Universitatea din București
Facultatea de Matematică și Informatică
Specializarea Informatică, anul II

Proiect pentru laboratorul cursului *Metode de dezvolatre software*

Coordonatori științifici
Conf.dr.Alin ȘTEFĂNESCU

Studenți
Șerban-Corneliu Mateescu
Vlad Mihai Panait
Vlad-Vasile Chichirău
Grupa 235

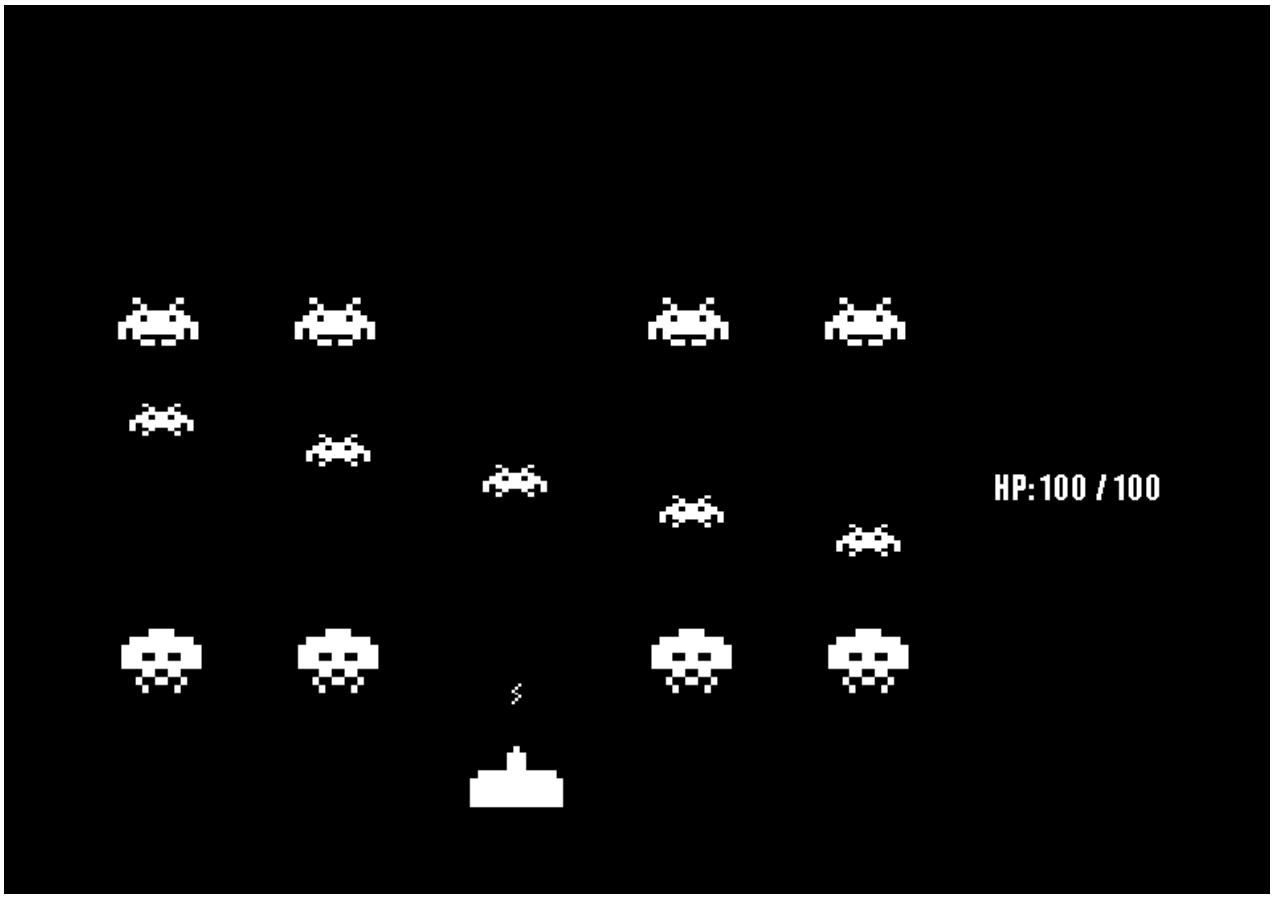
București, 2016

1. Introducere

Clasicul joc “Space Invaders”, in care playerul controleaza o nava ce poate lansa proiectile tintite catre navele extraterestre ce se apropie amenintator, este unul dintre cele mai vechi exemple de joc bazate pe un concept simplist, dar placut. Competitive Space Invaders, proiectul in discutie, este bazat pe acelasi principiu, dar cu unele modificari.

