The Ruined One

Darie Alexandru
1210B

Proiectarea contextului și descriere detaliată:

Noxian de la naștere, Shieda Kayn și alții ca el au fost recrutați ca și copii soldați, o practică crudă folosită doar de cei mai crunți comandanți din imperiul lui Boram Darkwill. După bătălia dezastruoasă de la Placidium din Navori, invazia noxienilor a devenit un război prelungit.

Kayn și ceilalți au fost o avangardă reticentă, confruntându-se cu trupe dezorganizate de localnici care își apărau casa de acești invadatori care se întorceau. În timp ce tinerii săi camarazi erau uciși sau fugeau de pe câmpul de luptă, Kayn nu a arătat nici o teamă. Și-a aruncat sabia grea și a smuls o seceră căzută, întorcându-se să-i înfrunte pe satenii șocați din Ionia.

Peste două zile, după ce s-a raspândit din vorbă in vorbă bătălia, Ordinul Umbrei a venit la scena masacrului. Zed, liderul acestui ordin, știa că zona invadată nu a avut nici o semnificație tactică. Masacrul a fost doar un mesaj, acela că Noxus nu arată milă.

O sclipire a oțelului i-a atras atenția lui Zed. Un copil, nu mai mult de 10 ani întins în noroi, cu o secure ruptă o ridica spre maestrul umbrelor. Acesta vede potențialul baiatului și decide să-l antreneze. Dupa mult timp și antrenament, Zed îi dă un ultim test, sa recupereze securea însuflețită numită Rhast îngropată în Noxus.

Kayn recuperează securea, dar din momentul în care acesta își încătușează degetele pe mănerul securei, corupția se instaurează. Rhast are acum o gazdă pe care o poate folosi pentru a-și distruge inamicul etern, Aatrox, însa Kayn nu se lasă ușor controlat.

*Kayn acum, împreuna cu Rhast se află în catacombele lui **Aatrox** pentru a-l înfrunta. Pe parcursul luptei, unul din cei doi v-a prelua controlul asupra altuia, fiecare având calități (și defecte) diferite.*

Proiectarea sistemului și descriere detaliată:

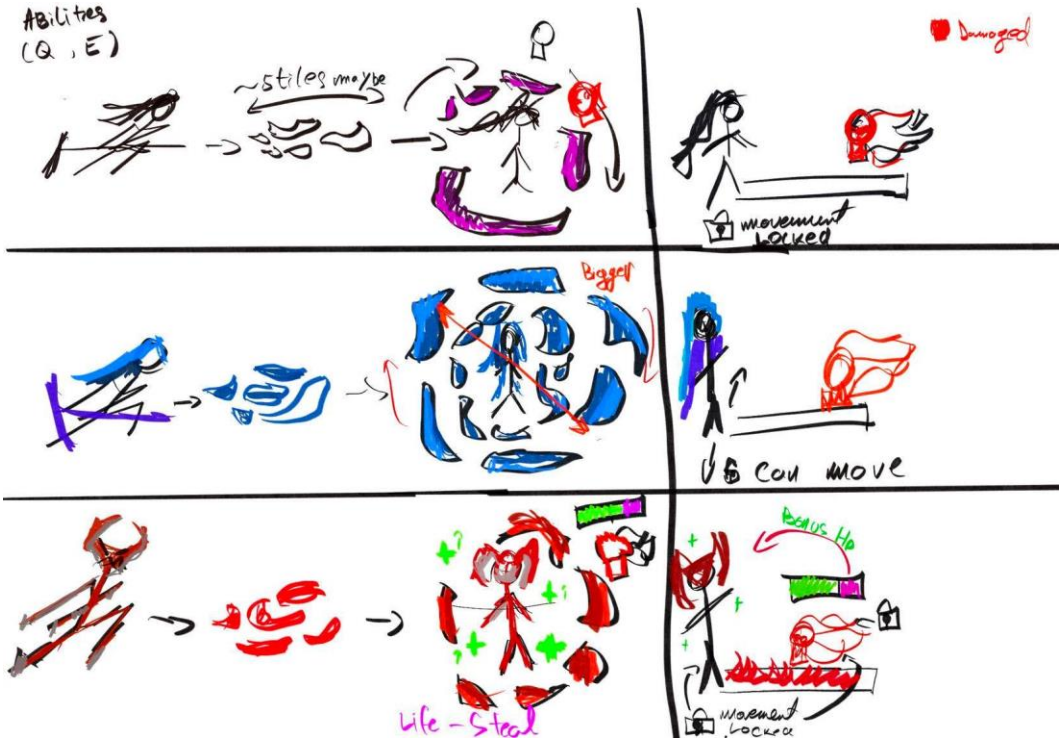
Playerul v-a avea de confruntat diferiți inamici, fiecare inamic oferindu-i două tipuri de **puncte de evoluție**. Acesta , dupa ce a acumulat suficiente puncte de un fel, le poate folosi pentru a **evolua** caracterul în două forme diferite.

Shadow Reaper Kayn – Cazul în care **Kayn** îl stăpânește pe **Rhast**.

-Mult mai mobil
-Daunele provocate crescute
-Viata totală scăzută

Darkin Rhast – Cazul în care **Rhast** preia controlul asupra gazdei, **Kayn**.

-Daunele provocate regenerează viata
-Viată crescută



- **Caracterul** se mișcă de pe **WASD** sau **ARROWS**.
- De pe **SPACE** se interactionează cu **OBIECTE**. (ex: Uși, torțe, etc.)
- **Click-Stânga** este **BASIC ATTACK**.

*Acest atac constă in aruncarea securei (Rhast) în inamici precum un Boomerang. Jucătorul **nu poate refolosi** acest atac pană când securea nu ajunge înapoi la acesta, motiv pentru care aceste lovituri trebuie sa fie bine gândite.*

- **Q, E** folosesc **abilitațile 1** și **2**.

Acestea pot fi folosite fără prea multe constrângeri. (Nu pot fi folosite simultan decât dacă caracterul jucătorului se află in forma lui Shadow Reaper Kayn)

- **Camera** nu este centrată pe caracter, ci pe camera în care se află; aceasta se schimbă odată ce caracterul iese din cadru.

<u>O performanță bună constă în Highscore-ul obținut la final cât si timpul în care l-a înfrânt pe Aatrox. Jocul se poate considera câștigat dacă playerul l-a ucis pe Aatrox.</u> **<u>Pentru a intra în camera lui Aatrox, acesta v-a avea de completat anumite taskuri (ex: Colectarea unei chei) pentru a deschide ușile camerei.</u>**

În funcție de dificultatea aleasă, inamicii au viata crescută cu un anumit procentaj , iar playerul are viata scăzută. Sunt 3 tipuri de dificultăți: ușor, mediu, greu.

