A leggyengébb ellenség, ami egy <i>Katanát</i> 2.3.4.6 használ a fő fegyvereként.	2.3.7.1 2.3.7.2
Erős Ellenség		Egy lényegesen erősebb ellenség, ami egy <i>Masamunet</i> 2.3.4.7 használ a fő fegyvereként.	2.3.7.3

Tűzvarázsló		Egy varázsló egység, ami egy <i>Muramasat</i> 2.3.4.8 használ a fő fegyvereként, azonban képes a <i>Tűzgyűrű</i> 2.3.4.8 varázslat használatára is.	2.3.7.4
Jégvarázsló		Egy rendkívül veszélyes varázsló egység, ami egy <i>Masamunet</i> 2.3.4.7 használ a fő fegyvereként és képes a <i>Jéglándza</i> 2.3.4.8 varázslat használatára is.	2.3.7.5

3. fejezet

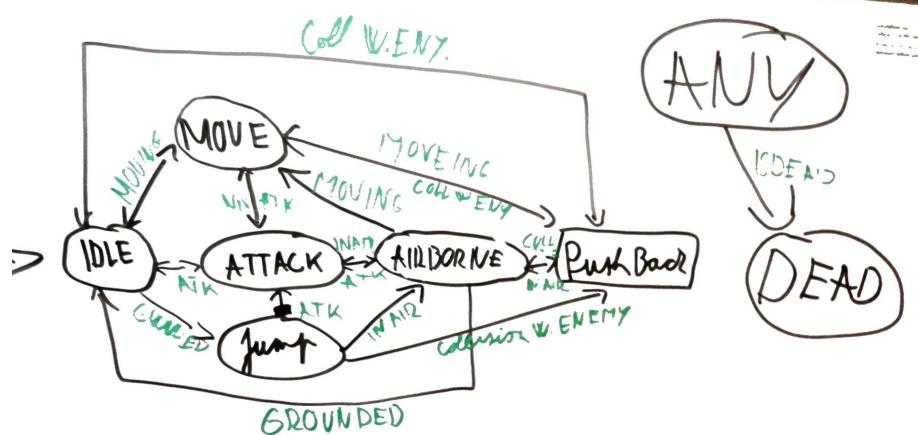
Fejlesztői dokumentáció

3.1. Tervezés

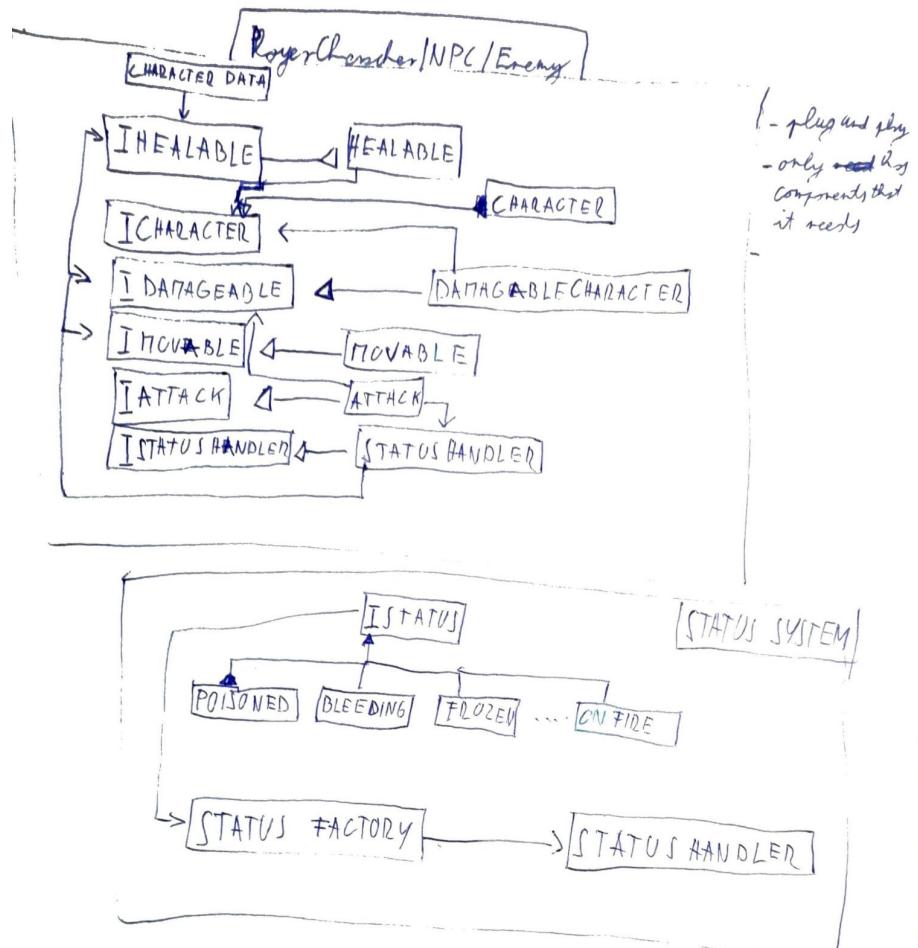
3.1.1. Kezdeti tervezés

Mivel egy játék (legyen akármekkora is) egy bonyolult rendszer, és mivel alapértelmezetten is egy jó gyakorlat, a projekt elejétől kezdve meg szerettem volna tervezni, átfogólag a különböző rendszereket. Tudtam előre, hogy a Unity játékmotort fogom használni a példajáték elkészítéséhez, és a tervminták alkalmazására, amiknek fontosságára Jason Weimann[7], DapperDino[10] és a Game Programming Patterns-ben[1] olvasottak inspiráltak. Azt is figyelmiben tartottam, hogy szakdolgozatom célja a tervezési minták és programozási elvek felhasználása a játékfejlesztésben és egy nem feltétlenül teljes mértékben kidolgozott játék piacra dobása.

Mindezek után három dolgot tartottam fontosnak eleinte, a játékos (és ellenségek) statemachine-ját, egy általánost leírás az alap rendszerek kapcsolatáról, valamint a játékom elkészítése a S.O.L.I.D elvek szerint.



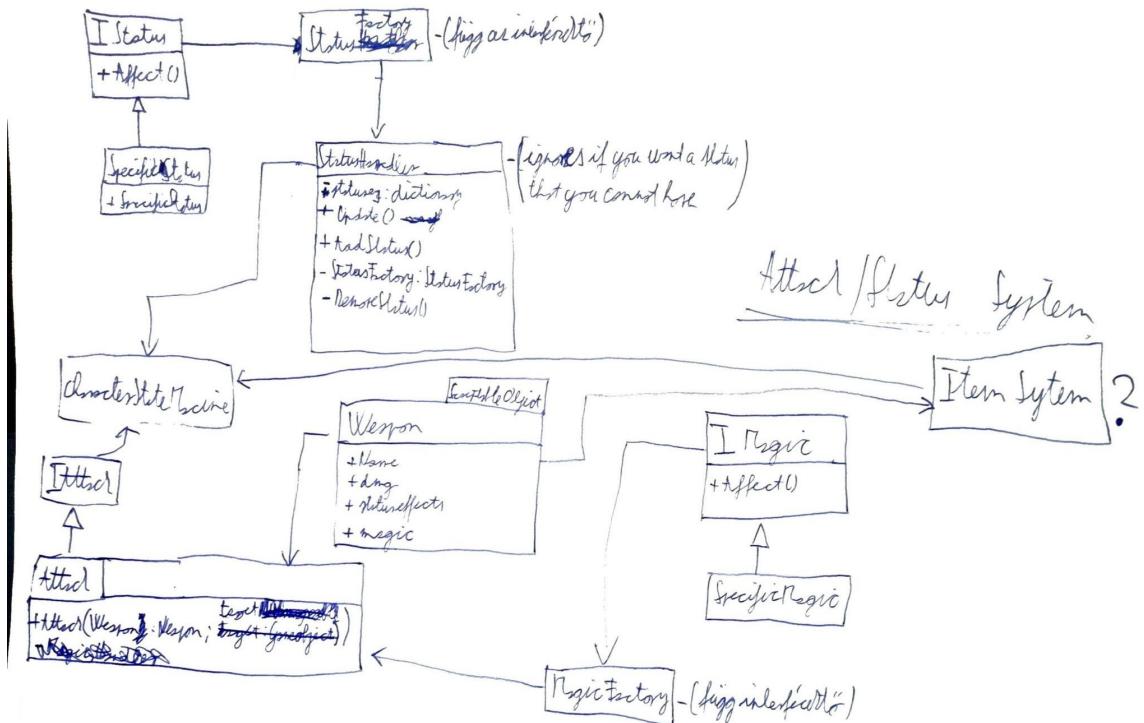
3.1. ábra. Kezdeti statemachine



3.2. ábra. Rendszerek általános leírása

Azonban hamar rájöttem, hogy bizonyos bonyolultan rendszerek kapcsolatát

érdesmes jobban kifejteni. Ilyen volt a támadással foglalkozó csoport is, ahol három kisebb rendszer összeépítését kellett kivitelezni. Ezek a varázslatok, státuszeffektusok és maga a támadás rendszere voltak.



3.3. ábra. Támadási rendszer

3.1.2. Unity

A Unity játékmotor egy cross-platform komponens alapú játékfejlesztői motor. Ebben a fejlesztői környezetben, relatívan egyszerűen hozhatunk létre játékokat (főleg az új visual scripting segítségével, hiszen így fejlesztők nem kényszerülnek rá a C# nyelv elsajátítására). Unity-ben 2 fontos dolgot kell megértenünk, egyik az Editor, amit használni fogunk a színterünk komponálására, valamit a komponenseink (GameObjects) létrehozására, a másik a monoBehaviour-ök megértése, hiszen ez az osztály az összes Unity script ősosztálya.

Először is a **Unity Editor**-ról beszélnek. Ez a Unity Engine grafikus interfésze, innen érhetjük el a Unity által biztosított funkciókat, mint például az *animator*, ahol animációk átmeneteit határozzatjuk meg, a *shader graph*-ot, ahol egy grafikus felületen készíthetünk vertex és fragment shader-eket, vagy akár a *test runner*-t,

hogy párat említsek.

