

Battle City - Documentație

Membrii echipei:

- Balogh Szilárd - 10LF211
- Opra-Bódi Botond - 10LF212
- Vitályos Norbert - 10LF213

Scopul proiectului

Scopul proiectului a fost de a recrea jocul Battle City într-un mod cât mai autentic în C++ folosind biblioteca SFML. În implementarea noastră, am folosit o abordare orientată pe obiecte, dând un accent puternic principiilor design pattern-urilor: lucrul pe interfețe și nu pe implementări, folosirea compoziției în dauna moștenirii, încapsularea a ceea ce variază. De asemenea, aplicația este structurată în 3 proiecte: SFML (UI), Core (logica jocului) și Testing. Cu această structurare, se obține o delimitare clară a responsabilităților.

Separarea UI-GameLogic

Așa cum am menționat și în paragraful anterior, interfața grafică și logica jocului sunt separate. Pe partea de logică, se află doar acele componente care țin de funcționalitatea claselor: cum ar fi mișcarea tancului și a gloanței, și verificarea coliziunii. Așadar, proiectul Core nu conține nicio informație legată de cum arată aplicația.

Clasa Tank în Core:

```
namespace BattleCity::GameLogic
{
    class Tank
    {
    public:
        Tank();

        void setPosition(float x, float y);

        [[nodiscard]] float getXPosition() const;

        [[nodiscard]] float getYPosition() const;

        [[nodiscard]] int getTankSpeed() const;

    private:
        float m_xPos, m_yPos;
        int m_tankSpeed;
        GameConfig::MoveDirection m_tankDirection;
    public:
        [[nodiscard]] GameConfig::MoveDirection getTankDirection() const;

        void setTankDirection(GameConfig::MoveDirection mTankDirection);

    };
}
```

Se reține poziția, direcția și viteza tancului și se observă metodele aferente.

Clasa GameLogic în Core:

```

7 namespace BattleCity::GameLogic
8 {
9     class GameLogic
10    {
11    public:
12
13        void checkCollision();
14
15    private:
16
17        void checkEnemyTankPlayerBulletCollision();
18        void checkPlayerTankEnemyBulletCollision();
19        void checkPlayerBulletEnemyBulletCollision();
20        void checkTankTileCollision();
21        void checkBulletTileCollision();
22
23        bool isCollision(int x1, int x2, int width1, int width2, int y1, int y2, int height1, int height2);
24
25        std::vector<Tank> m_enemyTanks;
26        std::vector<Tank> m_playerTanks;
27        std::vector<Bullet> m_playerBullets;
28        std::vector<Bullet> m_enemyBullets;
29
30        Map m_map;
31    };
32 }

```

Clasa GameLogic conține tancurile, gloanțele și harta pentru a verifica coliziunile.
Clasa SFMLTank:

```

15 namespace BattleCity::SFML
16 {
17     class SFMLTank : public sf::Drawable, public sf::Transformable
18     {
19     public:
20         using OnBulletShotCallback = std::function<void(const SFMLTank&, SFMLBullet&& bullet)>;
21
22         OnBulletShotCallback onBulletShot;
23
24         void setOnBulletShot(OnBulletShotCallback&& callback);
25
26         void shootBullet();
27
28         explicit SFMLTank(TextureManager& textureManager);
29
30         void draw(sf::RenderTarget& target, sf::RenderStates states) const override;
31
32         // this function moves the tank relative to its current position
33         void moveTank(float x, float y);
34
35         GameConfig::MoveDirection getMoveDirection();
36         void setMoveDirection(GameConfig::MoveDirection direction);
37     private:
38         void initTankTexture();
39
40         TextureManager& m_textureManager;
41
42         GameLogic::Tank m_tankModel;
43
44         sf::Texture& m_tankTexture;
45         sf::Sprite m_tankSprite;
46     };

```

Reține textura, sprite-ul și funcțiile care țin de grafică precum draw sau initTankTexture. Se remarcă compoziția cu tancul din Core.