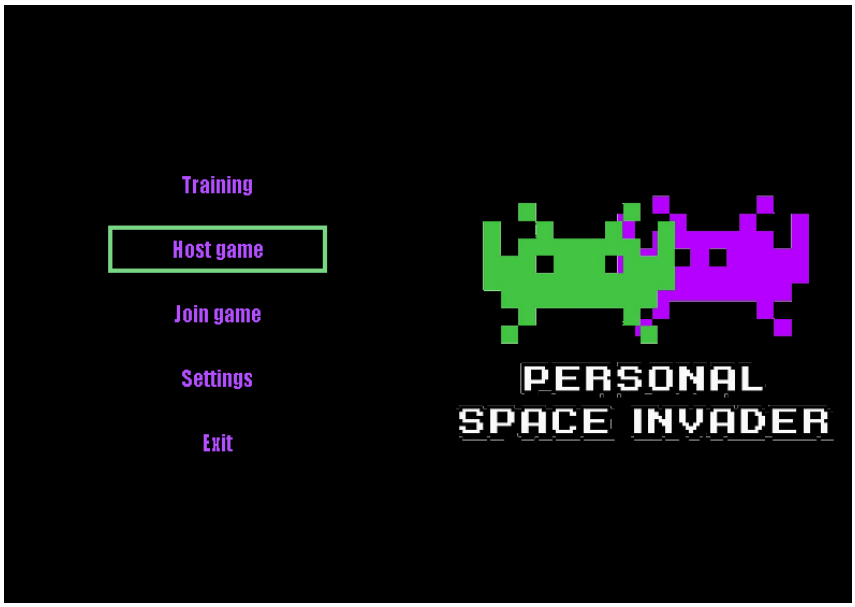
Jocul este construit in jurul ideii de multiplayer competitiv (LAN). Partidele constau in infruntarea dintre doi jucatori, fiecare alegand la inceputul rundelor cu ce wave-uri de nave extraterestre se va confrunta oponentul. La sfarsitul fiecarei runde, daca ambii jucatori au supravietuit, jocul continua pana in runda cu numarul 10 (remiza). Daca, insa, unul dintre jucatori pierde, castigator va fi cel care a supravietuit. Exista, totusi, posibilitatea ca amandoi sa piarda in aceeasi runda, caz in care rezultatul partidei va fi remiza.

Utilizatorul are, totodata, optiunea de a alege modul “Training”, in care isi poate selecta ce wave-uri sa infrunte cu scopul de a-si pregati tacticile pentru meciurile 1v1.



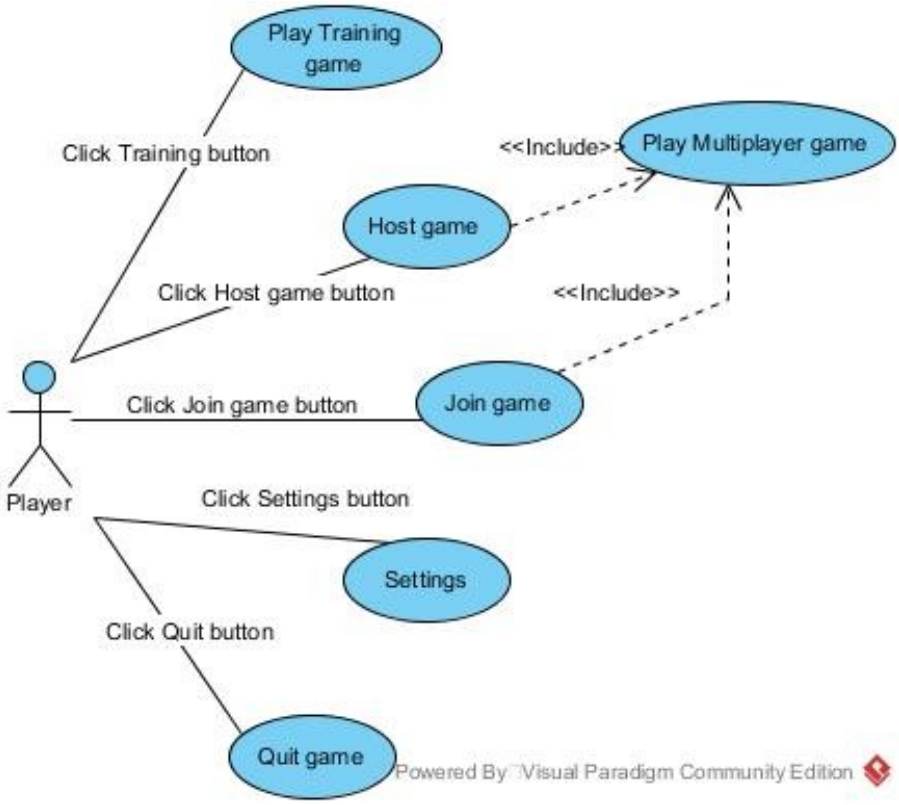
2. Detalii de utilizare

a. Meniul principal



Meniul principal asigura utilizatorului accesul la principalele functionalitati ale soft-ului in discutie. De aici utilizatorul poate opta pentru inceperea unui joc nou, avand optiunea de-a alege intre optiunile: multiplayer sau singleplayer. De asemenea acesta are posibilitatea sa modifice setarile programului sau poate inchide programul.