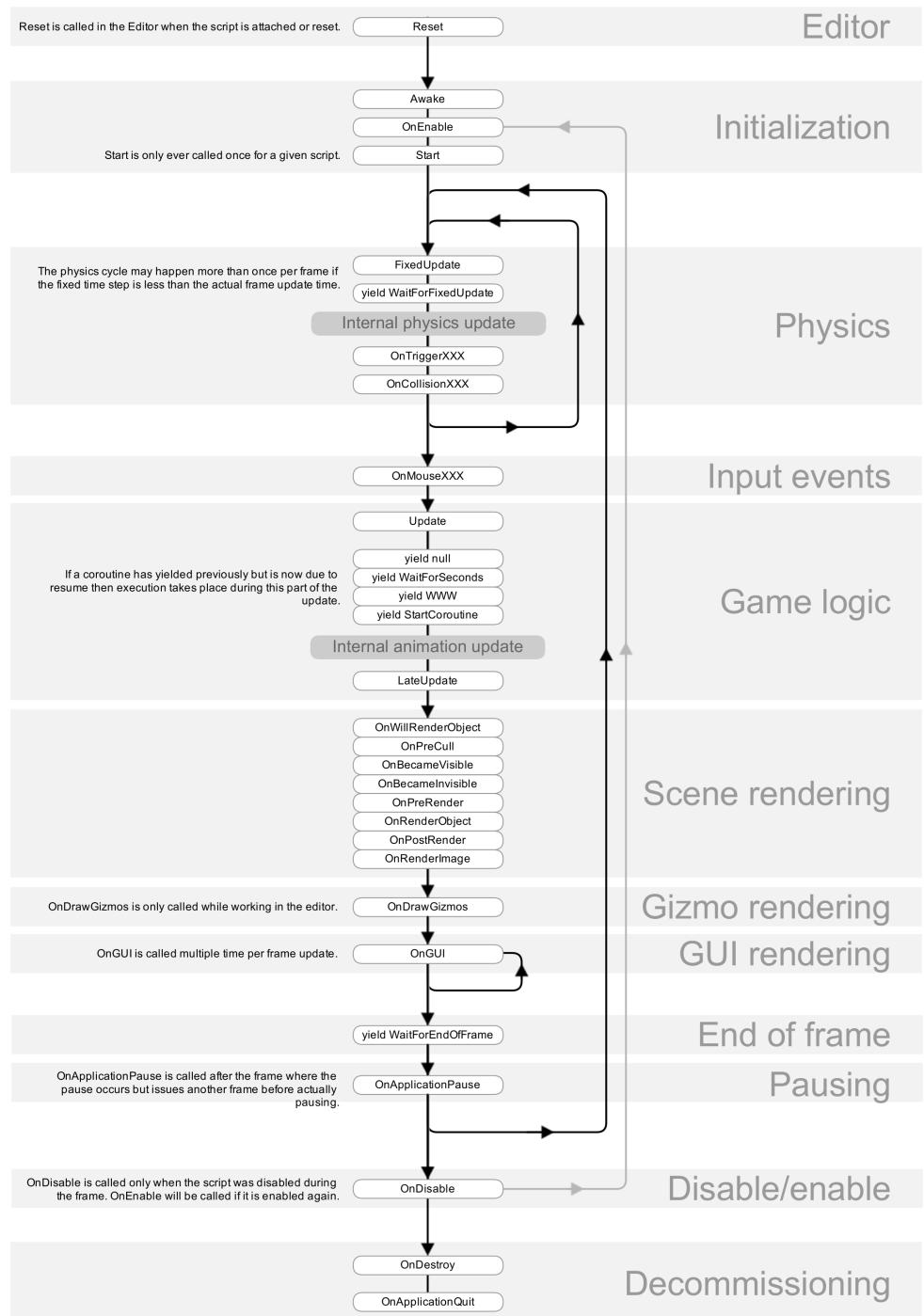
Azonban a 2 legfontosabb funkciója az Editornak a színtereink(scenes) és *gameObject*-jeink komponálása. A színtereinken készíthetünk pályákat, menüket, adhatunk hozzá hangforrásokat, fényeket vagy utóhatásokat, nem mellesleg ide kell hozzáadnunk a *gameObject*-eket, hogy megjelenjenek.

A *gameObject*-eket el lehet képzelni úgy, mint egy üres dobozt, amihez különböző komponenseket adhatunk. Ha azt akarjuk, hogy ne láthatatlan legyen a dobozunk adhatunk hozzá egy *Sprite Renderer*-t, ha akarjuk, hogy a fizika hasson rá akkor hozzáadhatjuk a *Rigidbody* komponenst és így tovább. Egy *gameObject* rengeteg minden lehet és több funkciót is elláthat, nem mellesleg rájuk aggathatjuk a saját script-jeinket is, hogy valami egyedi hatást/viselkedést érjünk el.

Most, hogy beszéltem az Editor-ról, a helyről, ahol a játék építése folyik, beszélnék egy kicsit az egyik legfontosabb építőelemről, a **monoBehaviour**-ről, ami az összes script-ünk ősosztálya, amit egy *gameObject*-re rá akarunk rakni, mint egy komponenst. A monoBehaviour és minden leszármazottja egy olyan osztály, amit nem lehet a **new** kulcsszóval példányosítani, csak az *addComponent* függvénytel. Ebből következve ezen osztályok, csak egy *gameObject*-en élhetnek és kötöttek annak élettartalmához.

A monoBehaviour biztosít számunkra pár fontos metódust is [2].

- **Start** - inicializációs fázis utolsó lépése.
- **FixedUpdate** - fizikai frissítés. Itt lehet olyan fizikai számításokat kezelní, amiket minden ütemezett fizikai frissítésben végre akarunk hajtani.
- **OnCollisionXXX** - mi történjen ha valami ütközés történik.
- **Update** - elérhető mint egy monoBehaviour-re korlátozott *gameLoop*. Akkor használjuk, ha ismétlődő számításokat akarunk végrehajtani (minden frame-ben).



3.4. ábra. MonoBehaviour életciklusa

3.1.3. S.O.L.I.D.

A S.O.L.I.D. elvek[3]

S - Single-responsibility Principle, avagy az *Egyetlen felelősség elve*

Ennek az elvnek a lényege, hogy egy osztálynak, egyetlen egy oka lehet a változásra, vagyis egy céljának kell lennie. Ennek az elvnek a megvalósítása Unity-ben triviális, hiszen kisebb odafigyeléssel ez adódik is a komponens alapú összetételeből.

O - Open-closed Principle, avagy az *Nyílt/zárt elv*

Itt a legfontosabb tudnivaló, hogy egy objektumnak képesnek (nyitottnak) kell lennie a bővítésekre, azonban ezeknek a bővítéseknek nem szabad módosítania a meglévő osztályt, zárva kell lennie a módosításokra tekintve. Ugye ez az elv interfések és ősosztályok megvalósításával könnyen elérhető, hiszen ha megvannak a mintáink, hogy miképp kell egy osztálynak felépülnie, és azt meg is valósítottuk, esetleg egységesztekkel le is védtük működését, akkor annak módosítása hiba nélkül nehéz lesz, azonban a bővítése triviális. Az a fontos itt, hogy a régi kódban lévő logika a bővítés miatt ne kényszerüljön az új kód rész függésére, ebben az esetben lehet érdemes meggondolni a kód feldarabolását a *Egyetlen felelősség elve* mintájára.

L - Liskov Substitution Principle, avagy az *Liskov helyettesítési elv*

Liskov helyettesítési elv lényege, hogy minden osztályt be kéne tudni helyettesíteni annak ősosztályába, interfészébe. Ez egy rendkívül fontos megállapítás, hiszen ezzel meg tudjuk szüntetni a függést egy specifikus osztályimplementációtól, ezáltal képesek leszünk könnyen cserálni konkrét működést egy komponensben/osztályban annak megváltoztatása nélkül.

I - Interface Segregation Principle, avagy az *Interfész elválasztási elv*

Az elv azt mondja ki, hogy egy osztálynak nem szabad függenie olyan interfések től, függvényektől, amiket nem valósít meg, nem használ. Ez könnyen felfogható, úgy mint az *Egyetlen felelősség elve* alkalmazása interfésekre, hiszen így több kisebb közelebb kapcsolódó interfést kapunk, amiket szabadon implementálhat egy osztály szükségszerűen, ellentétben egy nagy interfésszel.

D - Dependency Inversion Principle, avagy az *Függőség megfordítási elv*

Ez az elv azt mondja ki, hogy az entitásoknak absztraktiókon kell függeniük, valamint, hogy magasabb rendű rendszereknek nem szabad függeniük az alacsonyrendűektől. Ezt a komponens alapú Unity-ben könnyen elérhetjük, hiszen

ha osztályain egy interfészt implementálnak, ami megfelel az absztrakcióknak és ezen osztályainkat a *Liskov helyettesítési elv* és a *Egyetlen felelősség elve* szerint hoztuk létre és mint Unity-s komponens használunk, akkor a magas szintű rendszereink tudnak csak az absztrakciótól függeni és Unity-ben intuitívan megvalósulni.

Mindezeket látva a Unity mint keretrendszer rendkívül alkalmas a S.O.L.I.D. elvek alkalmazására.

3.1.4. Tervminták

Miután átnéztük, hogy milyen elvek szerint lett elkészítve a játék, térjünk is át a szakdolgozat lényegére, a felhasznált tervmintákra és azok hasznára szerepére. A Game Programming Patterns-ben[1] olvasott tervmintákat, néhány más hasznos mintával kiegészítve fogom kifejteni itt részletesebben, Jason Weimann[7] és DapperDino[10] implementációit és példáit alapul véve. Ezek a minták kettő csoportba sorolhatók alapvetően, amik az **általunk implementált** tervezési minták és a **Unity/C# által biztosított** tervezési minták, kezdjük is az utóbbikkal.