Design patternuri utilizate

În ingineria software, un design pattern este o soluție generală, reutilizabilă la o problemă care apare frecvent în proiectarea software. Am folosit mai multe patternuri, cum ar fi strategy, pentru a efectua mișcările gloanței în direcții diferite, factory, respectiv observer ca Game să fie notificat atunci când un tanc a tras și, prin urmare, Game trebuie să verifice dacă există coliziuni.

Unit testing

Unit testing este o metodologie de testare a software-ului în care unitățile individuale sau componentele unui software sunt testate izolat de restul aplicației, pentru a se asigura că fiecare parte funcționează corect. În aplicația noastră, am testat cele mai esențiale părți ale proiectului, cum ar fi mișcarea gloanței și a tancului. De asemenea, am utilizat mocking pentru a verifica dacă un nivel, stocat în fișier ca matrice, se încarcă corect.

Test pentru a verifica mișcarea gloanței:

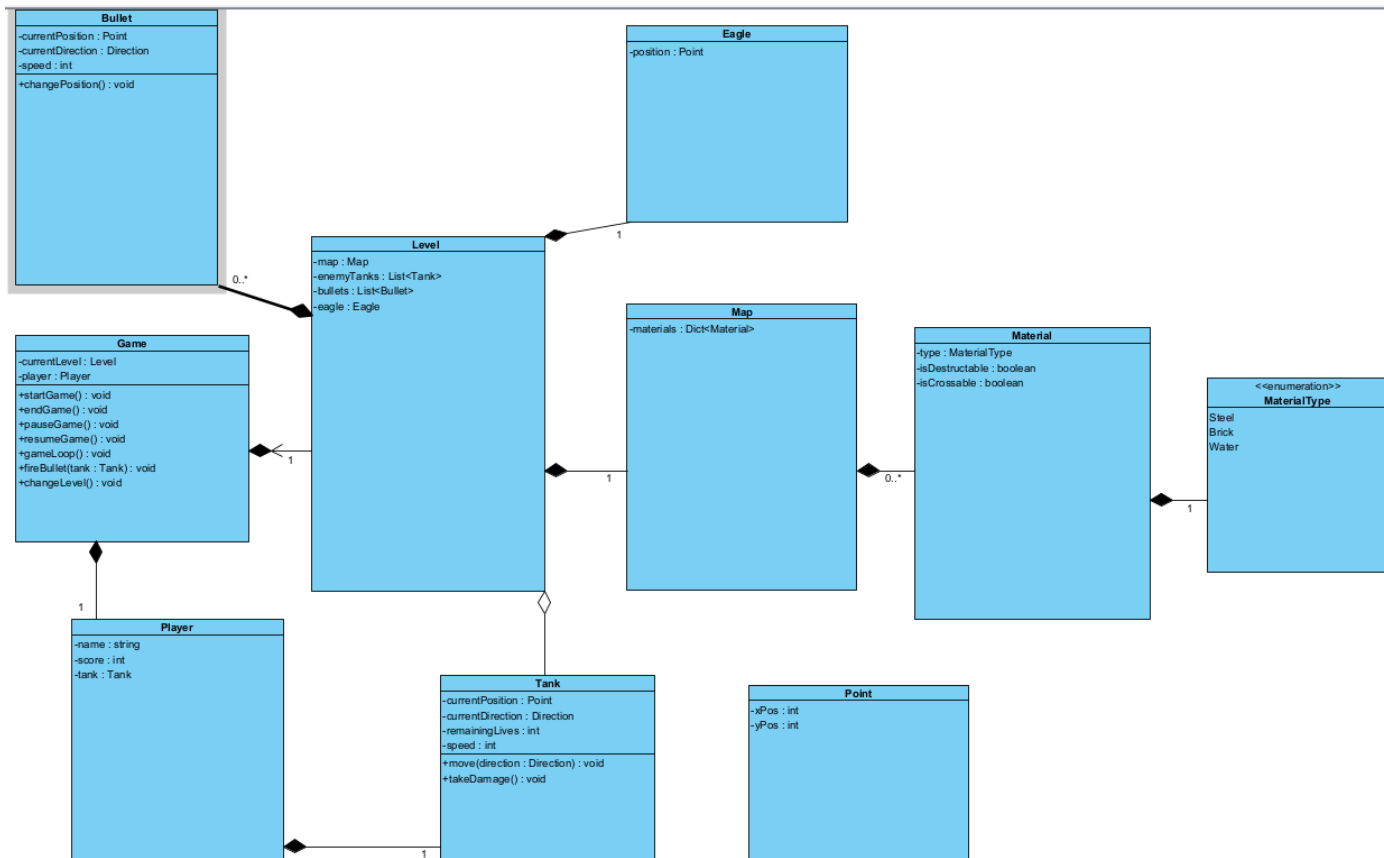
```
19 TEST(BulletTest, BulletMoveMultiple) {
20     auto originalPosition = Position{ 20, 30 };
21     Bullet bullet{ originalPosition, GameConfig::MoveDirection::DOWN, Bullet::BulletType::PlayerBullet };
22
23     bullet.move();
24     bullet.move();
25     bullet.move();
26
27     auto bulletSpeed = bullet.getSpeed();
28     auto expectedPosition = Position{ originalPosition.x, originalPosition.y + 3 * bulletSpeed };
29
30     EXPECT_EQ(bullet.getPosition().x, expectedPosition.x);
31     EXPECT_EQ(bullet.getPosition().y, expectedPosition.y);
32 }
33
34 TEST(BulletTest, BulletDirection) {
35     Bullet bullet{ Position{0, 0}, GameConfig::MoveDirection::UP, Bullet::BulletType::PlayerBullet };
36
37     EXPECT_EQ(bullet.getDirection(), GameConfig::MoveDirection::UP);
38 }
```