Proiectarea conținutului și descriere detaliată:

Definirea Caracterelor:

Kayn este un tânăr de 20 de ani. Acesta a fost recrutat în armată de la o vârstă fragedă. După o luptă sângeroasă, Kayn este găsit și rectutat de către **Zed**. Acum misiunea acestuia este de a-l înfrânge pe Aatrox.

Aatrox este un **Darkin**, o armă a distrugerii ce ucide tot ce-i iese în cale. Acesta a fost blestemat cu viață eternă și tot ce caută este moartea. Acesta în joc are 3 tipuri de atac: Spawneaza gardieni-fantoma, ploaie de săbii și o lovitură wide de sabie.

Fantomele sunt doar niște suflete pierdute ce au fost odată niște aventurieri curioși și îndrăzneți subestimând puterea lui Aatrox. Acestea se află sub controlul lui **Aatrox**, prin urmare vor încerca sa ucidă tot ce intră în catacombe. E clar că cine moare din cauza lor nu e demn sa lupte contra lui **Aatrox**.

Animatiile Obiectelor:

Deschiderea torței (altar):

Deschiderea / Închiderea unei uși mari

Mișcarea Playerului pe axele X Y:

(FOLDERUL RES CONTINE MAI MULTE)

Obiectele:

În joc există obiecte de care playerul are nevoie neapărat pentru a progresa , obiecte de interes ce nu influentează neapărat gameplay-ul respectiv jocul și obiecte ce necesită alte obiecte pentru a le folosi.

Din prima categorie putem enumera **chei**, pentru a deschide anumite uși inclusiv camera lui Aatrox, **chibrite/torțe**, pentru a aprinde anumite altare, orburile de evoluție ce ajută la transformare, etc.

Obiectele de interes ce nu influentează neapărat gamplayul sunt banii, extra-life-points de la inamici și orice alte lucruri de care playerul s-ar putea lipsi și tot ar fi capabil să termine jocul.

În ultima categorie se află și obiecte din prima categorie, ușile , altarele și cuferele sunt obiecte ce necesită **Cheie_Dungeon_Door**, **Torță**, **Cheie_Cufar.**

Interacțiuni:

Dacă jucătorul este lovit de fantome, acesta este rănit iar fantomele se teleportează intr-o locație puțin mai îndepărtată de jucător.

Ușile, cuferele , torțele și alte obiecte ce necesită obiecte adiționale nu vor putea fi folosite până când jucătorul are cele necesare.

Abilitătile lui Aatrox au „o umbră” de care jucătorul se poate folosi pentru a avea timp să se ferească înainte de a fi lovit de ele.

Parcursului pe care îl poate avea un jucător:

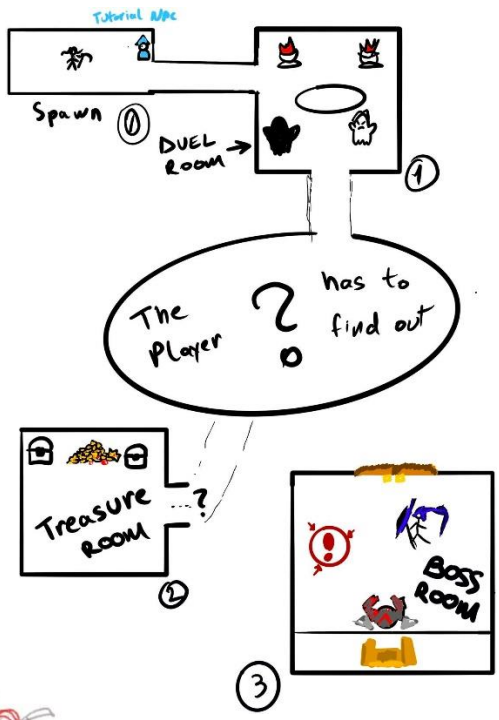
Jucătorul poate finaliza jocul cu un punctaj cât mai mare prin diferite strategii, mai ales prin evoluția pe care o alege și timpul petrecut în explorarea dungeonului.

Proiectarea nivelurilor și descriere detaliată:

(Nivelurile trebuie sa fie generate într-o oare care măsură aleator, așadar nu Pot define un nivel concret, dar… :)

Există trei (+1) tipuri de camere pe care jucătorul le poate Întâmpina în parcurgerea catacomberi:

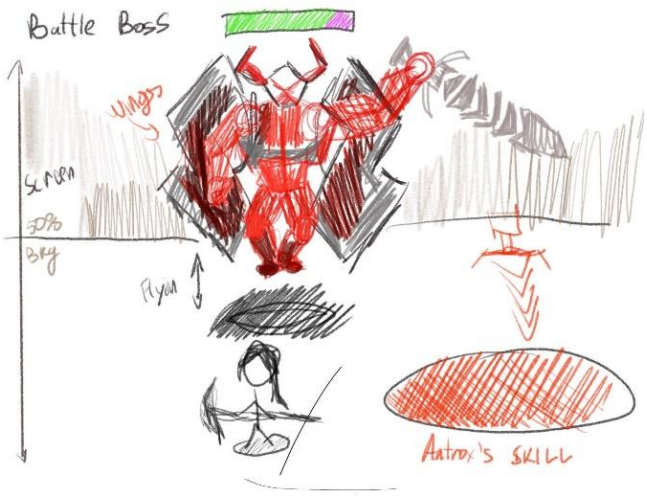
0- Camera Tutorial / Lobby
   În această camera, playerul nu poate fi rănit deoarece nu există inamici; de asemenea acesta nu poate acumula resursele necesare pentru finalizarea jocului.