Game Loop - avagy a *Játékciklus*

Ez a játék magja, legfontosabb alapköve. A Unity automatikusan biztosítja, nincs szükség egy komponáló osztályra, ahol egy while ciklusban hívódnának osztályaink. A tervminta lényege, hogy biztosítanunk kell egy a játék terminállásáig tartó folyamatos környezetet, ahol a játékos inputot, eseményeket és a játékon belüli időt kezelünk kell. Mivel ezt a modern játékfejlesztői környezetek alapértelmezetten tudják és elrejtik a felhasználó elől (mint a Unity), ezért érdemes úgy tekinteni az ezen eszközökkel történő fejlesztés közben, mintha minden a Játékciklusban lennének.

Update Method - avagy a *Frissítési metódus*

Ez a minta mögött az a gondolat húzódik, hogy vannak bizonyos számítások (például egy karakter pozícióváltozásából származó újraraajzolás) amiket minden egyes kirajzolt frame-ben végre szeretnénk hajtani. Unity-ben ezt a tervmintát még jobban felaprították, hiszen a monoBehaviour biztosít számunkra három különböző Update metódust is. A monoBehaviour életciklusa szerint *FixedUpdate* amiben a fizikai számítások hajtódnak végre fix intervallumonként, *Update*, ami egy általános frissítési metódus, itt végezzük a saját logikánk módosításainak nagyját, legvégül a *LateUpdate*, ami egy a sima Update metódust, animáció frisítéseket és a

coroutine-okat követő frissítés, ahol olyan operációkat hajthatunk végre, amiket az Update után szeretnénk végrehajtani.

Component - avagy a *Komponens*

Ez a tervminta egy csoportosítási problémát szándékoz megoldani hasonlóan a Egyetlen felelősség elvéhez. Ezt a Unity szintén alapjaiban támogatja, hiszen az egész játékmotor komponens alapú (rigidbody, amiator, spriteRenderer, mint külön komponensek amiket egy gameObject-re tudunk ráaggatni, mint a saját script-jeinket).

Type Object/Flyweight - avagy a *Típusobjektum/Pehelysúlyú*

Ezen tervezési minták, habár különböző, azonban egymáshoz rendkívüli módon kapcsolódó koncepciókat írnak le. A Típusobjektum lényege, hogy a példányspecifikus adatokat a típusolt objektumban tároljuk, míg az ezeket alkalmazó metódusokat/függvényeket, vagy közös adatokat a típusobjektumban definiáljuk, így a közös részeket és az egyedi adatokat elszeparáljuk. A Pehelysúlyú tervminta nagyon hasonlóan el akarja különíteni a példányspecifikus adatokat a megosztott minden azonos osztály számára szükséges és azonos adatuktól, ezáltal létrehozva egy pehelykönnyű osztályt. Mint láthattuk mindenki tervminta célja a példányspecifikus adatok szeparálása, Unity-ben erre rendkívül jól használhatóak a *ScriptableObject*-ek. Ezek egy különleges osztályok, amik többnyire adatstruktúrákat és közös működéseket írnak le. Miután definiáltunk egy *ScriptableObject*-et, azt legtöbbször nem a tradicionális módon kód ból példányosítjuk, hanem a Unity Editor-ban hozzuk létre, mint új Asset-et, itt tudjuk az általunk megadott mezők értékeit kitölteni. Ha egy a *ScriptableObject*-et hozzárendelünk *GameObject*-ekhez akkor a különböző *GameObject*-ek között a *ScriptableObject* azonos marad, csak egyszer töltődik be a memóriába és minden változást az összes *GameObject* lát ha az adott *ScriptableObject* hozzá van rendelve.

Prototype - avagy a *Prototípus*

A Prototípus minta lényege, hogy elkerüljük a felesleges alosztályok létrehozását azzal, hogy létrehozunk, egy prototípus osztályt, ami klónozással hoz létre új példányokat. Unity-ben ez a **prefab**-ekkel van megoldva. A prefab-ek egy asset-ként elmentett elemek, amit az editoron keresztül tudunk módosítani tetszés szerint, majd példányosítani azt.

Observer - avagy a *Figyelő*

Ez a tervminta annyira elterjedt, hogy a C# nyelvbe natívan támogatva is van az eseményrendszeren keresztül. Unity-ben is támogatva van a C#-os eseményrendszer, azonban a Unity-biztosít egy saját eseménykezelést is ez nem más mint a *Unity event*-ek. Ezek lényegében egy kényelmi funkciót látnak el, hogy egyszerűen tudjunk az Editor-ban összekötni eseményeket, ezen események általában olyan elemek, amit kóddal ritkán váltunk ki (általában UI gombnyomások vagy az Input System-ből érkező események), azonban saját funkciót szeretnénk kötni különösebb komplikáció nélkül.

Object Pool/Object Pooling - avagy az *Objektumkészlet*

Az Objektumkészlet tervezési minta mögötti elv, a memória használat csökkentése és a teljesítmény növelése. Ezt azzal érjük el, hogy az objektumainkat, nem minden újonnan hozzuk létre, hanem egy előre legyártott készletből kérünk egy elemet, és mikor erre az elemre már nincs szükségünk, akkor ez visszakerül a készletbe. Ez egy játékba különösen fontos, hiszen egy objektum létrehozása költséges művelet. Unity-ben, ami ugye egy C# alapú játékmotor, nem is feltétlen a példányosításnál látjuk a problémákat, hanem, mikor a Garbage Collector lefut és felszabadítja a nem használt erőforrásokat. Ilyen esetekben nagy a valószínűsége a frame rate bezuhanásának, ami rossz felhasználói élményhez vezet. Erre tökéletes megoldást jelent az Objektumkészlet, hiszen, ha nincs megszűnt objektum, akkor nincs Garbage Collection sem.

Command - avagy a *Parancs*

A Command pattern lényegében egy callback objektumorientált megfelelője, vagyis használhatunk vele objektumokat arra, hogy más rendszerek/objektumok mit csináljanak. A játékfejlesztésben ez rengeteg helyen hasznosítható. Egyszerűen lehet a parancs tervmintával játékos bemeneteket, kezelní, és akár visszavonni azokat, vagy akár egy játékon belüli boltban a vételel is lehet egy parancs és, ha meggondolunk magunkat, akkor a vételel parancs undo metódusával visszacsinálhatjuk a tranzakciókat, csak hogy egy konkrétabb példával éljek.

Singleton - avagy az *Egyke*

Az egyke tervezési minta, egy egyszerű elven alapszik, biztosítuk, hogy egy osztálynak, csak egy globálisan elérhető példánya legyen. Ez bizonyos

rendszerenkél, például mentési rendszer, scene menedzser rendszer, hangkezelő rendszer. Ezen rendszerekben közös, hogy ha több lenne belőlük, akkor az problémákhoz vezethet (csak egy példány legyen), elérni nem feltétlenül egy helyen szeretnénk őket (legyenek globálisok), szóval láthatjuk miért hasznos ilyen esetekben ha ezen osztályaink/rendszereink egykék lennének. Azonban a singleton tervminta túlhasználtsága, architekturális problémákhoz vezethet. Hiszen a bonyolultabb tervezést megkerülve könnyen elérhető, azonban globális struktúrákat definiálhatunk, amik problémákhoz vezethet, például ha publikusan elérhető akárki által változtatható adatokat tárolunk egy egyke osztályban, az könnyen látható, hogy problémákhoz vezethet. Azonban talán az egyik legnagyobb hátrányuk az egyke osztályoknak, hogy egységesítelhetőségük nehézkes, különösen Unity-ben ahol ezen osztályok gyakran MonoBehaviour-ök is, amik tesztelése rendkívül körülményes.

Humble Object - avagy a *Szerény Objektum*

A Szerény Objektum, Unity-ben egy kimagaslóan hasznos tervezési minta, hiszen az a lényege, hogy egy bonyolultabb objektumból, ami több nehezen tesztelhető függőséggel rendelkezik (például a MonoBehaviour, aminek tesztelése sok felesleges előkészüettel jár), kiemeljük a tesztelni kívánt logikát, olyan szinten, hogy az így keletkezett Szerény Objektumot, már ne kelljen tesztelni. Az így kapott logikai osztály már csak minimális függőségekkel kell rendelkeznie és már (egység) tesztelni egyszerű.

Factory - avagy a *Gyártó*

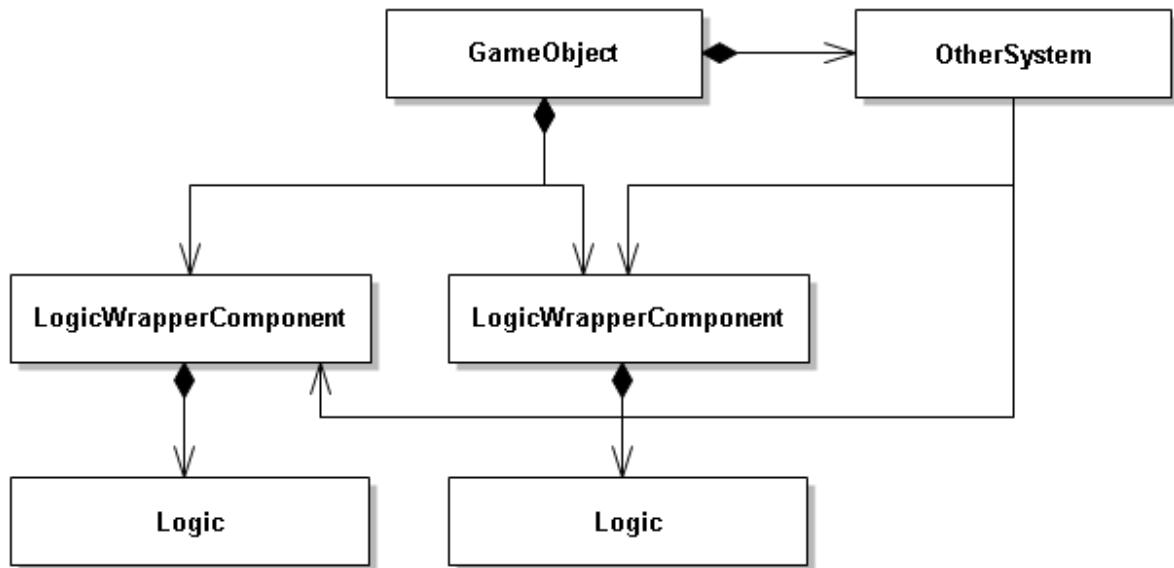
A gyártó minta célja, hogy átadjuk az példánykészítés folyamatát egy erre speciálisan kitalált osztálynak. Ennek az az értelme, hogy nem keletkeznek elszórt példányosítások, amiknek a módosítása egyesével bonyodalmas és problémákat vethet fel. Ehelyett a példályaink egy helyről származnak, ha valahogy módosítani szeretnénk létrejöttjük, azt egy helyen meg tudjuk tenni. Ennek a mintának az előnye a játékfejlesztésben nagyon hamar elő tud jönni, hiszen a játékokban, gyakran vannak különböző tárgyak, varázslatok, effektusok, akár ellenségek, amiket dinamikusan akarunk létrehozni és egy központi gyártó osztállyal mindezt strukturálatabban el tudjuk érni.

