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 structură, 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 glonțului și verificarea coliziunii. Așadar, proiectul Core nu conține nicio informație legată de cum arată aplicația.

Clasa tanc 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:

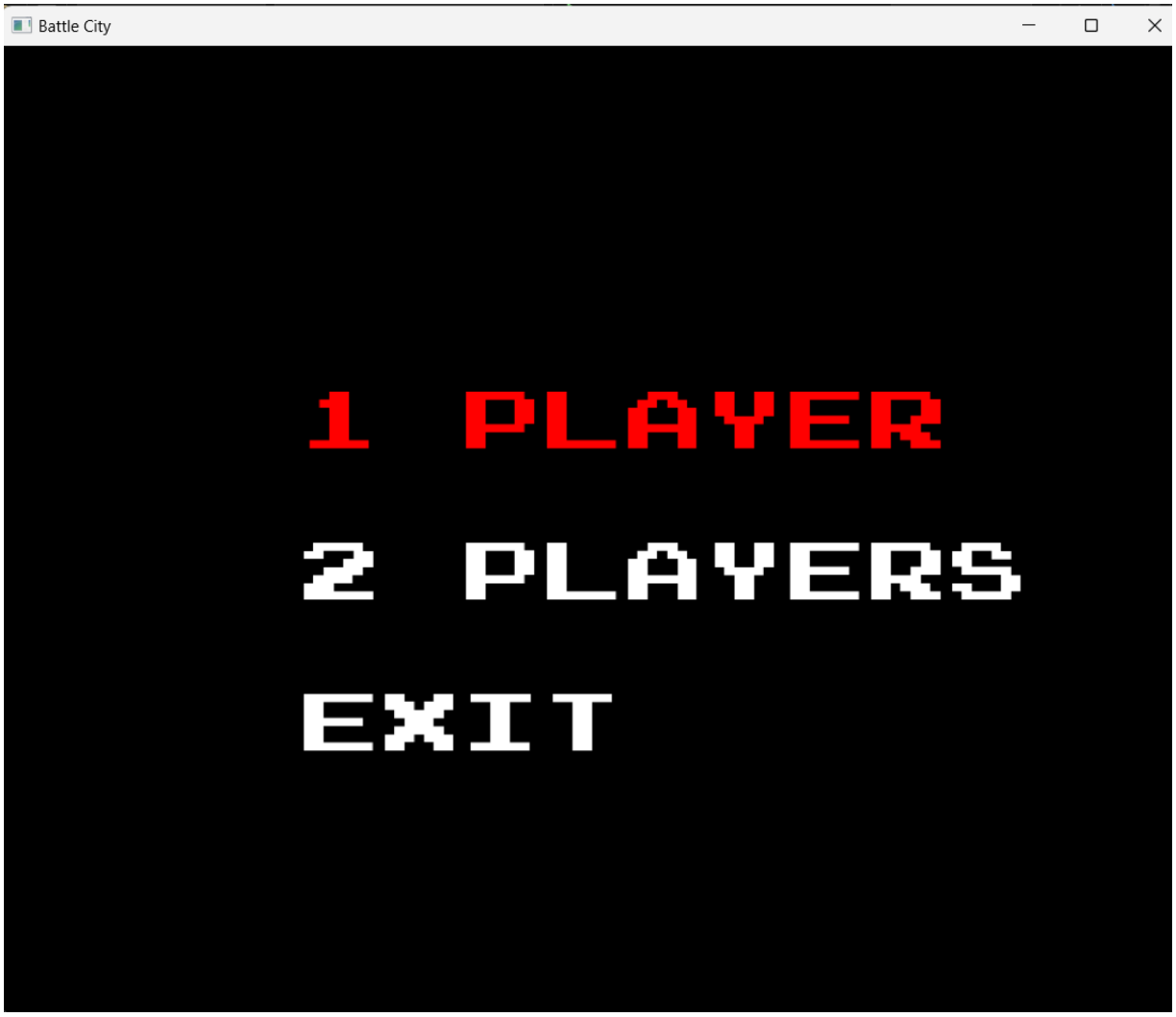
```cpp
namespace BattleCity::GameLogic
{
    class GameLogic
    {
    public:

        void checkCollision();

    private:

        void checkEnemyTankPlayerBulletCollision();
        void checkPlayerTankEnemyBulletCollision();
        void checkPlayerBulletEnemyBulletCollision();
        void checkTankTileCollision();
        void checkBulletTileCollision();

        bool isCollision(int x1, int x2, int width1, int width2, int y1, int y2, int height1, int height2);

        std::vector<Tank> m_enemyTanks;
        std::vector<Tank> m_playerTanks;
        std::vector<Bullet> m_playerBullets;
        std::vector<Bullet> m_enemyBullets;

        Map m_map;
    };
}
```