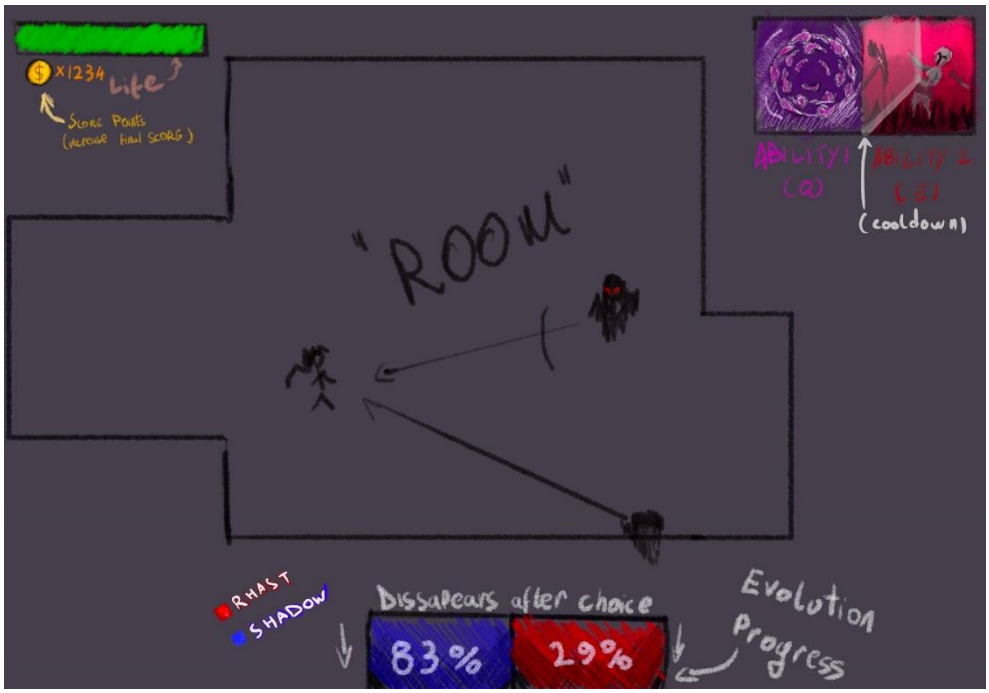
1- Duel Room
   În camerele de tip „Duel” , playerul are de confruntat Inamici de diferite forme. Fantome ce au fost odată niște soldați cu aceeași dorință de a-l înfrânge pe Aatrox.

2- Treasure Room
   După cum spune și numele, aceste camere sunt pline cu diferite obiecte ce influentează Highscore-ul cât și gameplayul. Acestea pot conține iteme de care playerul are nevoie pentru a continua jocul. Unele camere pot sa aiba si inamici în ele!

3- Boss Room
   Intuitiv, aceasta este unică iar în ea se află Aatrox, The World Ender. Playerul, odată ce a intrat în ea, nu o mai poate părăsi până la înfrângerea lui Aatrox (sau pâna când pierde).

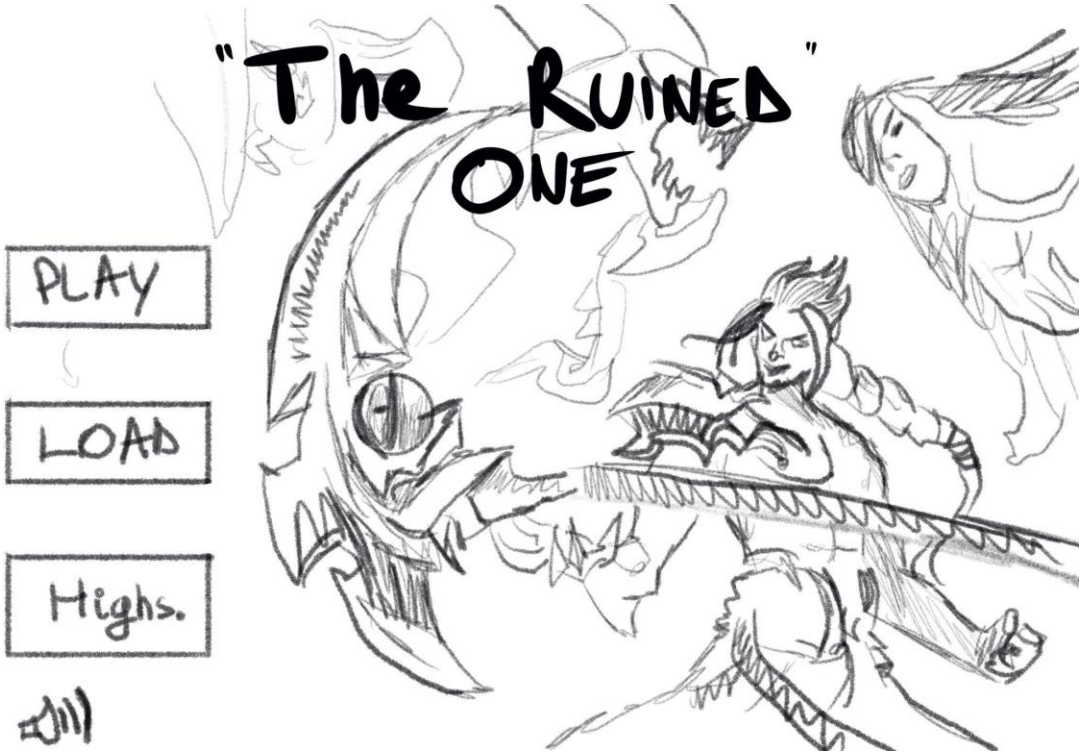


Proiectarea interfeței cu utilizatorul și descriere detaliată:



În timpul jocului, playerul vede pe ecran în dreapta sus cele două abilități si eventual durata de timp pâna cand o poate refolosi, în stâga sus viata pe care o are, fix sub viată, scorul (ce v-a fi folosit în calculul Highscore-ului), iar in centru jos poate vedea progresul evolutiei, acest GUI v-a dispărea din momentul in care playerul își alege forma (întrucât nu poti sa ti-o schimbi și nu are sens sa mai apară odată ce și-a schimbat forma).

Meniul principal



Are 3 butoane principale:

PLAY – Creează un nou joc si introduce playerul în acțiune.

LOAD – Încarcă o salvare din baza de date și incepe jocul în functie de datele preluate. (Pozitie, evolutie, scor)

HIGHSCORE – Afișează top #(n) scoruri obținute de alte instanțe ale jocului.

Și un buton în stânga jos a ecranului ce opreste sau nu sonorul.

Minim 2 game sprite-uri ce vor fi utilizate:

ETAPA 2:

-Diagrame UML:

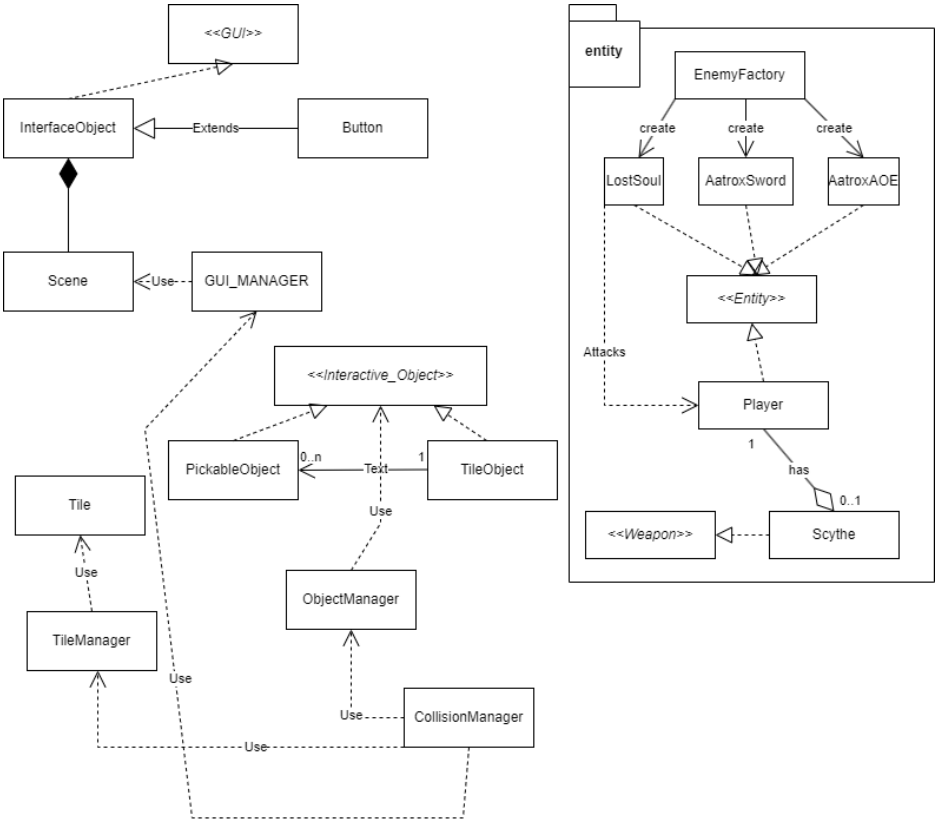
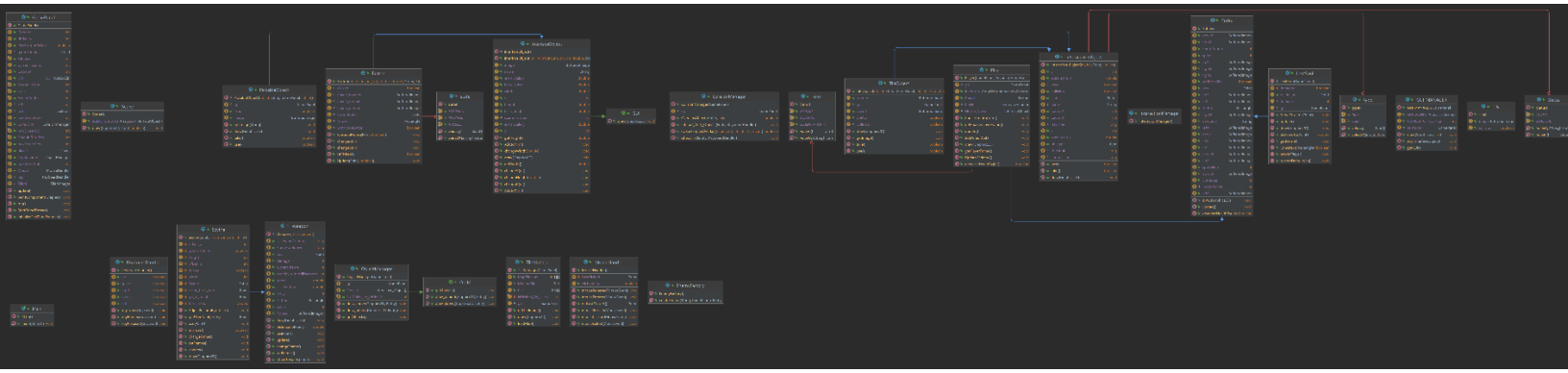


Diagrama de mai sus este o diagrama UML “simbolica” cu scopul de a prezenta clasele jocului si relatiile dintre acestea. Toate acestea fac parte din anumite packageuri, cel mai important (in opinia mea) fiind “entity”. Toate clasele prezentate mai sus fac parte dintr-un container “GamePanel” pe care am decis sa nu il prezint in diagrama intrucat o face mult prea amestecata.

IN FOLDERUL JOCULUI SE AFLA UN FISIER “UML.png” CE CONTINE DIAGRAMA FULL A JOCULUI GENERATA DE IntelliJ (Ultimate Edition)



Cum functioneaza fiecare clasa [IN DETALIU] (pe cat posibil)

MAIN PACKAGE:

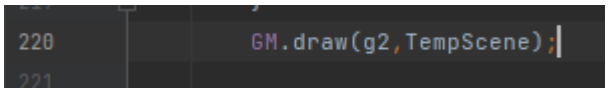
- **Main:**
  Clasa Main conține doar instanța containerului **GamePanel**. Tot ce ține de joc se intâmplă în containerul GamePanel.
- **Gamepanel:**
  Clasa Gamepanel conține informații necesare rulării jocului precum numarul de randuri si coloane de tiles pe care le poate genera, lungimea și lățimea windowului, viteza cu care ruleaza jocul (60 FPS), dimensiunea hărții, Cadranul pe care îl randeaza (optimizare), scena etc.

  Jocul rulează pe un thread (.StartGameThread() si .run()), ce controlează viteza randării si apeleaza metoda **update()**, în care se afla toate componentele jocului ce trebuie updatate la fiecare frame in functie de restrictiile pe care le are.

  În functie de ce apasă utilizatorul, variabila „Current_Scene” se schimbă:
  Scena 0 reprezintă meniul principal.
  Scena 1 reprezintă jocul.
  Un exemplu de gestionare a ce desenează pe ecran se poate vedea aici:

```
switch (Current_Scene)
{

    case 0->
    {...}
    case 1->
    {
        TileM.draw(g2);
        //ObjManager.draw(g2);      Pentru Better Visuals, facem ca functia obj manager sa primeasca un al 2-lea parametru care sa randeze doar obiectele din spatele/fata playerului
        ObjManager.draw_behind(g2,player);
        player.draw(g2);
        ObjManager.draw_above(g2,player);
        test.draw(g2);
        try
        {
            if(player.Rhaast.isIn_use())
                player.Rhaast.draw(g2);
        }catch(Exception e)
        {e.printStackTrace();}

    }
```

  Aici trece printr-un switch ce determina ce trebuie desenat.

Iar 

desenează toate guiurile respective scenei. (Ajung acolo la GUI_MANAGER)
Cele mai importante componente sunt **MouseHandler** si **KeyboardHandler** ce fac diferența dintre un joc si un videoclip. Fără acestea, jocul nu ar fi jucabil.