State/State machine - avagy a *Állapot/Állapotgép*

Ennek a mintának a játékfejlesztésre vetített lényege, hogy a játék, ne kerülhessen olyan állapotba, amibe nem szabadna, neki. Az állapot tervezési minta, megfogalmaz a nevéből eredően állapotokat, a köztük lévő átmentetek és azok feltételeit, valamint az egész rendszert összekötő állapotgépet. Ez több szempontból is hasznos, egy játékos karaktere komplex viselkedéssel bír, éppen ezért fontos, hogy ne lehessen kódolási hibából nem megengedett állapotba lépni (például, ha ugrani csak akkor lehet, ha a földön vagyunk, akkor ne lehessen a levegőbe ugrani). A játékoskarakter állapotainak és a köztük lévő kapcsolatok leírásával, tehát egy álltopgéppel tiszta architektúrával le tudjuk írni, elkerülve a végtelenségig nyúló if-else és switch elágazásokat. Egy karakterkontrolleren kívül azonban használható önállóbb rendszerekre is az állapotgép, például ellenségek/játékon belüli objektumok viselkedésének (pszeudo AI) leírására is. A Unity-ben az animációk kapcsolata, egy grafikus állapotgépen keresztül van definiálva.

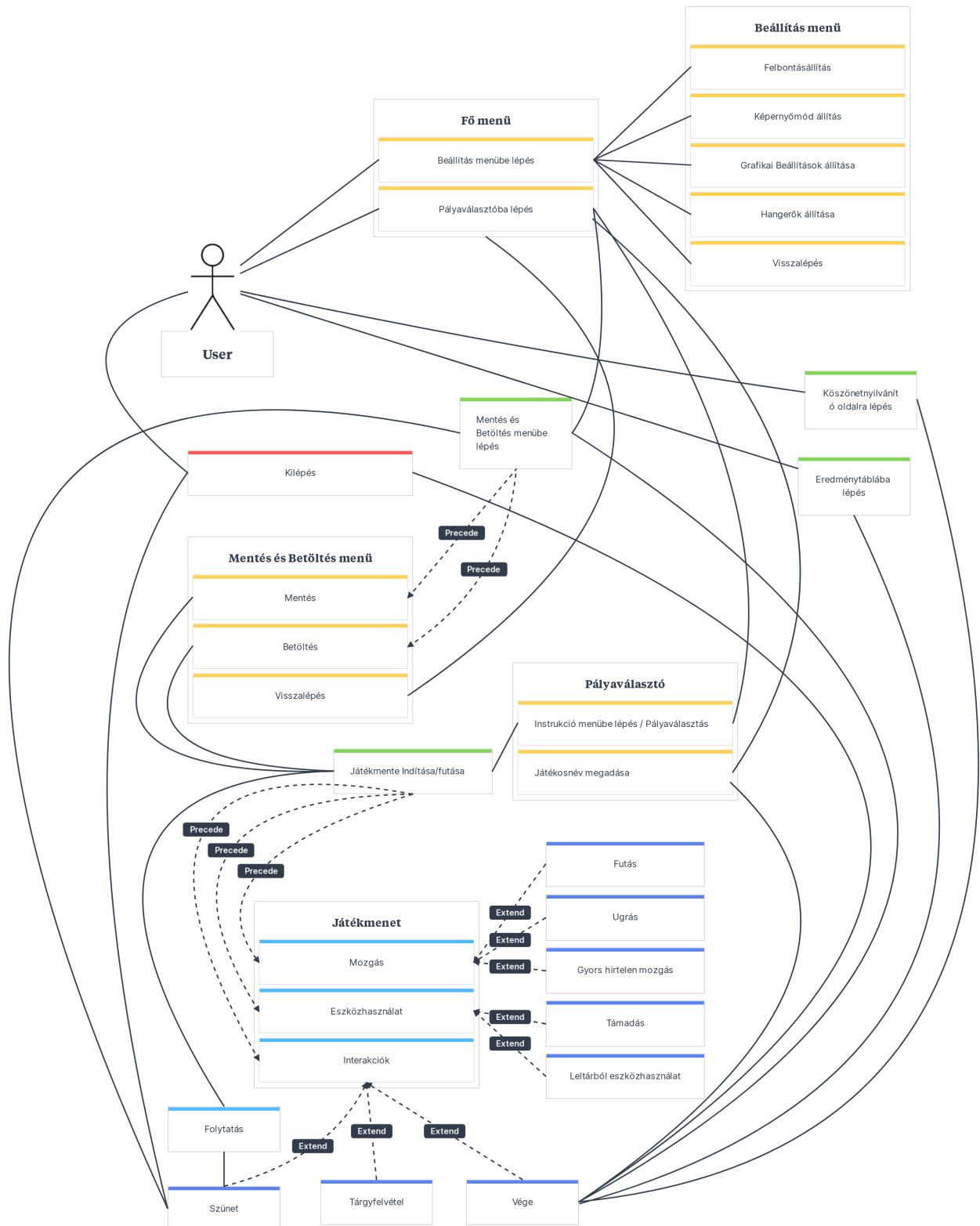
3.2. Megvalósítás

A megvalósítás során látható, hogy alkalmaztam a Szerény objektum tervezési mintát a legtöbb helyen. A logikai részeket (move, attack, inventory) kiemeltem egy osztályba, amiket egy Unity-s MonoBehaviour osztály csomagol be, hogy az Editor-ban ezeket a komponenseket, csak rá kelljen "húzni" arra az objektumra, aminek szüksége van rá (természetesen, ahol lehetett az osztályok az interfészeken keresztül hivatkoznak egymásra). Az így készült komponens alapú megközelítés nagy személyre szabhatóságot biztosít, hiszen ha készítünk, teszem azt egy új bolt komponensem, ahol lehet vásárolnunk, akkor csak rá kell húznunk arra az objektumra amelyiken szeretnénk, hogy elérhető legyen ez a funkció, az interfészeken keresztül történő elérés, pedig biztosítja, hogy a különböző megvalósítások nem okoznak problémát.