Mocking pentru a verifica harta:

```
42 class MockFileHandler : public BattleCity::FileHandler {
43 public:
44     MOCK_METHOD(BattleCity::Matrix<int>, getMapData, ());
45 };
46
47 TEST(MapTest, MockedFileHandler) {
48     MockFileHandler fileHandler;
49
50     ON_CALL(fileHandler, getMapData())
51         .WillByDefault(Return(std::vector{ std::vector{0, 0}, std::vector{1, 2}, std::vector{1, 2}, }));
52
53     BattleCity::GameLogic::Map map{ fileHandler.getMapData() };
54
55     EXPECT_EQ(map.getWidth(), 2);
56     EXPECT_EQ(map.getHeight(), 3);
57     EXPECT_EQ(map.at(0, 1), 0);
58 }
```

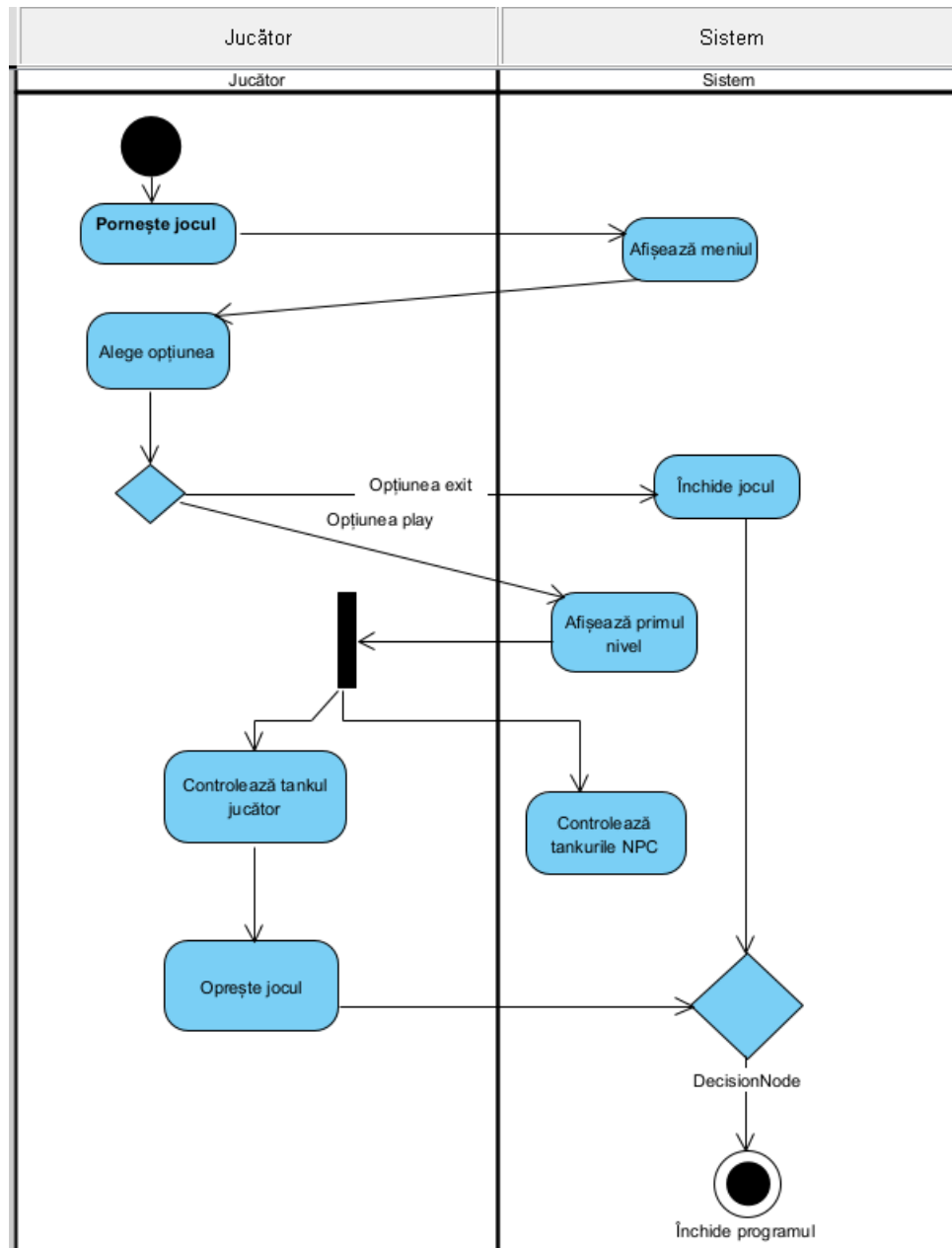
Diagrame

Class Diagram