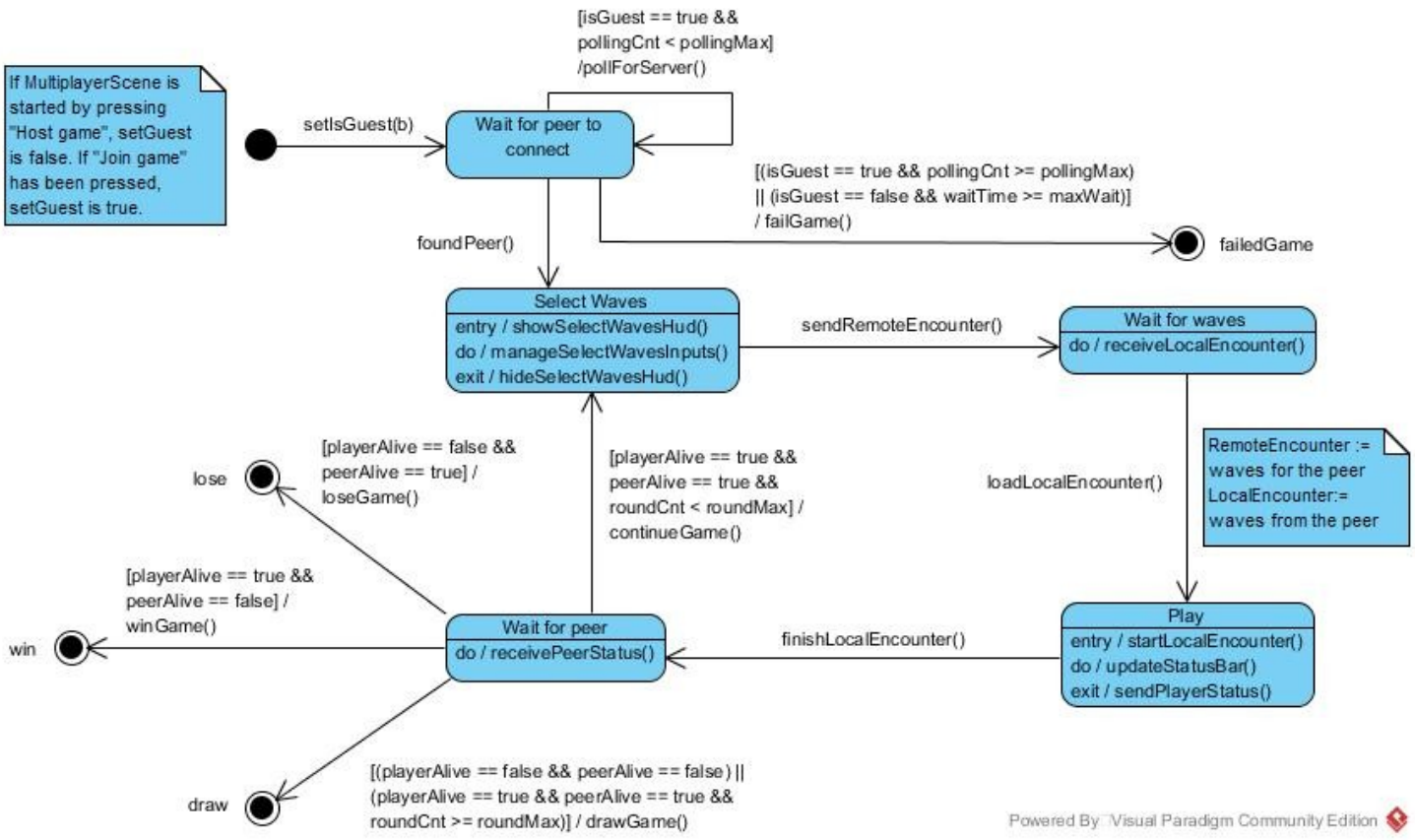
Urmatoarea diagrama ilustreaza mai in detaliu aceste functionalitati:



b. Multiplayer gameplay

Partea principala a jocului o reprezinta infruntarea dintre jucatori prin meciuri 1vs1. Initierea unui joc in retea necesita un host si un guest. Astfel, optiunea **Host game** va pune playerul intr-un stadiu de asteptare, pana cand un altul va selecta **Join game** si va introduce ip-ul host-ului. De mentionat este ca nu conteaza ordinea in care se fac cele doua operatiuni, guest-ul si host-ul putand astepta unul dupa celalalt in cazul in care peer-ul sau nu a selectat optiunea respectiva.