3.5. ábra. Általános komponens kapcsolat

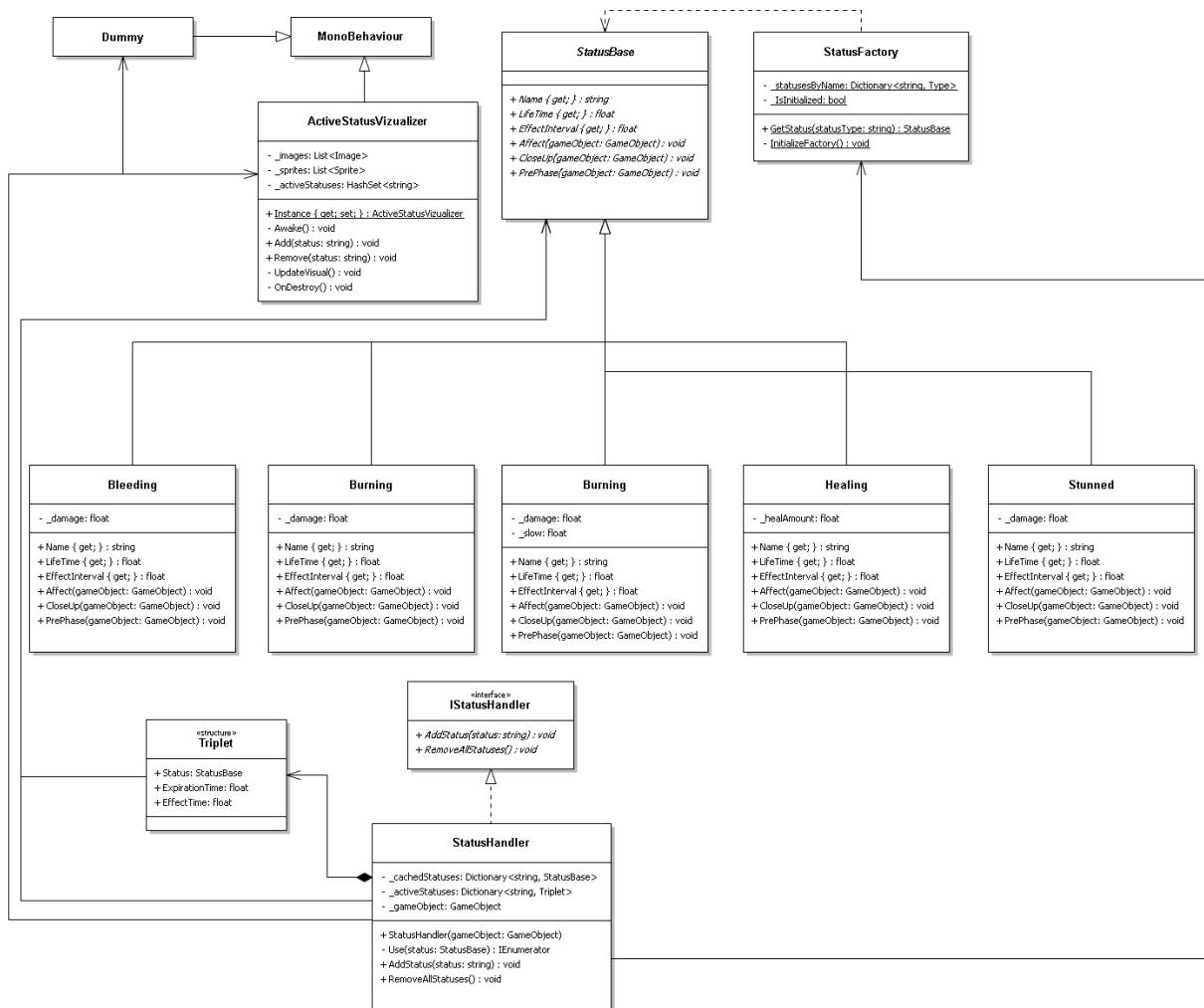
3.2.1. Felhasználói esetdiagram



3.6. ábra. Felhasználói eset diagram

3.2.2. Státusz rendszer

A státusz rendszer a második legbonyolultabb rendszer a játékos rendszer után. A lényege, hogy a *StatusHandler* osztály lekezeli az aktív státuszokat, az újakat el cache-eli, hogy ne kelljen a költséges statikus gyártóosztályt hívogatni potenciálisan frame-enként. A *StatusFactory* osztály reflektióval megtalálja az összes *StatusBase* leszármazottat, hogy abból példányt tudjon szolgáltatni, a *StatusFactory* számára. A *StatusBase* minden specifikus státusz ősosztálya. Egy-egy specifikus osztály merőben különböző hatásokat válthat ki, azonban mindegyik fog hatni valamilyen módon bizonyos időintervallumonként. Lehet előhatása, ahol eltárolhat szükséges adatokat és módosíthat más rendszerekben paramétereket, lehet utóhatása, ahol visszaállíthatja az elmentett állapotot, jelezve a státusz hatásának végét. A különböző funkciókhoz szükséges függőséginjektálást a *GetComponent* függvényel szerzi meg. Egy *StatusHandler*-t a *StatusHandlerComponent* burkolja, amit hozzátudunk adni objektumokhoz, ha szeretnénk, hogy a státuszok hassanak rájuk.

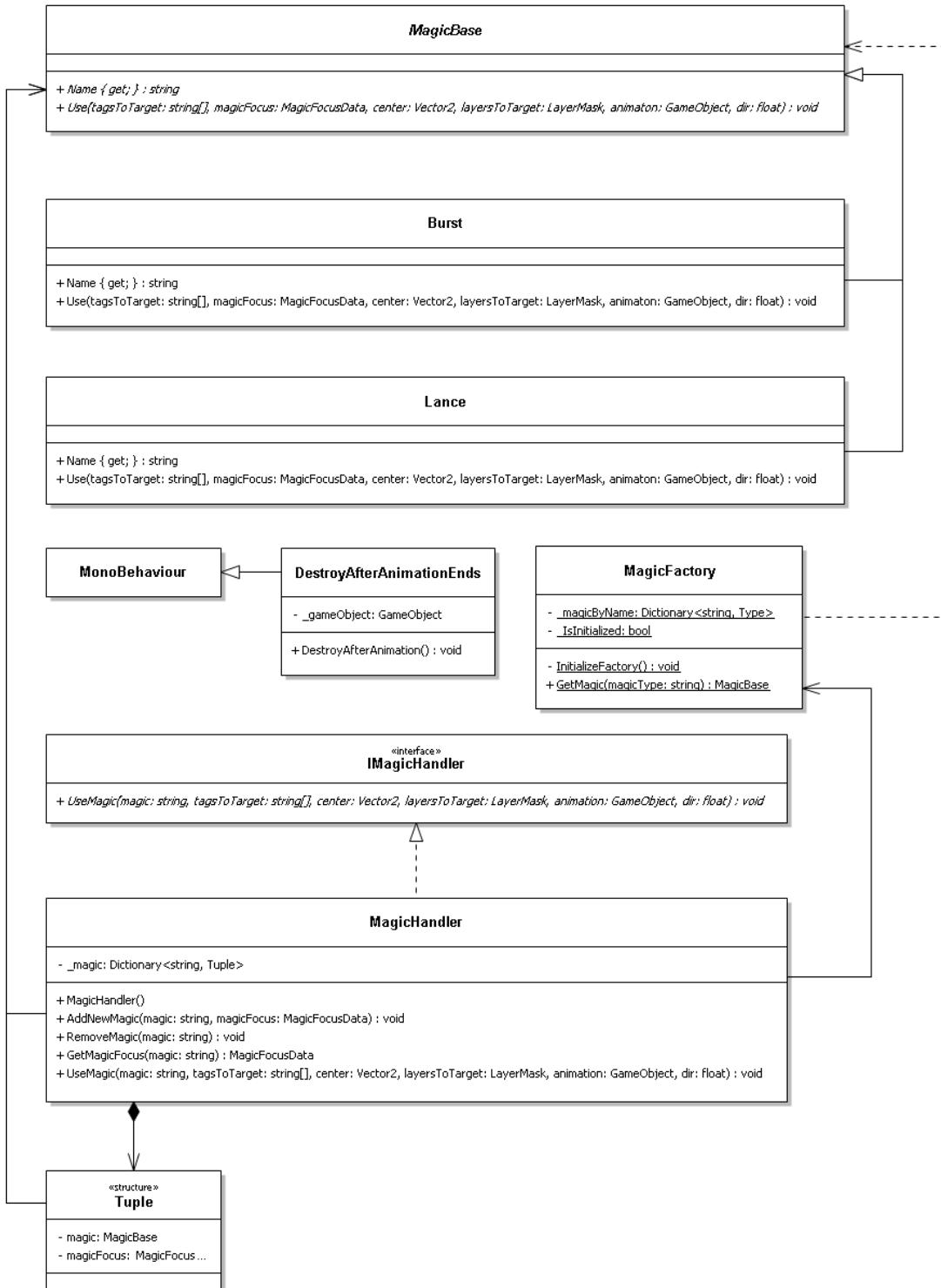


3.7. ábra. Státusz rendszer UML

3.2.3. Varázslás rendszer

A varázslatok rendszere strukturálisan, nagyban hasonlít a státuszok rendszerére.

A nagy különbség, hogy a specifikus varázsaltok, nem automatikusan, hanem egy támadás keretén belül vannak aktiválva és a *Use* metódusuk sokkal komplexebb logikát tartalmaz, mint egy státusz. *MagicHandler* itt is eltárolja a használt varázslatot. A specifikus varázslatok használatát a *UseMagic* metóduson érhetjük el. Itt is mint a státusz rendszernél, egy komponensbe van beburkolva a *MagicHandler* és ezt a komponensem azokra az osztályokra rakjuk rá, amiket szeretnénk, hogy varázslatot tudjon használni, fontos, hogy nem azokra, amikre szeretnénk, hogy egy varázslat hasson.

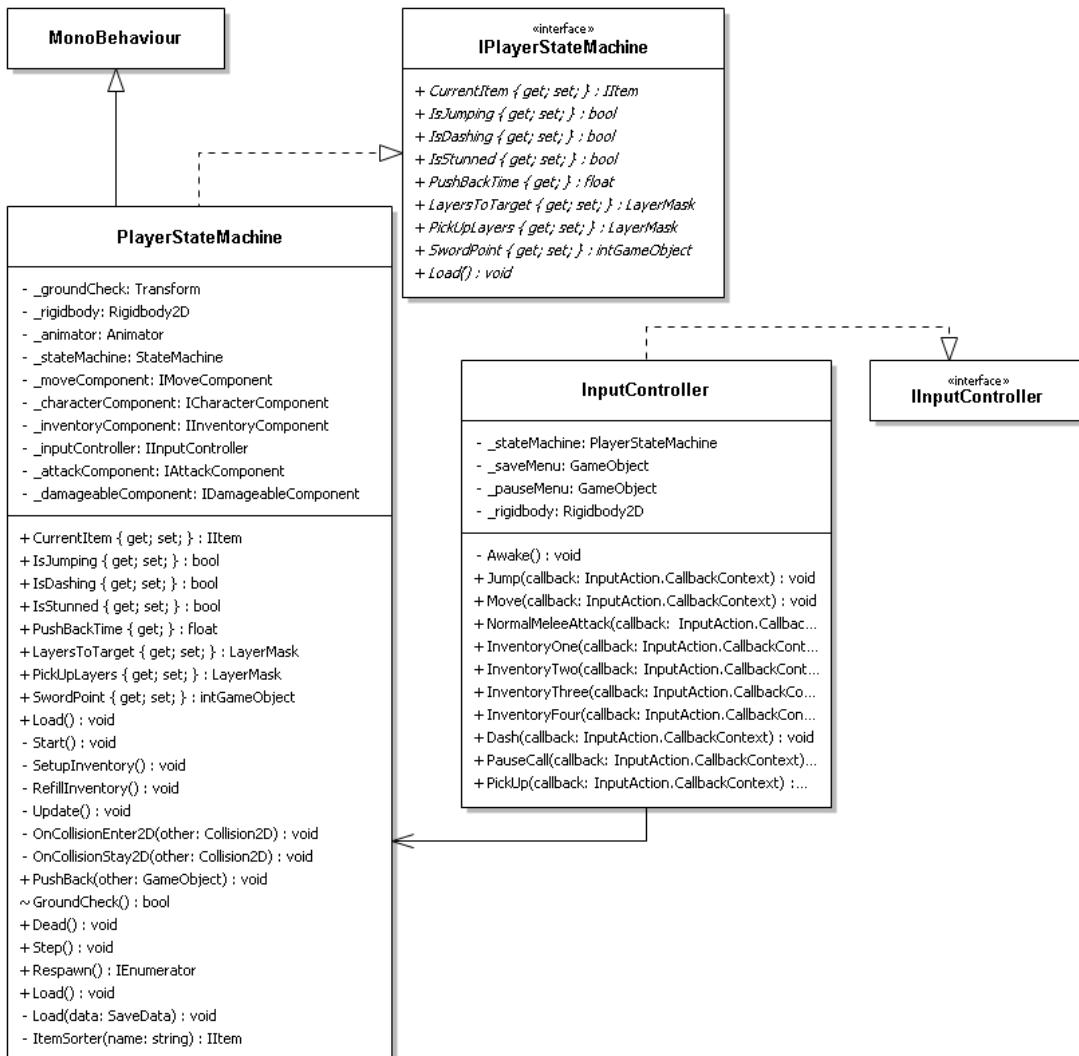


3.8. ábra. Varázslat rendszer UML

3.2.4. Játékos rendszer

A játékos rendszer a legbonyolultabb rendszer a szakdolgozatomban. *PlayerStateMachine* osztályban vannak definiálva a játékoskarakter fontos

metódusai (Respawn, Dead, Load), valamint itt helyezkedik el maga az állapotgép is. Itt található az állapotok közti átmentek és azok feltételei is, valamint az állapotokhoz szükséges dependenciák. Az *InputController* osztályban találhatók azok a metódusok, amiket a Játékos bemenetekből érkező események hívnak, valamint ezek időfeltétele. Ez az osztály felel az állapotváltozók beállításáért, amiken az állapotok átmenetfeltételei alapszanak.