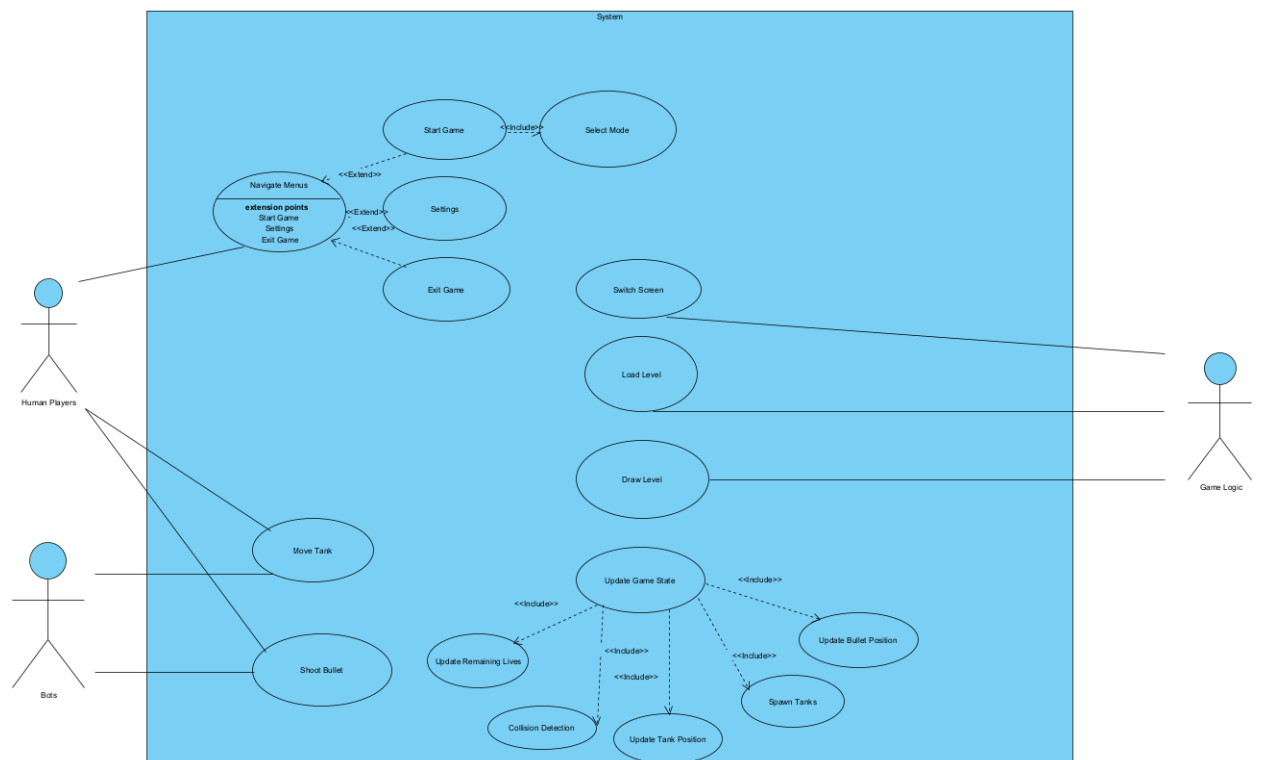
O diagramă de clase oferă o vedere statică, concentrându-se pe structura sistemului. În cazul aplicației noastre, conturează arhitectura jocului, delimitând relațiile și rolurile diferitelor entități din sistem. Clasele centrale precum **Player**, **Game**, **Map**, **PlayerTank** și **EnemyTank** formează coloana vertebrală a structurii sistemului. Clasa **Player** servește ca punct de intrare, interacționând cu clasa **Game**, care încapsulează funcționalitatea de bază prin metode precum `startGame()` și `loadLevel()`. **PlayerTank** și **EnemyTank** sunt subclase care moștenesc din clasa **Tank**, permițând comportamente polimorfe în joc.