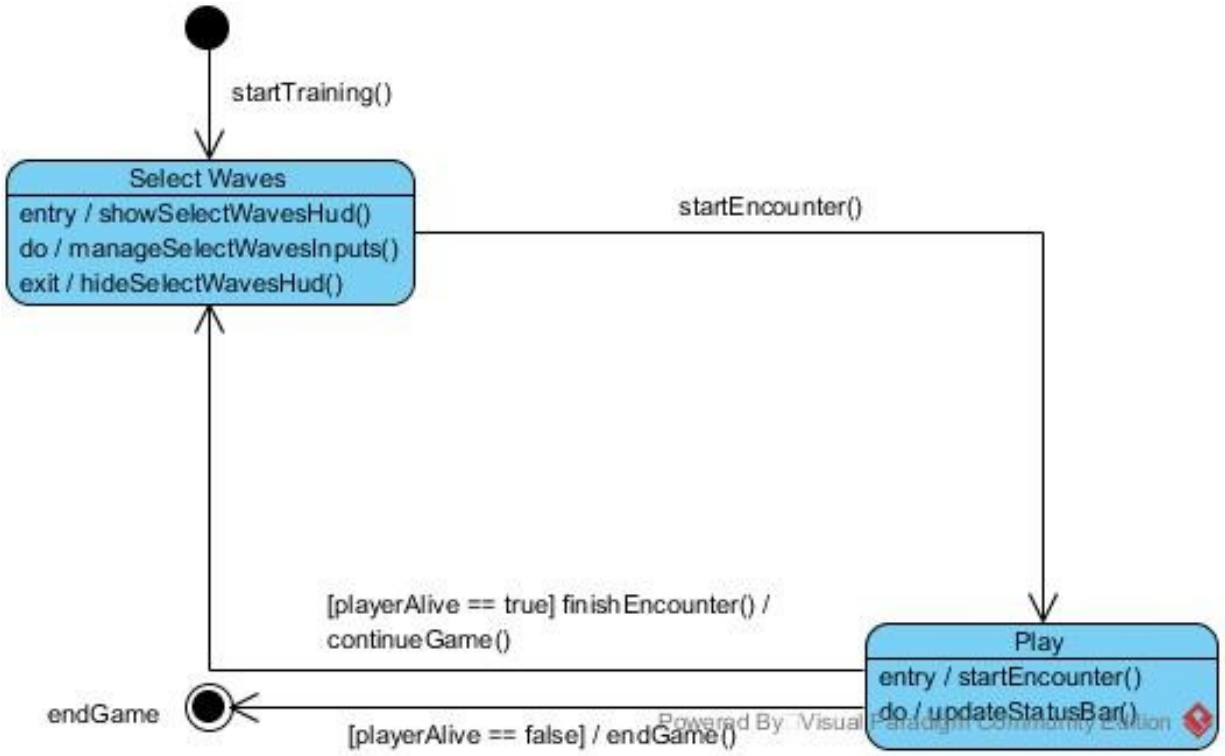
Jocul propriu-zis este impartit pe runde (fiind limitate la 10). O runda este constituita din doua mari faze: selectarea **wave-urilor** pe care oponentul le va infrunta. Cea de-a doua faza este encounter-ul local, adica lupta cu wave-urile selectate de peer. Aceste detalii, incluzand si posibilele outcome-uri ale meciului sunt prezentate in urmatoarea diagrama:



c. Singleplayer gameplay

Competitive Space Invaders dispune si de un mod singleplayer, numit **Training mode**. Scopul acestuia, dupa cum sugereaza si numele, este antrenarea pentru modul multiplayer. Astfel, jucatorul are posibilitatea de a-si construi singur encounter-ul, pentru a experimenta de dinainte diferitele combinatii pe care le-ar putea primi de la oponenti.

Progresia jocului este o versiune simplificata a celei de la modul multiplayer. Jocul este, deci, impartit pe runde cu cele doua faze mai sus mentionate. Diferenta este, evident, data de absenta oponentului. In faza de selectare a wave-urilor, deci, playerul va alege, intr-o maniera identica celuilalt mod de joc, proprii inamici. Diagrama ce urmeaza ilustreaza aceasta versiune simplificata a jocului:



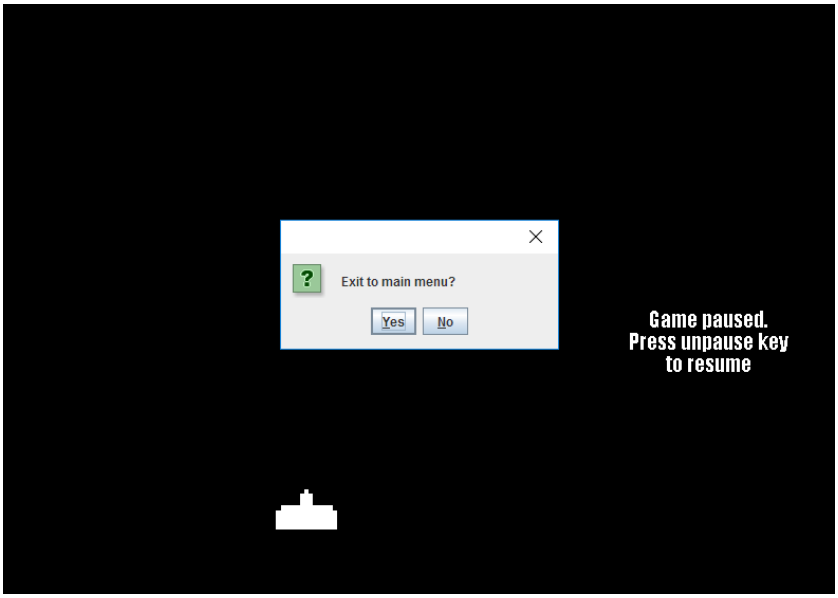
d. Settings

Din ecranul de **Settings** (inca neimplementat), utilizatorul poate ajusta diferite optiuni, cum ar fi tastele de control ale navei proprii, culoarea acesteia, volumul muzicii de fundal si a sunetelor. De asemenea, jucatorul poate seta numarul de runde. Aceasta setare va avea efect in multiplayer numai in cazul in care utilizatorul in discutie va initia jocul prin apasarea butonului **Host game**.