3.9. ábra. Játékos főosztály UML

Azonban ennek a rendszernek a legfontosabb eleme az állapotok. Ezek nélkül nem lehetne a játékoskaraktert irányítani.

Az alapértelmezett állapot az **Idle** állapot, ez egy "üres" helyzet, amikor a játékos

egy helybe áll.

A **Move** állapot lehetővé teszi a jobb és a bal irányú mozgást, valamint a "dash" műveletet és elindítja a futásanimációt.

A **Jump** állapot lehetővé teszi, hogy ugorunk és közben horizontálisan mozogunk, vagy dash-eljünk, valamint elindítja az ugrás animációt.

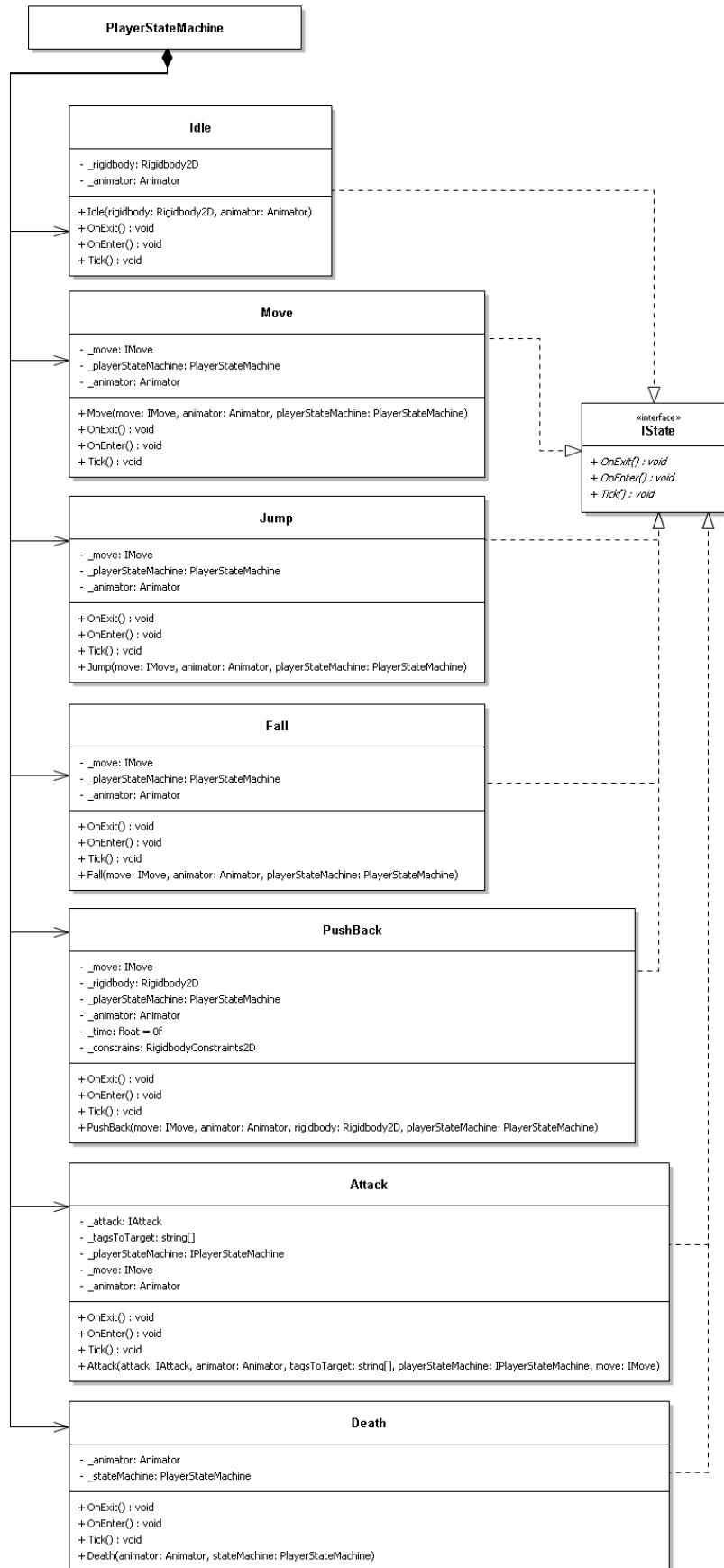
A **Fall** állapotba akkor érkezünk, ha ugrás után lefele ívelő lesz a mozgásunk, vagyis akkor, ha elfogy az ugrásból érkező lendületünk. Miután ebbe az állapotba érünk a normális gravitációnál egy kicsit gyorsabban kezdünk zuhanni, ezáltal a már más játékokból megszokott hatást keltve. Mindezen kívül az állapot elkezdi játszani a zuhanás animációt.

A **PushBack** állapotba akkor érkezünk, ha egy ellenségbe ütközünk, ekkor hátralök minket a játék, és lejátssza a hátralökődés animációt.

A **Attack** állapot lehetővé teszi, az aktuális felhasználói bemenettől függően, az eszközeink használatát és a hozzájuk illő animáció lejátszását.

A **Death** állapot megkezdi a játékos újraéledési procedúráját és lejátssza a halott animációt.

Mint látható az összes a menürendszeren kívüli felhasználói interakció a játékoskarakter állapotgépen keresztül történik, aminek a belépőpontja a *PlayerStateMachine* osztály, aminek komplexitása emiatt érthető. Az is látszik, hogyha nem egy állapotgép lenne a karakterkontroller, hanem egy egyszerű osztály, akkor annak komplexitása és átláthatósága is megnőne, hiszen egyszerre kéne ellátnia az állapotok és az azokat menedzselő osztály feladatát.

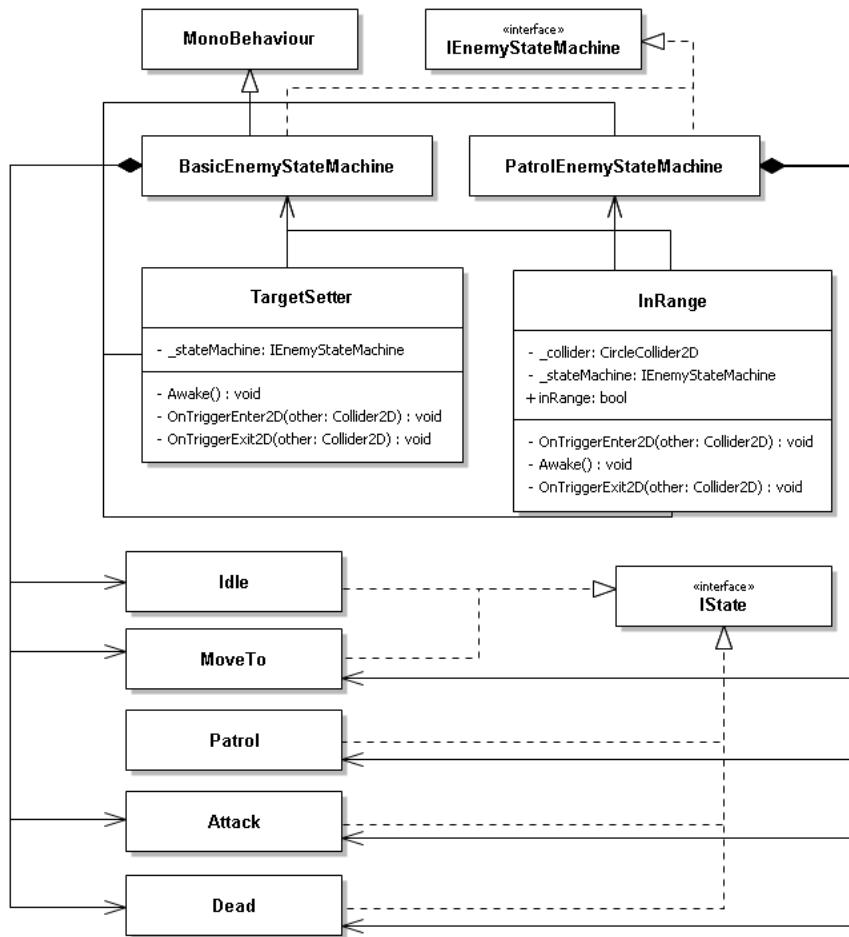


3.10. ábra. Játékos állapotok UML

3.2.5. Ellenség rendszer

Az ellenségek rendszere a játékos rendszerének egy leegyszerűsítése. A játékos rendszerből a felhasználói bemenet kivételével és a játékoskarakterre specifikus metódusok eltávolításával és egy kevés ellenség specifikus kóddal meg is kapjuk, az ellenség rendszert. A legnagyobb kettő különbség, az ellenség specifikus állapotok és a játékos megtalálására/érékelésére felállított rendszer.