Activity Diagram

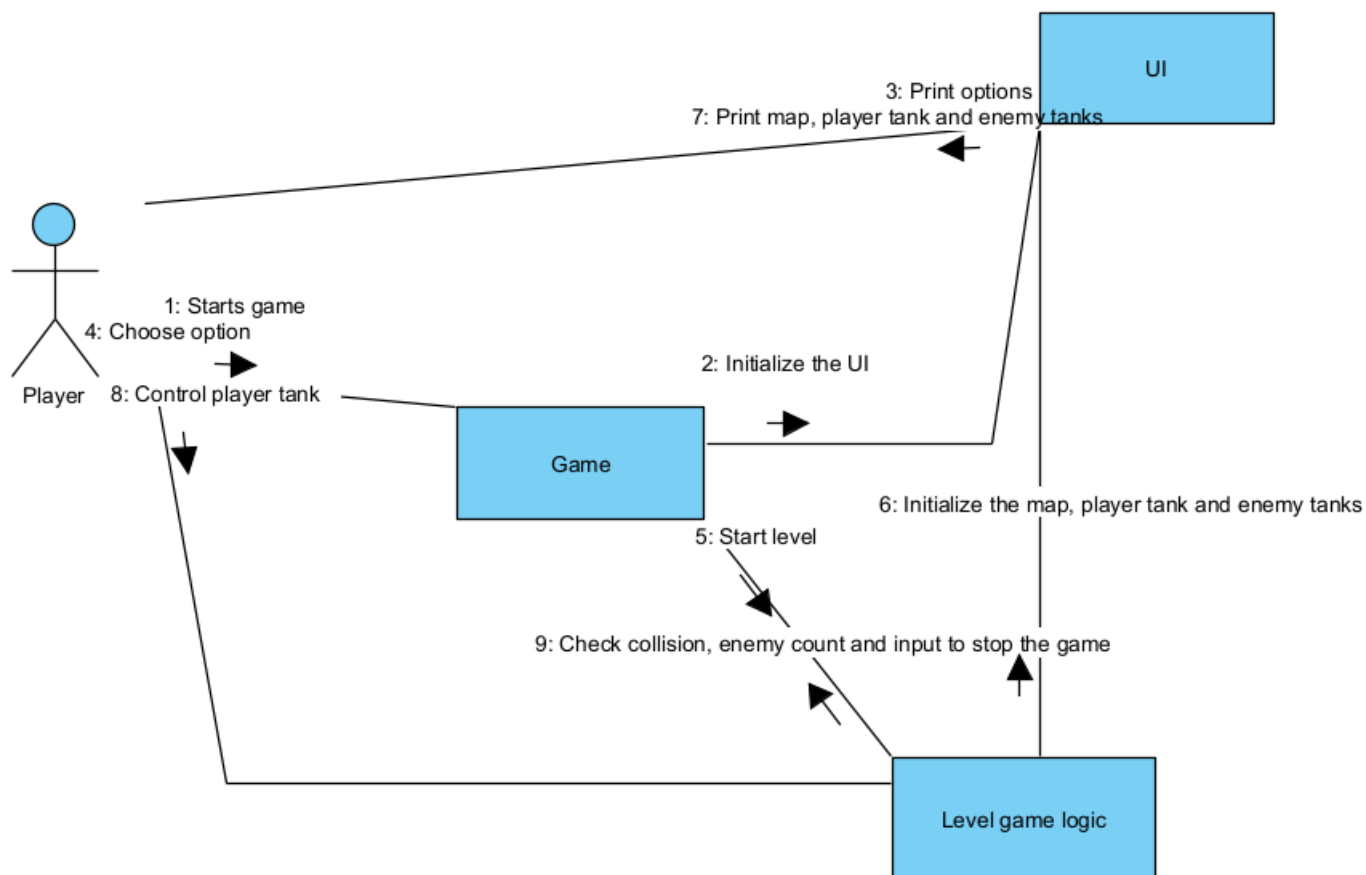


O diagramă de activități oferă o vedere dinamică a sistemului, concentrându-se pe fluxul de control și operațiuni. În contextul aplicației noastre, diagrama activității ilustrează fluxul acțiunilor din perspectiva jucătorului. Procesul începe atunci când jucătorul lansează aplicația. După pornirea jocului, este prezentată o interfață de meniu, care oferă două opțiuni principale: pornirea jocului sau ieșirea din aplicație. Dacă jucătorul optează pentru începerea jocului propriu-zis, fluxul trece la primul nivel al jocului, unde jucătorul poate prelua controlul tankului său, în timp ce sistemul controlează tankurile inamice. Alegerea de a părăsi jocul în orice moment duce la închiderea aplicației.