e. Iesire din joc, pauza

Pentru a iesi din joc, pe langa inchiderea efectiva a ferestrei de joc, exista si optiunea de a folosi butonul **Exit** din meniul principal. Daca, in timpul unui joc multiplayer, unul dintre jucatori inchide programul, conexiunea se va reseta pentru celalalt, iar ecranul I se va schimba inapoi in **Main menu**.

Legat de optiunea de a pune pauza in timpul jocului, aceasta este posibila, momentan, doar in modul singleplayer. Pe viitor, aceasta ar putea fi disponibila si in modul multiplayer, dar numai dupa ce va fi implementat un sistem de evitare a abuzurilor (spre exemplu, un jucator pune pauza si asteapta foarte mult, oponentul sau fiind, de asemenea, blocat in tot acest timp).



3. Detalii de implementare

Jocul a fost implementat in Java, fara folosirea vreunui framework third party. Ca si pattern arhitectural, programul a fost construit pe modelul **Entity-Component-System**, o versiune simplificata a acestuia, cel putin. Flexibilitatea acestei paradigme consta in faptul ca orice obiect dintr-o scena (nave, proiectile, obiecte de control, elemente de hud etc.) sunt considerate entitati, functionalitatea lor fiind data de componentele ce le sunt atasate.

Simplificarea pattern-ului consta in omiterea componentei **System** (care se ocupa de logica si behaviour) sau, altfel spus, integrarea acesteia in cadrul componentelor. Astfel, acestea vor contine, pe langa campurile cu informatii despre stare, metode pentru logica si behaviour.

Desi mai putin cache-friendly, aceasta optiune este mai usor de implementat si, prin omiterea unor constrangeri, ofera libertate mai mare de a face prototipuri.

Nota: pentru ca e mai scurt cuvantul, in cod, **Components** sunt **Parts**.

a. SceneManager-Scene-Entity-Part

Clasa ce sta la baza logicii programului este **SceneManager**. Aceasta nu face altceva, decat sa pastreze o **Scena** curenta, careia sa-i apeleze, la fiecare frame, metodele **update()** si **draw()**. De asemenea, se ocupa si de schimbarea scenei curente, moment in care apeleaza **unload()** pe cea veche si **load()** pe cea noua.

Clasa **Scene** contine un array de **Entities** pe care le updateaza si deseneaza la fiecare frame. De asemenea, derivatele acestei clase, cum ar fi **MainMenuScene** sau **TrainingScene**, au si alte elemente de logica, pe langa cele de baza, cum ar fi, spre exemplu, faptul ca pastreaza in campuri speciale entitati importante (**playerShip**, de pilda) sau metode de control ale **State**-ului (acestea au fost descrise amanuntit in sectiunea “Detalii de utilizare”).

Clasa **Entity** este, dupa cum sugereaza si numele, echivalentul elementului cu acelasi nume din patternul **ECS**. Aceasta nu este mostenita de alte clase, customizarea bazei facandu-se prin intermediul array-ului de **Parts**. Acestea, dupa cum era de asteptat, sunt updatate si desenate la fiecare frame.

In cele din urma, clasa **Part** este modalitatea de a diferentia tipurile de entitati intre ele. Exista un numar de clase ce o mostenesc, acestea fiind atasate in momentul construirii (cel mai des, in factory-uri, cum ar fi **EntityFactory**) in functie de rezultatul dorit. Spre exemplu, o nava inamica va fi o entitate ce va contine urmatoarele:

- **TransformPart**: retine detalii despre pozitia, rotatia si scala entitatii
- **HitboxPart**: se ocupa de detectarea coliziunilor cu alte hitbox-uri ale altor entitati
- **SpritePart**: continea imaginea ce va simboliza nava inamica
- **ControllerPart**: componenta foarte importanta care se ocupa de comportamentul entitatii; contine un array de instante ale claselor derivate din **Behaviour** ce descriu un anume comportament

Metodele **update()** si **draw()** au “cel mai vizibil efect” in cadrul acestor clase. Daca pana in acest punct, aceste metode au apelat corespondentul lor din clasele continute, acum (cu exceptia **ControllerPart**) ele fac posibile pentru entitati miscarea, interactionarea cu alte entitati, vizibilitatea in scena etc.