A *TargetSetter* osztály felel azért, hogyha a játékos látótávolságon belül van egy ellenséghoz képest, akkor az az ellenség erről tudjon és a megfelelő állapotba lépjen. A *InRange* osztály felel azért, hogy ha a játékos elég közel van egy ellenséghoz, akkor az ellenség a támadás állapotába (Attack) lépjen.



3.11. ábra. Ellenség rendszer UML

A **MoveTo** állapotban kezd el az ellenség a játékos felé mozogni. A **Patrol** állapotban a *PatrolEnemy* elkezd kettő pont között mozogni az **Idle** állapot helyett.

3.2.6. Adattárolás

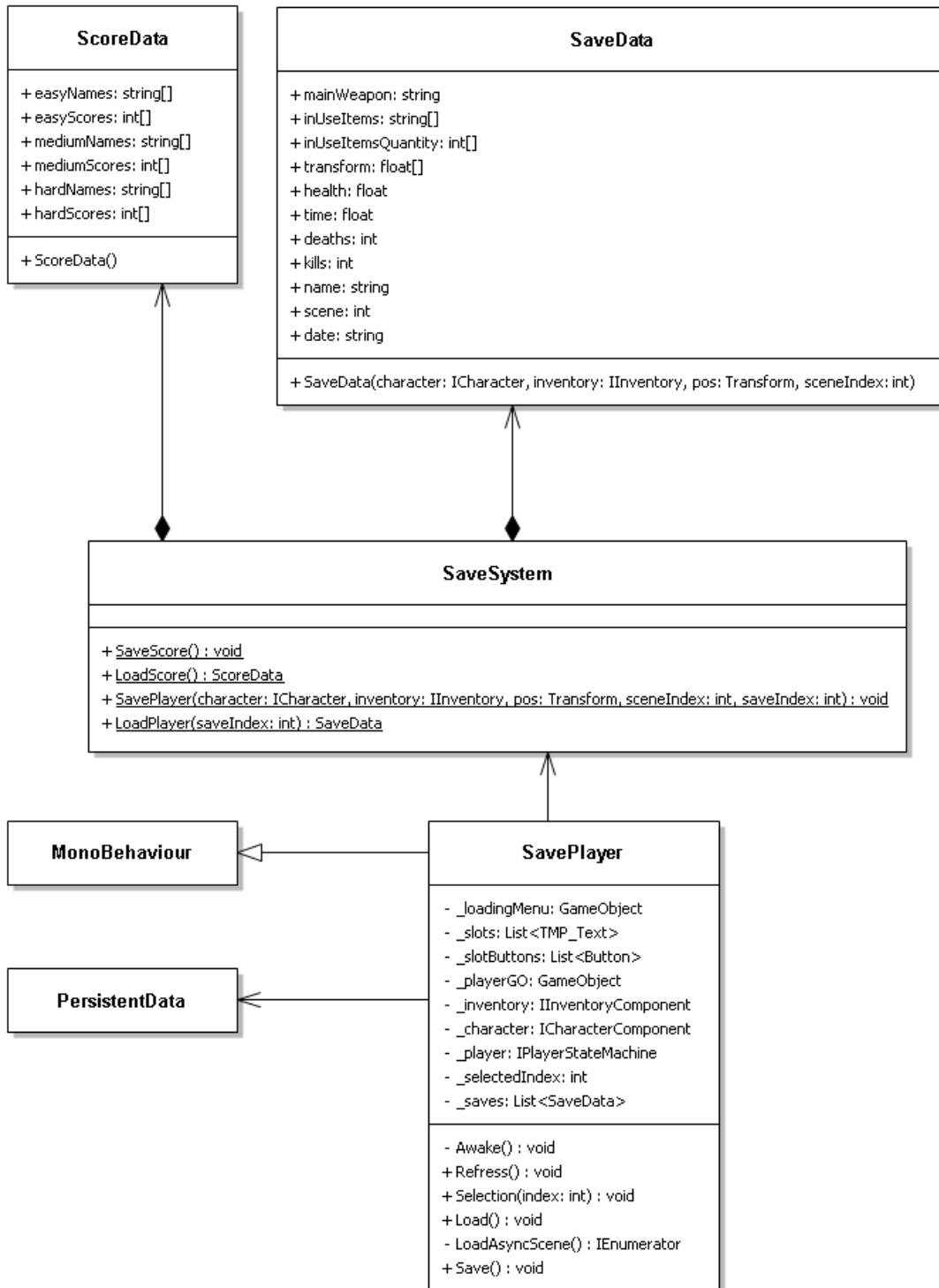
Az adatok tárolására kettő megoldást használtam. Az egyszerűbb adatokra, mint például a játékos neve és a beállítások, a Unity által biztosított PlayerPrefs-et használtam, ami a registry-be menti az adatokat. A bonyolultabb és nagyobb adatok mentésére (eredménytábla, játékosmentés) egy binaryFormatter-t használtam, ami a *SaveSystem* osztályban található, ehhez a menteni kívánt adatokra létrehoztam szerializált osztályokat (*ScoreData*, *SaveData*).

Windows-on a `|AppDat|LocalLow|HMF|The Quest for the Thesis` elérési úton találhatóak a mentések,

Linux-on a `/.config/unity3d` vagy a `/unity3d` elérési úton.

A *SavePlayer* osztály felelős a mentési rendszert összekötni a játék többi rendszerével (GUI, játékoskarakter).

A *PersistentData* osztály felelős a betöltött adatok átjuttatásához különböző scene-ek között. Ez az osztály egy egyke osztály, hiszen olyan adatokat tartalmaz, amiket globálisan elszeretnénk érni, azonban ha több példánya lenne, akkor nem lehetne követni, hogy melyikek a nekünk megfelelő betöltött adatok, ezeket látva döntöttem az egykesége felől.



3.12. ábra. Mentés rendszer UML

3.2.7. CI/CD pipeline

A szakdolgozatom verziókezelésére github-ot használtam. A github actions segítségével építettem ki a CI/CD folyamatomat. Először is, ha a fejlesztői ágra beérkezik egy új commit, vagy egy új pull request, akkor lefut a teszt folyamat. Ez a folyamat végrehajtja az egységeszteket, majd ha sikereske voltak a tesztek készít egy code coverage jelentést.

Logic	184	0	184	383	100%	
HMF.Thesis.Logic.AttackLogic	8	0	8	20	100%	
HMF.Thesis.Logic.CharacterLogic	16	0	16	46	100%	
HMF.Thesis.Logic.DamageableCharacterLogic	10	0	10	34	100%	
HMF.Thesis.Logic.HealableLogic	7	0	7	20	100%	
HMF.Thesis.Logic.InventoryLogic	87	0	87	134	100%	
HMF.Thesis.Logic.MoveLogic	56	0	56	129	100%	

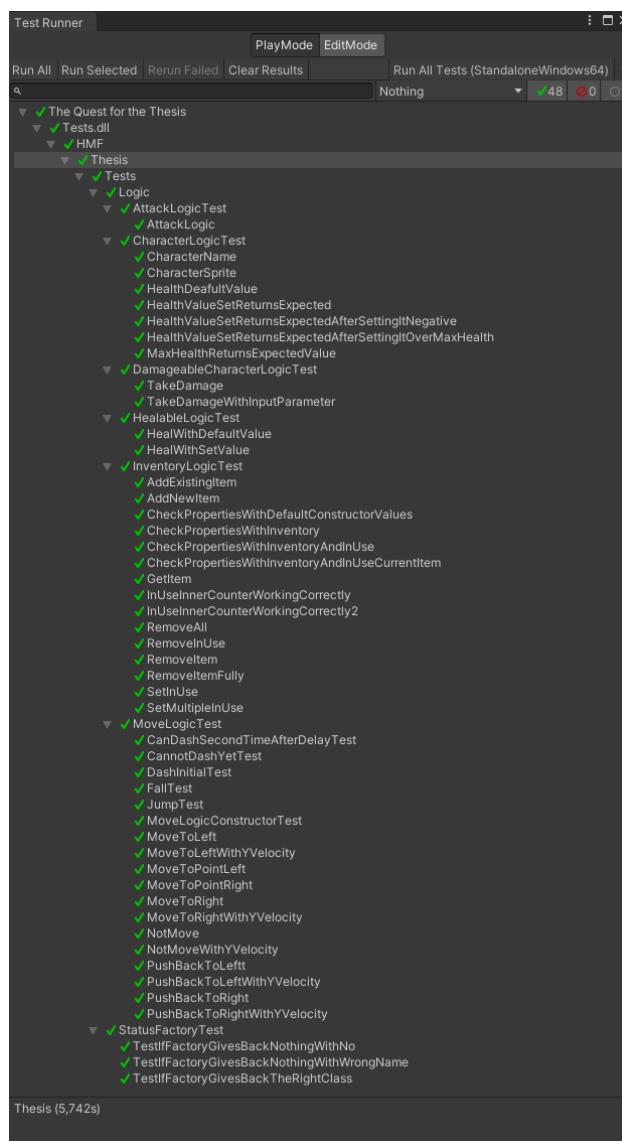
3.13. ábra. A Logic assembly tesztelési lefedettsége

Ha a main ágra érkezik egy új pull request, akkor ugyanúgy lefut a tesztelési folyamat és a code coverage jelentés generálása, azonban ha sikeres volt a teszt, akkor megkezdődik a build folyamat. Ez elkészíti a kívánt (előre beégettet) platformra a lefordított, futtatható játékot. Mindez után, ha sikeres volt a build folyamat, akkor elkészíti az automatikus dokumentációt és kitelepíti azt a <https://thesis-docs.netlify.app> címre.

3.3. Tesztelés

3.3.1. Egységesztések

Az egységesztést white box módon hajtottam végre. A Unity biztosít kettőféle tesztelési környezetet, Editor mode és Play mode. Én a Play mode-ot használtam, ami a játékban futtatja a teszteket, nem az editorban. A Szerény objektum (Humble Object) segítségével kiemelt logikai osztályaimat teszteltem. A teszteléshez NUnit keretrendszert használtam, mock-oláshoz pedig a Moq egy Unity kompatibilis verzióját.