Usecase Diagram



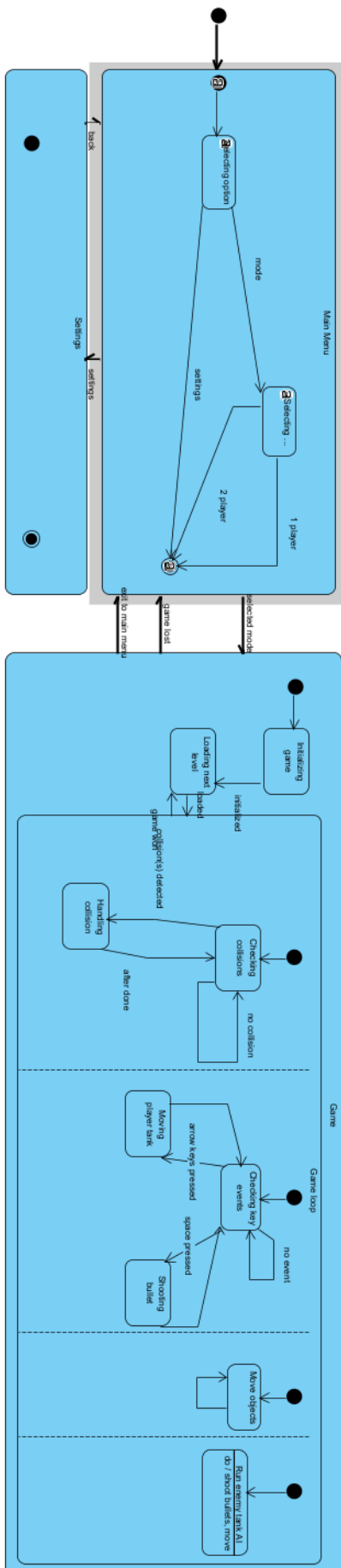
Collaboration Diagram



O diagramă de colaborare ilustrează modul în care obiectele interacționează într-

un context. La începerea jocului, jucătorului i se prezintă un UI care solicită alegerea opțiunilor de joc. Logica jocului este inițializată odată ce o opțiune este selectată, ceea ce duce la configurarea mediului de joc, care include inițializarea hărții și apariția atât a tancului jucătorului, cât și a tancurilor inamice. Jucătorul intră apoi în bucla de joc și se angajează în funcțiile de bază ale jocului, cum ar fi mișcarea tancului și tragerea cu glonțul. Simultan, logica jocului este responsabilă pentru monitorizarea coliziunilor, urmărirea unităților inamice și procesarea inputului jucătorului pentru a continua sau opri jocul.