Diagrama ce urmeaza ilustreaza relatia dintre clasele ilustrate mai sus:

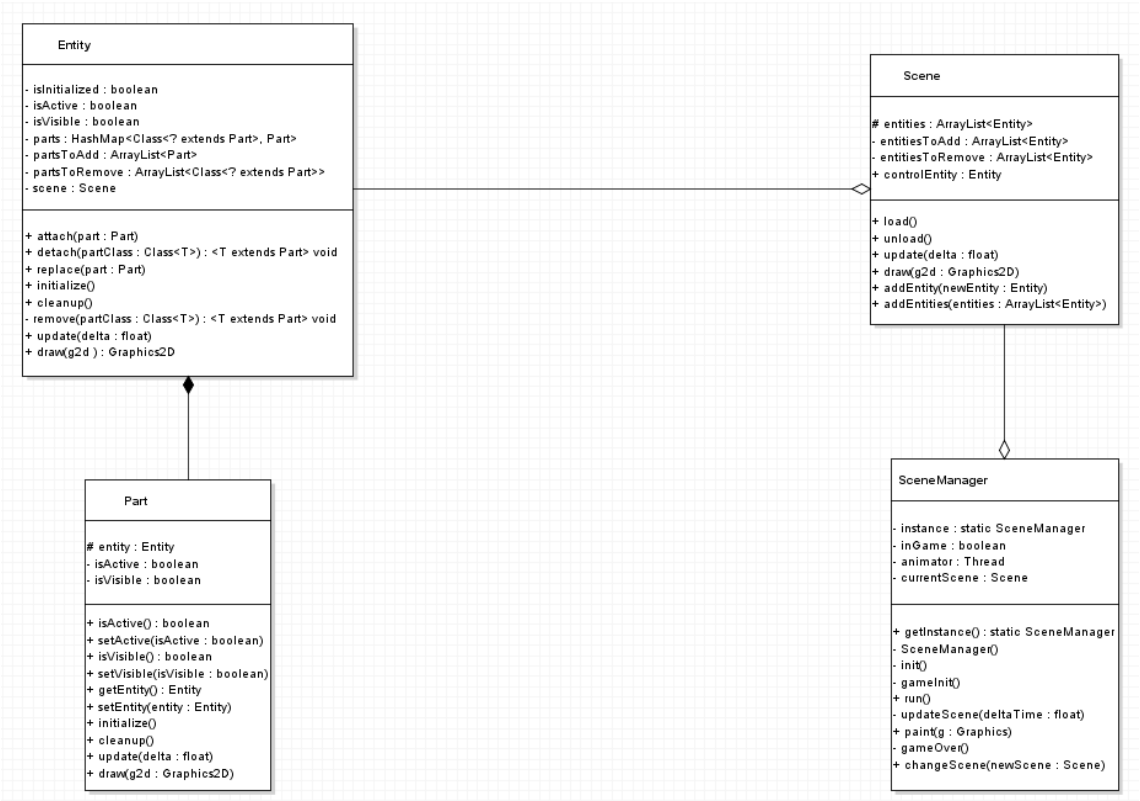
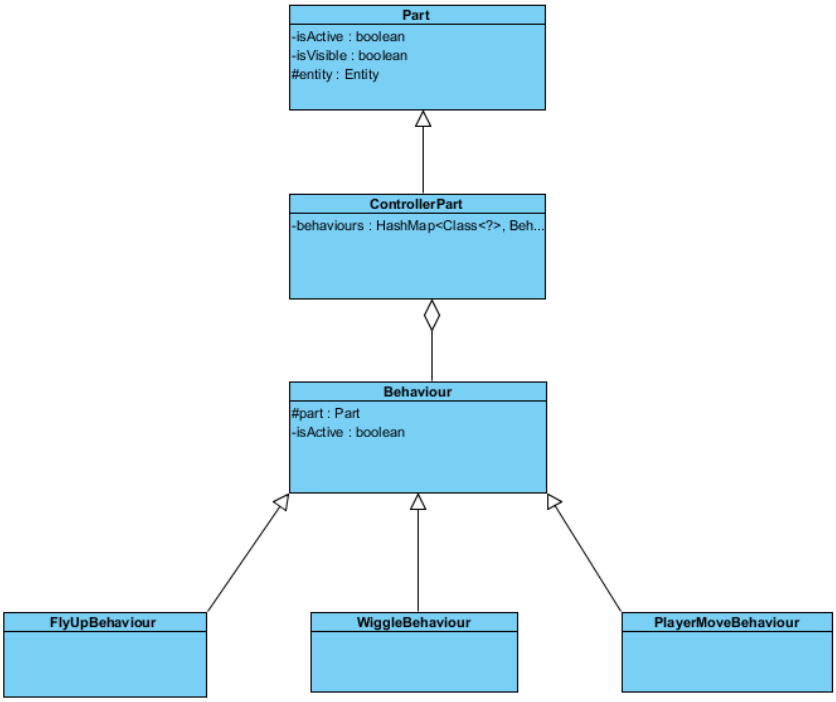