- **MouseHandler:**
  O clasă ce implementează interfața MouseListener. Simplă dar utilă. Aceasta are doi membri ce informează jocul unde s-a apăsat ultima dată pe ecran și daca s-a apasat click stânga.
- **KeyboardHandler:**
  Clasa KeyboardHandler implementează KeyboardListener. Asemanător cu MouseHandler, aceasta informeaza jocul ce taste relevante sunt apăsate. Cea mai mare utilitate a KeyboardHandlerului este că permite jucătorului să se deplaseze pe diagonală deoarece verifică individual ce taste sunt apăsate cu ajutorul membrilor Boolean din ea.

```
gp.CollisionM.CheckTile( entity: this,KeyH);
gp.CollisionM.Interact_With_Objects( entity: this, KeyH);
if (KeyH.up)// && y>0)
{...}
if (KeyH.down)//&&y<ScreenHeight-Tile_size)
{...}
if (KeyH.left)// && x>0)
{...}
if (KeyH.right)// && x<ScreenWidth-Tile_size)
{...}
```

- **ColisionManager:**
  ColisionManager comunică cu containerul din care face parte prin intermediul unui parametru privat numit „gp”. Gamepanelul cand initializează ColisionManager, se va transmite pe el însuși constructorului pentru a realiza conexiunea.

  Are o metodă ce verifică daca o entitate (Playerul adica) poate interactiona cu un obiect numită **Interact_With_Objects(…)**. Dacă e apasat space (tasta de interactiune) si Hitboxul playerului suprapune imaginea obiectului si se află in același cadran, atunci playerul face: take() sau use(). În funcție dacă obiectul e **Pickable** sau **nu (tipul e decis de un enum precum si faptul daca e ACTIV sau INACTIV. Variabila [USED] arata daca a fost „deblocat” sau nu).**

  O metodă „**CheckTile**(…)” ce verifică, în functie de directia entitatii, ce urmează să atingă (predicting)**.** Aceasta calculează pozitia tileului pe care urmează să il atingă și verifică daca e solid sau nu. Dacă e **solid** , atunci pentru frameul respectiv, variabila „collisionON” (atribut al entitatilor) devine True, blocând entitatea în loc.

  Restul metodelor sunt doar pentru lizibilitate.

GUI PACKAGE:

- **GUI:**
  O interfată ce clarifică că orice face parte din user **INTERFACE** trebuie sa fie desenabil.

```
public interface GUI {
    1 implementation
    void draw(Graphics2D g);
}
```

- **InterfaceObject:**

  Implementează GUI, conține informatii precum în ce scenă se află, pozitia si scalarii imaginii, Imaginea pe care o randează și numele obiectului. Ea singură nu prea face nimic. Are metode ce ajută la modificarea si obtinerea membrilor (setters și getters).

- **Button:**

  Butonul e un InterfaceObject mai complex, întrucât are 3 imagini în loc de una. Imaginea normală, imaginea când mausul suprapune butonul si imaginea butonului apăsat. În plus, are doua metode ce observă mausul și o metodă Update() (ce nu există la interfaceobjects) care schimbă imaginile corespunzător și specifică daca a fost apăsat sau nu.

  Fiecare buton ține minte ultima stare a mausului. Ca să se considere apăsat, mausul trebuie să se afle în interiorul butonului, ultima stare a mausului tinută minte sa fie „CLICKED" si starea actuală să fie „NOT_CLICKED". Adică pur si simplu a fost apasat click (**NU ȚINUT APĂSAT).**

- **Scene:**

  Scenele nu sunt nimic mai mult decât un Array de InterfaceObjects.

  Contine o metodă Draw() care desenează toate InterfaceObjects care apartin acesteia. **În Draw() se face Update() la butoane.**

```
public void draw(Graphics2D g2,Point MousePosition, boolean MouseState)
    for (int i = 0; i < SCENE_OBJECTS.size(); i++) {
        if(SCENE_OBJECTS.get(i).getClass()== Button.class)
        {
            Button temp;
            temp=(Button)(SCENE_OBJECTS.get(i));
            temp.Update(MousePosition,MouseState);
        }
        SCENE_OBJECTS.get(i).draw(g2);
    }
}
```

- **GUI_MANAGER:**

  Clasa aceasta este cea mai importantă d.p.d.v al randării guiului. GUI_MANAGER are un vector de scene (deci pot fi reprezentate ca o matrice), numărul de scene și containerul din care face parte.

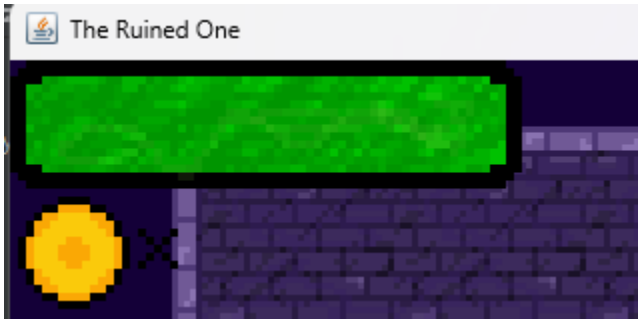
  Frumusețea clasei constă în ușurința de a adăuga obiecte si organizarea ei (sunt foarte mandru de ea).

  Are o metodă **getGUI()** ce adaugă toate imaginile corespunzătoare jocului, iar tot ce trebuie să facă cineva pentru a adăuga un obiect în gui este să facă un add(OBIECT) în getGUI(…), oferind codului o lizibilitate pe care o consider importantă.

De exemplu:

```
add(new InterfaceObject(1,0,0,64,32,4,4,"Health And Coins","res/Player/Health_bar_0.png"));
```

adaugă un InterfaceObject în scena **1**, la coordonatele 0,0 , cu **WIDTH** si **HEIGHT** 64 respectiv 32, scalarele fiind 4 și 4 pentru **W & H,** numit „**Health and Coins”** care are pathul imaginii; „res/…”.



În cazul în care programatorul introduce ceva gresit, functia add(…) se ocupă de asta, modificând valorile din GUI_MANAGER.

```
public void add(InterfaceObject obj)
{
    try
    {   //Preia scena numarul x si incearca sa adauge in scena obiectul 'obj'
        GUI_SCENES.get(obj.scene_number).SCENE_OBJECTS.add(obj);
    }
    catch(Exception e)  //Daca obiectul e dat cu un numar de scene mai mare decat cel asteptat (Caz de eroare)
    {
        e.printStackTrace();
        GUI_SCENES.add(new Scene());            //Creeam o noua scena
        obj.scene_number=GUI_SCENES.size()-1;   //Setam numarul scenei obiect ca facand parte din ultima scena
        NUMBER_OF_SCENES++;

    }

}
```

TILE PACKAGE:

- **Tile:**

  Clasa Tile surprinzător de simplistă. Nu conține nimic mai mult decât imaginea tileului și dacă e colisionable sau nu.

```
public class Tile {
    public BufferedImage image;
    public boolean collision=false;
}
```

- **TileManager:**

  TileManager e toată operatiunea.