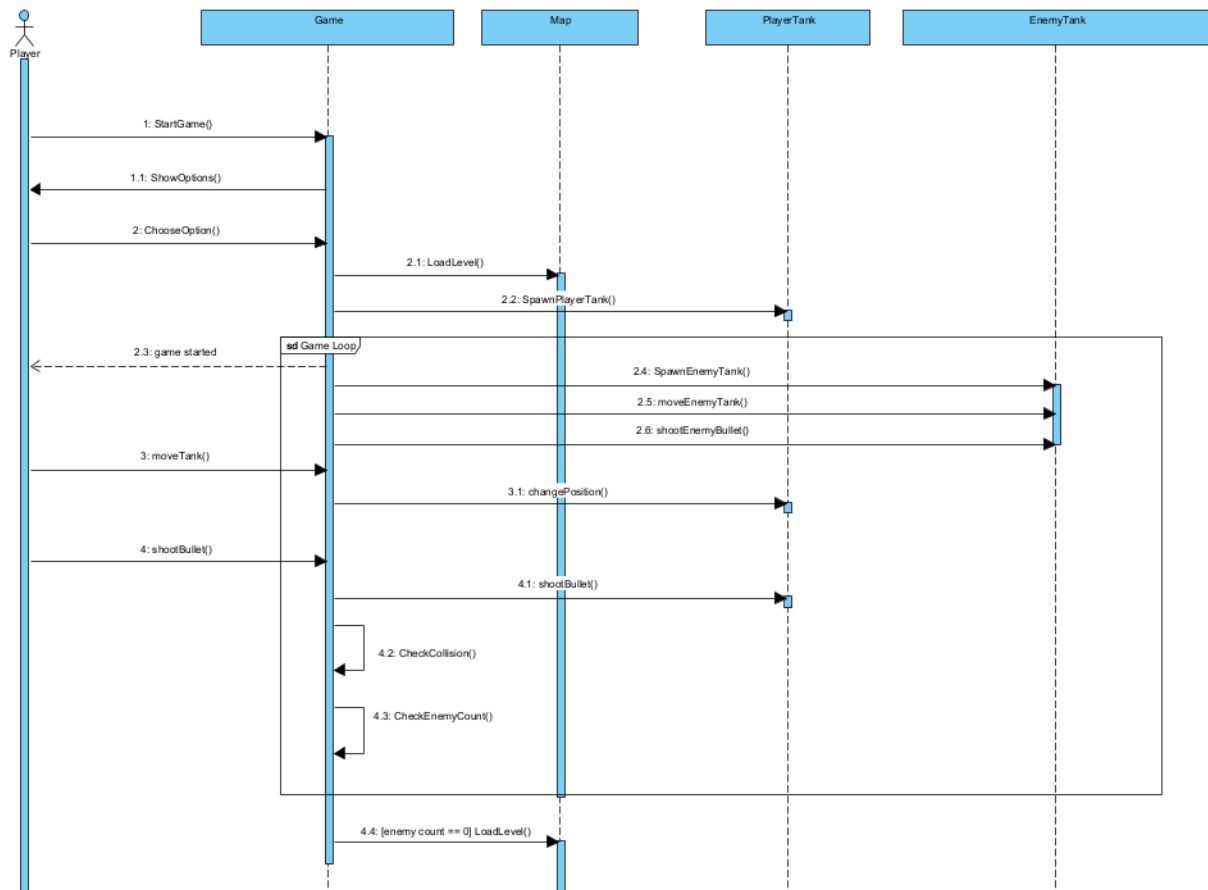
Statechart Diagram



O diagramă de stări descrie schimbările de stare ale unui sistem sau ale unui obiect ca

răspuns la evenimente. În cadrul aplicației noastre, prima stare este cea de "Main Menu", unde inputul jucătorului determină fluxul, fie prin începerea jocului propriu-zis, fie prin modificarea setărilor. Trecerea de la "Initializing Game" la "Loading next level" denotă etapele prin care trece sistemul înainte de a implica jucătorul în jocul propriu-zis. Stările "Game Loop" și "Checking Collisions" funcționează în tandem, ilustrând capacitatea de răspuns a sistemului. În cadrul buclei de joc, stările precum "Checking key events", "Moving Player Tank" și "Running Enemy Tank AI" reflectă natura interactivă și paralelă a jocului.

Sequence diagram



O diagramă de secvențe detaliază interacțiunile obiectelor într-o secvență de timp. Diagrama de secvență prezentată oferă o imagine cuprinzătoare a interacțiunilor dintre jucător și componentele sistemului de joc. Aceste componente sunt: jocul, harta, tancul jucător și tancul inamic. Diagrama reprezintă fiecare pas de la începerea jocului. Prima oară, jucătorul trebuie să aleagă una dintre opțiunile prezentate de joc. După aceea, se afișează primul nivel, are loc spawn-ul tancului jucător și începe bucla jocului. Bucla jocului include acțiuni, precum SpawnEnemyTank, MoveEnemyTank, ShootEnemyTank, CheckCollision și CheckEnemyCount. Dacă numărul tancurilor inamice scade la 0, înseamnă că jucătorul a câștigat nivelul respectiv și poate trece la următorul nivel.