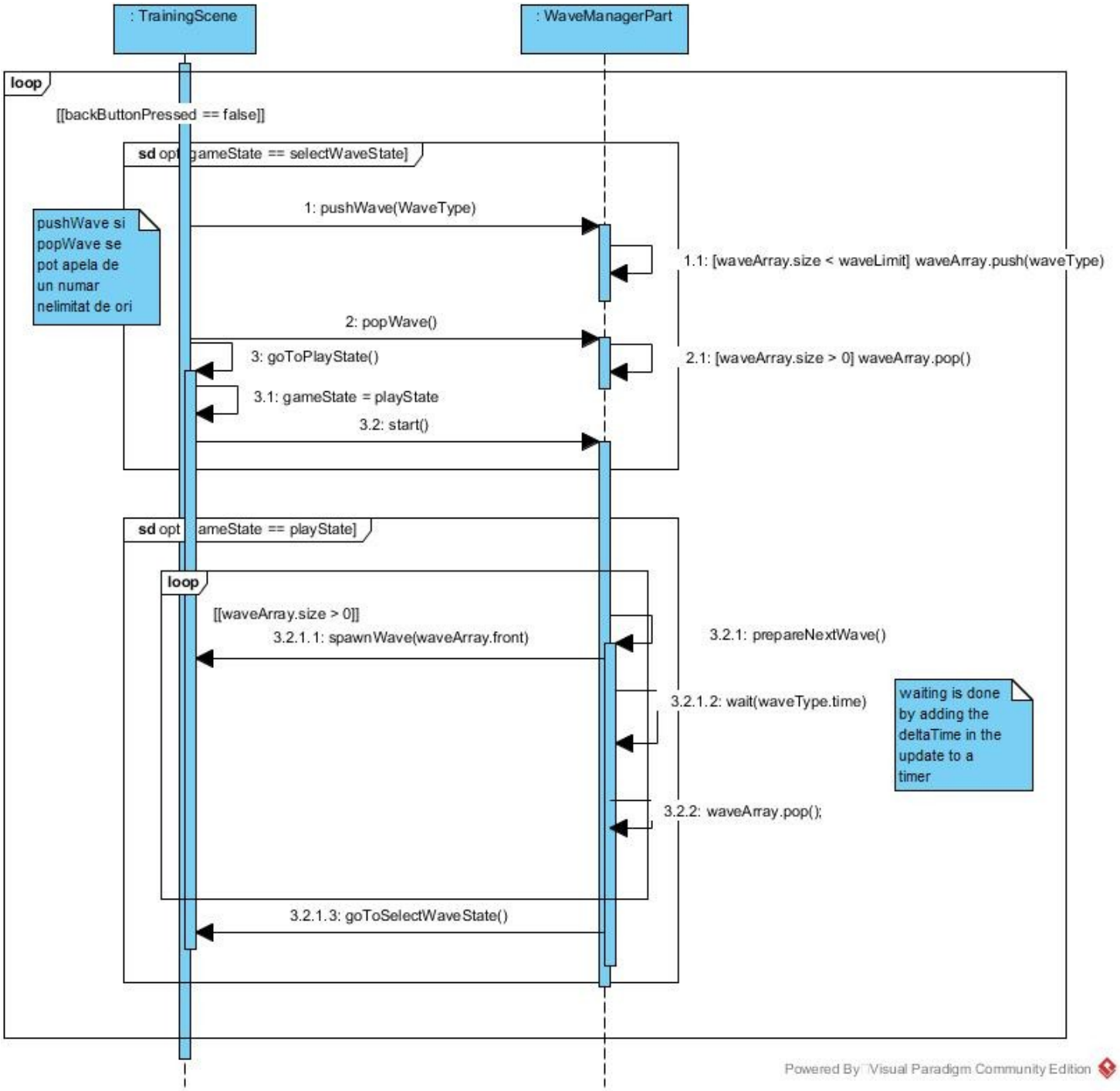
Diagrama urmatoare descrie relatia dintre clasele **ControllerPart** si diferite derivari ale clasei **Behaviour**:



b. WaveManagerPart

O alta clasa importanta din cadrul proiectului este **WaveManagerPart**. Aceasta se ocupa cu partajarea encounter-urilor. Contine metode pentru a adauga sau sterge tipuri de wave-uri dintr-o coada ce va fi folosita, dupa apelarea metodei **start()**, pentru a spawna la momentul potrivit valurile de nave inamice respective.