  Cel mai important: conține informatii despre containerul din care face parte.

  Acesta are un vector cu totalitatea tileurilor ce pot fi regăsite în joc, si care sunt solide.

```
tile[21].image=ImageIO.read(new FileInputStream("res/Sprites/carpet.png"));
//
tile[22].image=ImageIO.read(new
FileInputStream("res/Sprites/up_topleft_corner.png"));tile[22].collision=true;
```

si o matrice cu harta ”map1.txt” din ”./res” numită MapTiles.

Metoda „loadMap()” realizează citirea si parsingul fisierului, aceasta modificând atributele containerului ce reprezinta dimensiunile hărtii. Dacă IDUL unui tile din mapă nu există, atunci e înlocuit cu un ”MISSING TILE”. Dacă imaginea nu există atunci o sa fie un gol în hartă, dezvaluind culoarea ROZ a JFrameului.

Metoda Draw() randează harta de dimensiuni **row x col** comform cadranului indicat de gamepanel.

SUPEROBJECTS PACKAGE:

- **Interactive_Object:**

  Reprezintă un obiect menit să fie in game-scene(1) . Are, pe lânga informatii legate de poziționare, informatii ce țin de timpul cand poate fi refolosit, tipul obiectului (solid sau item) statusul obiectului(active sau inactiv) și daca e colisionable. ACEASTA ESTE O INTERFATĂ CE ESTE IMPLEMENTATĂ DE CLASELE DE MAI JOS!

  Fiecare Obiect ce implementeaza interfata aceasta trebuie să-și defineasca metoda use() si take().

- **PickableObject:**

  Generează imaginile obiectului în functie de numele acestuia.

```
switch(name)
{
    case "Dungeon_Key"->
    {
        image= ImageIO.read(new FileInputStream( name: "res/Sprites/
    }
    case "Chest_Key"->
    {
        image= ImageIO.read(new FileInputStream( name: "res/Sprites/
    }
    default->
    {
        image=null;
    }
}
```

  Ține containerul din care face parte și adaugă obiectul din inventarul playerului când se face take().

```
public boolean take()
{
    gp.player.Inventory.add(this);
    this.x=-1000; // Dispare de pe ecran;
    return true;
}
```

  După ce e luat obiectul, îl scoatem de pe hartă. (Eventual îl stergem dacă nu mai avem nevoie de el)

- **TileObject:**

  Cu aceeași „mecanică” de la obiecte. Alege textura în functie de nume și schimbă textura și faptul daca e colizibil în funcție de statusul ei si variabilile ColAct(ive) si ColInact(ive).

  De exemplu, usa:

```
Obiecte[0]=new TileObject(864-2*gp.Tile_size,0,"Dungeon_Door",gp,0,0,2,2,true,false);
```

E colisionable când e inactive (închisă) și invers când e deschisă.

Când se folosește use(), se verifică dacă a fost deblocat prin variabila ”used” , dacă nu a fost încă deblocat, caută în inventarul playerului obiectul necesar deblocării.

- **ObjectManager:**

  Ca și celelalte managere, are un membru ce specifică din ce container face parte, o metodă „getObjects()” ce instanțiază obiectele.

  Nimic special la ea cu excepția metodelor draw_behind/above() pentru a randa în ordinea „layerelor”, desenând peste player sau nu.

  Are un vector de Interactive_Objects.

ENTITY PACKAGE:

- **EnemyFactory:**

  E o fabrică ce generează inamici cu ajutorul metodei “createEnemy(String TipInamic)”.

  Un inamic nu poate fi creat cu ajutorul constructorului propriu, ci trebuie neapărat prin intermediul fabricii.

  Fabrica e statică pentru a nu fi necesar un obiect de tip fabrică

```
public class EnemyFactory {
    1 usage
    public static Entity createEnemy(String EntityName, GamePanel gp)
    {
        switch (EntityName)
        {
            case "LostSoul"->
            {
                return new LostSoul(gp);
            }
            case "AatroxAOE"->
            {...}
            case "FlyingSword"->
            {...}
            default ->
            {
                break;
            }
        }
        return new LostSoul(gp);

    }
}
```

- **Entity:**

  O entitate e foarte complexă.

  Aceasta trebuie să stocheze pozitia pe hartă, numărul spriteului curent (sprtieNum), viteza cu care să se schimbe spriteurile (SpriteCap) si un contor pentru viteză (SpriteCounter), 4 Imagini corespunzătoare fiecărei directii (up down left right), un Hitbox (Rectangle), viteza entității și cât HP are.

  Metodele ce trebuie scrise de fiecare clasă ce o implementează sunt ”draw(..)” și ”update()”.

  Are și o metodă ”decreaseHealthBy(int x)” ce scade viata entitătii cu un întreg si returnează TRUE sau FALSE daca e <=0 după lovitură.

- **LostSoul:**

  Fiind o entitate hostile, această clasă poate răni alte entități și sa fie lovită la rândul ei.

  Aceast inamic are o ”victimă”. Cu ajutorul metodei ”MoveTowards()” se îndreaptă spre victimă. Dacă își ia damage, vectorul direcției își schimbă sensul, dând în spate inamicul.

  Acesta se folosește de metoda **CanAttack**(Victima) să vadă dacă este în zona victimei, și dacă este, atunci să o atace cu ajutorul metodei **attack**(victim). Un „LostSoul” dispare odată ce reușește să atace cu success. Revenind la o pozitie random înafara ecranului. (Not random yet)

  Aceste două metode oferă o citire a codului ușor de înteles.

```
if(CanAttack(gp.player.HitBox))
{
    //System.out.println("I can attack!");
    attack(gp.player);
}
```

Acum, în loc sa verificam toate fantomele pe care le loveste securea printr-un „for”, verificam la fiecare update() al fantomei dacă aceasta intersectează singrua sursă de damage. (Cam Hardcoded dar ajută la eficiență).

```
if(gp.player.Rhaast.Hitbox.intersects(HitBox) && gp.player.Rhaast.isIn_use())      //Daca e lovit de secure && a fost aruncat (Rhaast ramane la pozitia playerului doar ca nu mai e desenat / updated)
{
    if(!damaged)        //Daca nu o fost lovit pana acum
    {                   //Scade-i viata fantomei
        if(decreaseHealthBy(gp.player.Rhaast.damage))   //functia returneaza true daca e mort, ii scade hpul
        {
            restoreDefaultVals();
        }

        //Si marcheaz-o ca lovita
        damaged=true;
    }
}
else {
    damaged=false;
}
```

- **Player:**

  Playerul este tot o entitate, dar nu hostile (deci nu are o victimă).
  Acesta are un inventor format din ”Interactive_Objects”, un KeyboardHandler și Containerul din care face parte. De asemenea are si un ”Viewer” pentru un InterfaceObject numit ”Health_Bar_Viewer” ce schimbă dimensiunea HPului cu un anumit procentaj atunci când își ia damage. Playerul folosește o armă de tip ”Scythe”, dar aceasta poate lipsi! (se poate comenta linia 29 din constructor) Nu v-a influența jocul cu exceptia că nu v-a putea ataca.