3.14. ábra. Az egységesztések eredményei

3.3.2. Kézi Tesztelés

Helyzet	Elvárás	Státusz
Játék indítása	Játék elindul	Sikeres
Főmenüben Scoreboard gombra kattintás	Megjelenik az eredménytábla	Sikeres
Eredménytáblánál a Back gombra kattintás	Visszatérünk a főmenübe	Sikeres
Főmenüben Scoreboard gombra kattintás	Megjelenik a köszönetnyilvánítás	Sikeres
Köszönetnyilvánításnál Back gombra kattintás	Visszatérünk a főmenübe	Sikeres
Főmenüben Exit gombra kattintás	Játék bezáródik	Sikeres
Főmenüben Settings gombra kattintás	Megjelennek a beállítások	Sikeres
Beállítások változtatása	Azonnali beállításváltás	Sikeres
Beállításoknál Back gombra kattintás	Visszatérünk a főmenübe	Sikeres
Főmenüben Settings gombra kattintás újra	Megjelennek a megváltoztatott beállítások	Sikeres
Főmenüben Start gombra kattintás	Megjelenik a pályaválasztó	Sikeres
Pályaválasztóban Name mezőbe írás	Megváltozik a nevünk	Sikeres
Pályaválasztóban Load gombra kattintás	Megjelennek a mentések	Sikeres
Mentésekkel egy mentett állapotra kattintás	Kijelöljük az állapotot	Sikeres
Pályaválasztóban Load gombra kattintás	Betölt a mentet állás az instrukció menüvel	Sikeres

Pályaválasztóban Back gombra kattintás	Visszatérünk a fő menübe	Sikeres
Pályaválasztóban az egyik pályára gombra kattintás	Betölt a pálya az instrukciómenüvel	Sikeres
Instrukciómenüben a Play gombra kattintás	Elkezdődik a játék	Sikeres
A játékban ESC megnyomása	Megjelenik a szünet menü	Sikeres
Szünetmenüben Resume gomb megnyomása	Játék folytatása	Sikeres
Szünetmenüben Save/Load gomb megnyomása	Megjelennek a mentések	Sikeres
Mentésmenüben egy hely kiválasztása után Save gomb megnyomása	A jelenlegi állás elmentődik	Sikeres
A játékban A/D megnyomása	Mozgás jobbra/balra	Sikeres
A játékban SPACE megnyomása	Ugrás	Sikeres
A játékban Bal ALT megnyomása	Gyors hirtelen mozgás	Sikeres
A játékban bal egérgomb megnyomása	Sima támadás	Sikeres
A játékban 1,2,3,4 megnyomása	Megfelelő eszköz használata, cooldown megjelenítése	Sikeres
A játékban ellenség közelébe menés	Ellenség jön felénk és támad	Sikeres
A játékban nullára megy az életünk	Újraéledés	Sikeres
A játékban ellenséget sokszor támadjuk	Ellenség meghal	Sikeres
A játékban ellenség stáruszefektust rak ránk	Megjelenik a stáruszefektus bal felül	Sikeres
A játékban stáruszefektus rajtunk van	Hat a stárusz	Sikeres

A játékban elmúlik a státuszeffektus	Eltűnik felülről a státusz	Sikeres
A játékban elérünk a pálya végére	Megjelenik a vége menü	Sikeres
A vége menüben	Megjelenik a pontunk és a többi pálya amire léphetünk	Sikeres

3.3.3. Tesztelési konklúzió

Az egységesztek miatt ha módosítani is kellett a logikai osztályokat, akkor is az elvárt viselkedést mutatták, vagy ha nem akkor az hamar és pontosan kiderült, ezzel időt spórolva a hibakeresés során. Az egységesztek írása során továbbá sikerült olyan hibákat találni, amikre, ha manuális tesztelés során találtam volna rá, sokkal több munka árán tudtam volna javítani.

A manuális tesztek segítették a rendszerek közötti kapcsolatok és az általános játékmenet tesztelését és finomítását. Összegezve a tesztek nélkül egyértelműen több időt vett volna igénybe a szakdolgozatom elkészítése, és lényegesen alacsonyabb minőségű szoftver keletkezett volna.

4. fejezet

Összegzés

A szakdolgozatom célja volt, hogy a teljesség igénye nélkül mutassak meg tervezési mintákat a játékfejlesztésben. Néhány mintára példát mutatni és szemléltetni, hogy miért növelik ezeknek a használata a szoftver minőségét és átláthatóságát. Saját meglátásom szerint ezt a célt elértem, mutattam tervezési mintákra felhasználási lehetőséget, kódbeli példákat, és további tervezési mintákat amiket más esetekben fel lehet használni. Végezetül egy olyan játékot készítettem, aminek értelmezés és továbbfejlesztése nem igényel felesleges erőfeszítést.

4.1. Köszönetnyilvánítás

Ezen szekcióba szeretném megköszönni Kovácsné Puszta Kinga Tanárőnek, hogy kérdéseimre, mindig hasznos információkkal szolgált és ezzel megfelelő irányba terelgette szakdolgozatom.

Köszönetemet szeretném nyilvánítani, Andi Petinek, Gyimesi Kristófnak, Szalai Patriknak, Yu Kai Tenak, Jakab Leventének és Orosz Ádámnak, hogy a félév során kipróbtálták a játékomat, és szakmailag hasznos és értelmes visszajelzésekkel láttak el, jelezve nekem, hogy jó úton járok a szakdolgozatom készítése során.

Irodalomjegyzék

- [1] Robert Nystrom. *Game Programming Patterns*. 2014. ISBN: 978-0-9905829-2-2.
URL: gameprogrammingpatterns.com (elérés dátuma 2021. 05. 06.).
- [2] *Unity 2020.2 Documentation*. URL: <https://docs.unity3d.com/2020.2/Documentation/Manual/index.html> (elérés dátuma 2021. 05. 06.).
- [3] *S.O.L.I.D Principles*. URL: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design (elérés dátuma 2021. 05. 06.).
- [4] *Character Controller*. URL: <https://medium.com/ironequal/unity-character-controller-vs-rigidbody-a1e243591483> (elérés dátuma 2021. 05. 06.).
- [5] *Parallax Script*. URL: <https://answers.unity.com/questions/551808/parallax-scrolling-using-orthographic-camera.html> (elérés dátuma 2021. 05. 06.).
- [6] *Palette Swapping with Shaders*. URL: https://www.youtube.com/watch?v=SQjeNhTp_Xg (elérés dátuma 2021. 05. 06.).
- [7] *Jason Weimann*. URL: https://www.youtube.com/channel/UCX_b3NNQN5bzExm-22-NVVg (elérés dátuma 2021. 05. 06.).
- [8] *Brackeys*. URL: https://www.youtube.com/channel/UCYbK_tjZ20rIZFBvU6CCMiA (elérés dátuma 2021. 05. 06.).
- [9] *Code Monkey*. URL: https://www.youtube.com/channel/UCFK6NCbuCIVzA6Yj1G_ZqCg (elérés dátuma 2021. 05. 06.).
- [10] *Dapper Dino*. URL: <https://www.youtube.com/channel/UCjCpZyil4D8TBb5nVTMMaUw> (elérés dátuma 2021. 05. 06.).

- [11] Thomas Brush. URL: <https://www.youtube.com/channel/UCuHVjteDW9tCb8QqMrtGvwQ> (elérés dátuma 2021. 05. 06.).
- [12] ISAo. SOUND AIRYLUVS. URL: <https://airyluvs.com/> (elérés dátuma 2021. 05. 06.).
- [13] TAD. *Samurai*. URL: <https://opengameart.org/content/samurai> (elérés dátuma 2021. 05. 06.).
- [14] Marcelo Fernandez. *Bamboo Forest*. URL: <https://soundcloud.com/marcelofernandezmusic> (elérés dátuma 2021. 05. 06.).
- [15] Johan Brodd. *Roof of the world*. URL: <https://opengameart.org/content/roof-of-the-world> (elérés dátuma 2021. 05. 06.).
- [16] Iwan 'qubodup'Gabovitch. *Screams*. URL: <https://opengameart.org/content/15-vocal-male-strainhurtpainjump-sounds> (elérés dátuma 2021. 05. 06.).
- [17] Artisticdude. *Fire & Evil Spell*. URL: <https://opengameart.org/content/fire-evil-spell> (elérés dátuma 2021. 05. 06.).
- [18] Iwan 'qubodup'Gabovitch. *Ice and Electricity Magic*. URL: <http://opengameart.org/users/qubodup> (elérés dátuma 2021. 05. 06.).
- [19] HaelDB. *Footsteps Leather, Cloth, Armor*. URL: <https://opengameart.org/content/footsteps-leather-cloth-armor> (elérés dátuma 2021. 05. 06.).
- [20] StarNinjas. *20 Sword Sound Effects*. URL: <https://opengameart.org/users/starninjas> (elérés dátuma 2021. 05. 06.).
- [21] OwlshMedia. *RPG UI Icons*. URL: <https://opengameart.org/content/rpg-ui-icons> (elérés dátuma 2021. 05. 06.).
- [22] Hugos's visual designa. *Samurai Assets*. URL: <https://hugoss-visual-design.itch.io/samurai-asset> (elérés dátuma 2021. 05. 06.).
- [23] Flip. *Animated Potion Assets Pack*. URL: <https://flippurgatory.itch.io/animated-potion-assets-pack-free> (elérés dátuma 2021. 05. 06.).
- [24] Ppeldo. *Pixel Art FX*. URL: <https://ppeldo.itch.io/2d-pixel-art-game-spellmagic-fx> (elérés dátuma 2021. 05. 06.).

- [25] NYKNCK. *Pixel Art Effect - FX033*. URL: <https://kvsr.itch.io/pixel-art-effect-fx033> (elérés dátuma 2021. 05. 06.).