Functionarea acestei componente este descrisa vizual cu ajutorul urmatoarei diagrame de secventa:



c. Specificatii JML

StatsPart

```
//@public normal_behaviour
//@requires !isInvulnerable;
//@requires !shouldInvulnerable;
//@assignable currentHitpoints;
//@ensures currentHitpoints -= inficted Damage;
//@also
//@requires !isInvulnerable;
//@requires !shouldInvulnerable;
//@requires currentHitpoints <= 0;
//@assignable shouldDie;
//@ensures shouldDie == true;
//@also
//@requires !isInvulnerable;
//@requires !shouldInvulnerable;
//@requires !isInvulnerable;
//@assignable invulnerablePassed;
//@assignable shouldInvulnerable;
//@ensures invulnerablePassed == 0;
//@ensures shouldInvulnerable == true;

public void inflictDamage(float inflictedDamage) {
    if (isInvulnerable || shouldInvulnerable) {
        return;
    }

    this.currentHitpoints -= inflictedDamage;
    if (currentHitpoints <= 0) {
        shouldDie = true;
    }

    if (invulnerableTime > 0 && !isInvulnerable) {
        invulnerablePassed = 0;
        shouldInvulnerable = true;
    }
}
```

```java
public class HudGraph {
    /*spec_public*/private boolean isActive = true;
    /*spec_public*/private boolean isVisible = true;

    /*spec_public*/private HashMap<String, HudNode> graph = new HashMap<>();

    //@public normal_behaviour
    //@ensures \result == isActive;

    public boolean isActive() {
        return isActive;
    }

    //@public normal_behaviour
    //@assignable this.isActive;
    //@ensures this.isActive == isActive;

    public void setActive(boolean isActive) {
        this.isActive = isActive;
    }

    //@public normal_behaviour
    //@ensures \result == isVisible;

    public boolean isVisible() {
        return isVisible;
    }

    //@public normal_behaviour
    //@assignable this.isVisible;
    //@ensures this.isVisible == isVisible;

    public void setVisible(boolean isVisible) {
        this.isVisible = isVisible;
    }

    //@public normal_behaviour
    //@requires graph.containsKey(nodeKey);
    //@ensures \result == true;
    //@also
    //@requires !graph.containsKey(nodeKey);
    //@ensures \result == false;
```

```
    public boolean has(String nodeKey) {
        return graph.containsKey(nodeKey);
    }

    //@requires this.has(nodeKey);
    //@ensures \result == graph.get(nodeKey);
    //@also
    //@requires !this.has(nodeKey);
    //@signals_only IllegalArgumentException;

    public HudNode get(String nodeKey) {
        if (!this.has(nodeKey)) {
            throw new IllegalArgumentException();
        }

        return graph.get(nodeKey);
    }

    //@requires !this.has(nodeKey);
    //@ensures graph.put(nodeKey, node);
    //@also
    //@requires this.has(nodeKey);
    //@signals_only IllegalArgumentException;

    public void attach(String nodeKey, HudNode node) {
        if (this.has(nodeKey)) {
            throw new IllegalArgumentException();
        }

        graph.put(nodeKey, node);
    }

}
```

d. Teste JUnit

```java
@Test
public void StatsPartTest()
{
    EntityFactory fac = EntityFactory.getInstance();
    Entity returned = fac.createEntity(EntityFactory.EntityType.ALIEN_BASIC_WHITE);
    StatsPart returnedStatsPart = new StatsPart();
    returnedStatsPart = returned.get(StatsPart.class);

    StatsPart expected = new StatsPart();
    expected.maxHitpoints = 120;
    expected.setCurrentHitpoints(80);
    expected.faction = StatsPart.Faction.FACTION_ENEMY;
    expected.statsType = StatsPart.StatsType.SHIP;
    expected.damage = 50;
    expected.invulnerableTime = (float) 0.1;
    expected.setActive(true);

    assertTrue(returnedStatsPart.equals(expected));

    // Testing inflicDamage method
    expected.setCurrentHitpoints(70);
    returnedStatsPart.inflictDamage(10);
    assertTrue(returnedStatsPart.equals(expected));
}
```

```java
@Test
public void createBasicBulletTest()
{
    // createBasicBullet will return an entity that has several parts
    // Testing for some of the parts

    EntityFactory fac = EntityFactory.getInstance();
    Entity returned = fac.createEntity(EntityFactory.EntityType.BULLET_BASIC);

    assertTrue(returned.get(HitboxPart.class).getColor().equals(Color.GREEN));

}
```

```java
@Test
public void waveDictionaryTest()
{
    WaveInfo expected = new WaveInfo();
    expected.type = WavesFactory.WaveType.WAVE_MIXED1_BLOCK;
    expected.name = "Mixed Block";
    expected.duration = 10;

    WaveInfo returned = new WaveInfo();
    returned = WaveDictionary.getInstance().getWaveInfo(WavesFactory.WaveType.WAVE_MIXED1_BLOCK);

    assertTrue(returned.equals(expected));


    expected = new WaveInfo();
    expected.type = WavesFactory.WaveType.WAVE_BASIC_BLOCK;
    expected.name = "Basic Block";
    expected.duration = 15;

    returned = new WaveInfo();
    returned = WaveDictionary.getInstance().getWaveInfo(WavesFactory.WaveType.WAVE_BASIC_BLOCK);

    assertTrue(returned.equals(expected));

}
```

```java
@Test
public void waveCreationTest()
{
    ArrayList<Entity> array = new ArrayList<Entity>();
    array = WavesFactory.getInstance().createWave(WavesFactory.WaveType.WAVE_BASIC_BLOCK);
    assertTrue(array.size() == 20);
}
```