  Esența Playerului și consider și cea mai interesantă parte este că acesta se poate transforma pentru a-și schimba STATSURILE cu ajutorul metodei ”transformInto(form FORM)”. ”form” este un enum ce conține formele de transformare: NORMAL, RHAAST și SHADOW_KAYN.

  Playerul schimbă cadranele containerului în functie de unde merge.

```
if(HitBox.y>gp.ScreenHeight)
{

    if(gp.CadranY<gp.maxWorldRow/gp.maxScreenRow) //Se verifica daca nu e capat de harta
    {
        gp.CadranY++;
        pos.y=0;
        //Rhaast.pos.y-=gp.ScreenHeight;
        Rhaast.AdjustPositionBy(0,-gp.ScreenHeight);
    }

}
```

- **Weapon:**

  Interfață care explică comportamentul unei arme. Orice armă are o viteză cu care se deplasează, un DAMAGE pe care îl provoacă , accelerație, și mai multe frameuri pentru animatie, un COOLDOWN înainte de a fi reutilizată, etc.

Cea mai importantă metodă pe care trebuie să o suprascrie clasa care o implementează e ”use” și ”update” ce explică modul de funcționare a acesteia.

- **Scythe:**

  Unele arme, precum securea (Scythe) are un OWNER la care să se întoarcă. Aceasta își adaptează pozitia atunci cand se schimba cadranele OWNERULUI (updateCadrane la player). Securea, fiind arma principală a playerului, nu are COOLDOWN dar poate fi utilizată doar dacă playerul o are (adică securea nu se îndreaptă spre un punct).

```
@Override
public void use(Point WhereToGo) {
    if(!in_use)             //daca nu se indreaptă nicaieri
    {
        in_use=true;                //atunci o marcăm ca si ”in use”
        goingToPoint=true;          //Variabila care determină dacă se duce/întoarce
        go_to_point=WhereToGo;      //Ii zic unde sa se ducă
        //come_back_point=getUserPoint(Owner);
        pos=new Point(Owner.pos); //Incepe de la aruncător
    }

}
```

SABLOANELE FOLOSITE IN JOC SUNT “COMMAND”, “FABRICA”, si “STATE” (dar adaptat pe joc)

(fabrica e mai sus)

Aceasta este interfata unei comenzi. Ea are o metoda “execute” care face un anumit task

```
public interface DB_Command {
    void execute() throws NoDatabaseFoundException;
}
```

Spre exemplu:

```
    public saveInstance(GamePanel gamePanel, Database_Manager DB) {
        this.gamePanel = gamePanel;
        On_DB=DB;
    }
    @Override
    public void execute() throws NoDatabaseFoundException {

        if(On_DB!=null)
        {
            int x = gamePanel.player.pos.x;
            int y = gamePanel.player.pos.y;
            int cadranX = gamePanel.CadranX;
            int cadrany = gamePanel.CadranY;
            String transformation = (gamePanel.player.currentForm==
Player.form.NORMAL)?"Normal":((gamePanel.player.currentForm ==
Player.form.RHAAST) ? "Rhaast" : "Shadow_Kayn");
            int level = gamePanel.level;
            int score = gamePanel.Current_Score;
            On_DB.saveInstance(x, y, cadranX, cadrany, transformation, level,
score);
        }
        else throw new NoDatabaseFoundException("Nu am gasit baza de date");
```

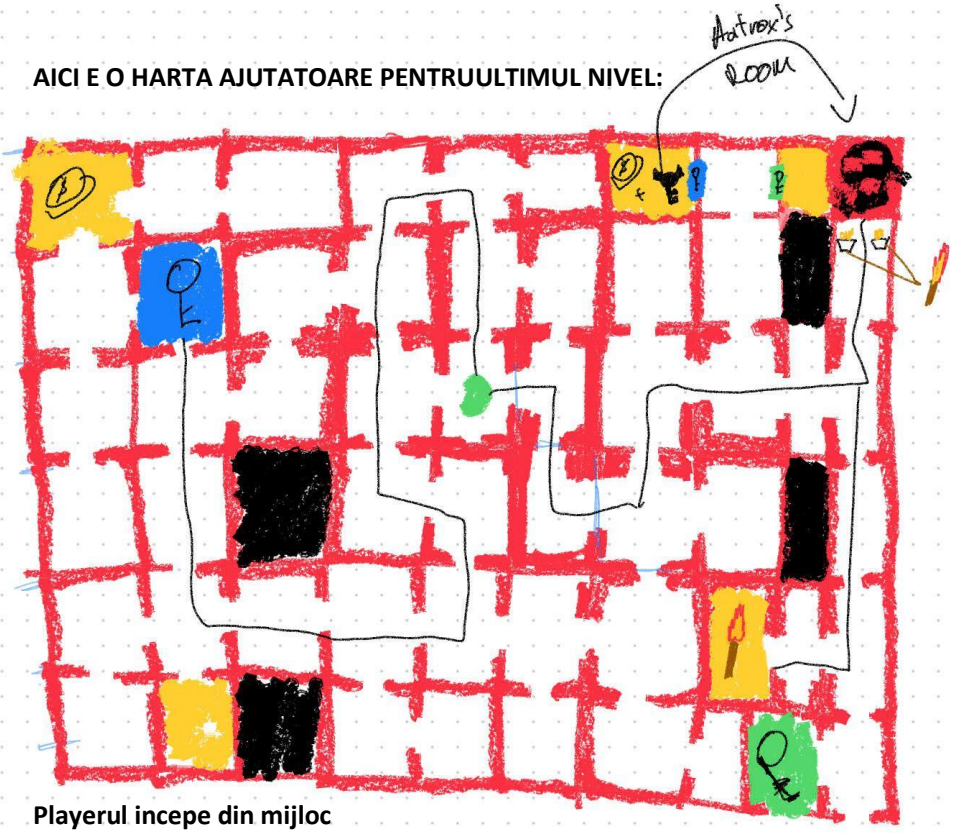
```
    }
}
```

Salveaza in baza de date, dar daca conexiunea se pierde aceasta arunca o eroare “NoDatabaseFoundException” care este tratata la randul ei.

Modelul “State” are acelasi principiu. In loc de PlayState si LoseState am PlayScene , LoseScene. O scena e un state ce contine anumite obiecte ce apar pe ecran. Scena se schimba la apasarea obiectelor.

```
if(PlayButton.isClicked()) //Daca [Scena 0 Buttonul cu ID 1] Este apasat
(PLAY) -> Schimba scena
{
    loadLevel();

    Current_Scene=1;

      ...
}
```

AICI E O HARTA AJUTATOARE PENTRUULTIMUL NIVEL:



Playerul incepe din mijloc

Algoritmi Utilizati:

- coliziuni

```
// Functia verifica daca hitboxul cuiva se suprapune cu un obiect
//Aceasta testeaza cu un hitbox mai mare decat cel dat pentru a trata
obiectele care se afla in perete, precum usile
public boolean OverlapsObjects(Rectangle hitbox, int objectIndex)
```

```
// Calculeaza fiecare colt al playerului (Hitbox) si dupa in functie de
directia in care se misca face
// un predict pentru a verifica daca playerul va lovi un perete sau nu
// Q: Care este consecinta daca nu dau predict la movement?
// A: Daca playerul se duce in perete la maxim (in sus sa zicem), acesta nu
se v-a putea misca stanga/dreapta deoarece coltul de sus atinge se afla in
perete.
//      Therefore, daca dau predict, coltul se scoate singur din perete.
(Sper ca am fost clar)
public void CheckTile(Entity entity, KeyboardHandler KeyH)
```

T.L.D.R: Coliziunile fac un predict al hitboxului in functie de tasta apasata. Daca unul din cele 4 colturi al hitboxului predicted se intersecteaza cu un obiect solid, atunci nu I permite playerului sa se deplaseze pe directia respective.

Coliziunile cu obiectele prin care poti trece se fac in aceeasi maniera, dar se ia in calcul daca playerul intentioneaza sa apuce sau nu obiectul (tasta space).

- gestiune obiecte/inventar/etc

Obiectele se afla intr-un ObjectManager si se initializeaza, unde sunt initializate toate obiectele de pe harta. Acestea se specifica daca sunt pickable sau nu si niste parametrii pentru ai pozitiona.

Obiectele Pickable se adauga in inventar (un array de obiecte pickable) si dispar atunci cand sunt ridicate. Se adauga in inventar cu ajutorul functiei de "OverlapsObject" si "Interacts_With_Object" din "CollisionManager" class.

Pe scurt, daca Obiectul se afla in acelasi cadru ca playerul si playerul se afla pe obiect si a apasat space, atunci se apeleaza .take() care scoate itemul si il adauga in inventarul playerului.

La folosirea itemului, se cauta in inventar obiectul necesar si daca e prezent, se foloseste si deblocheaza obiectul.

- deplasarea inamicilor

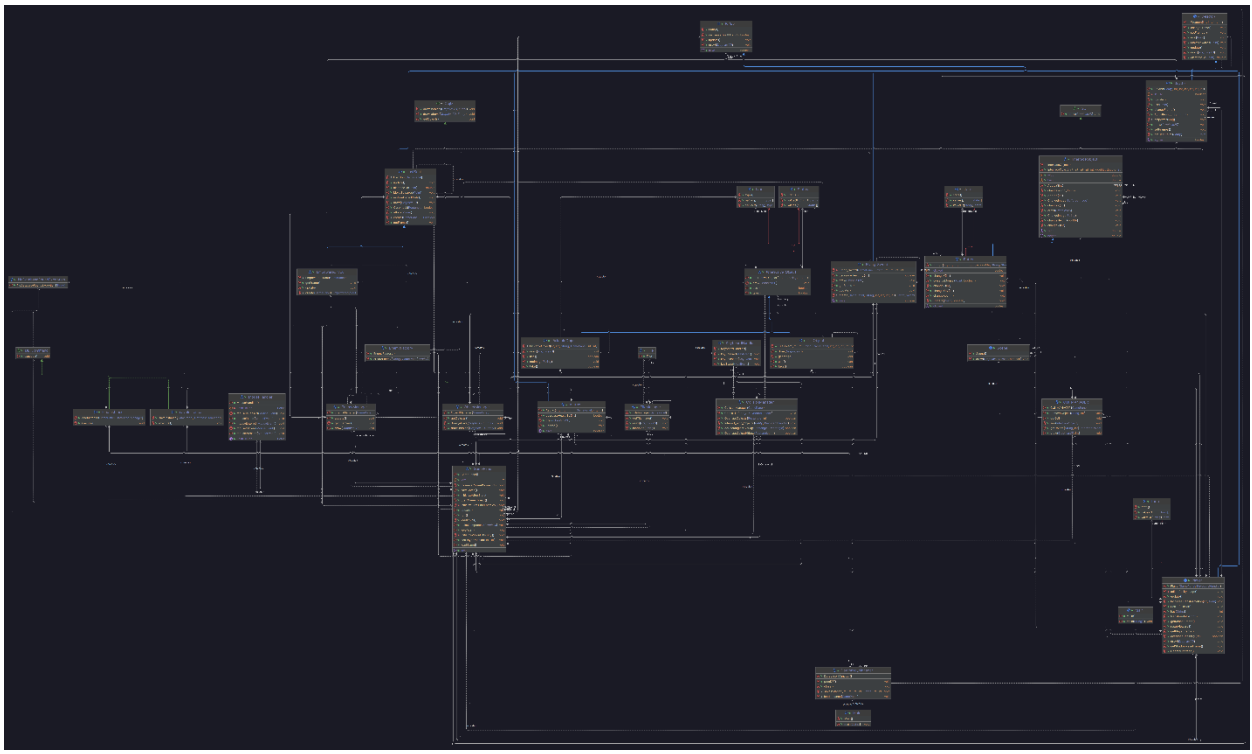
Cel mai complex inamic e "LostSoul" intrucat are metode ce calculeaza vectorul pentru a se deplasa spre player. Acesta isi ia knockback atunci cand este lovit.

```
//Oleaca de matematica si putem determina vectorul pentru directie, pe care
ulterior il scalam cu viteza

public void MoveTowards(Point go_to_point)


…
```

```
if(damaged)
{
    directionX*=-1; //Daca ii damaged, schimbam directia vectorului aka il
impingem
    directionY*=-1;
}
```

Acestia daca ataca cu success, “intra” in player, provocand daune.

Diagrama UML:



(se regaseste si in folder)

Resursele biografice utilizate bibliografie:
https://universe.leagueoflegends.com/en_GB/story/champion/kayn/
SPRITEURILE SUNT TOATE FĂCUTE DE MINE!!!! 😊
https://www.youtube.com/@RyiSnow [primele 4 episoade din playlistul how to make a java game]