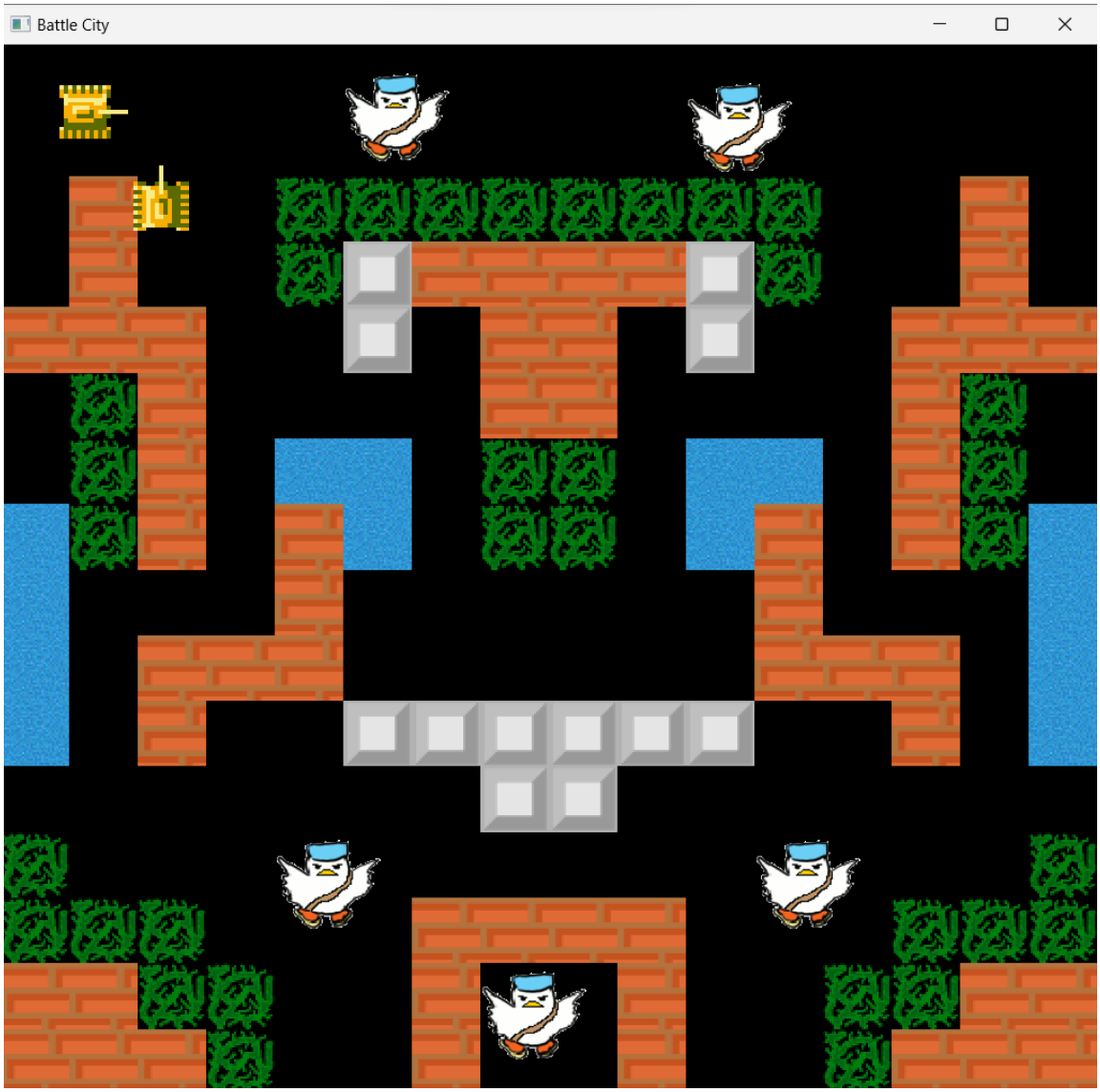
Clasa GameLogic conține tancurile, gloanțele și harta pentru a verifica coliziunile.
Clasa SFMLtanc:

```cpp
namespace BattleCity::SFML
{
    class SFMLTank : public sf::Drawable, public sf::Transformable
    {
    public:
        using OnBulletShotCallback = std::function<void(const SFMLTank&, SFMLBullet&& bullet)>;

        OnBulletShotCallback onBulletShot;

        void setOnBulletShot(OnBulletShotCallback&& callback);

        void shootBullet();

        explicit SFMLTank(TextureManager& textureManager);

        void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

        // this function moves the tank relative to its current position
        void moveTank(float x, float y);

        GameConfig::MoveDirection getMoveDirection();
        void setMoveDirection(GameConfig::MoveDirection direction);
    private:
        void initTankTexture();

        TextureManager& m_textureManager;

        GameLogic::Tank m_tankModel;

        sf::Texture& m_tankTexture;
        sf::Sprite m_tankSprite;
    };
```

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

UI-ul aplicației:


Battle City
1 PLAYER
2 PLAYERS
EXIT



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 glonțului în direcții diferite, factory, respectiv observer pentru a notifica clasa Game atunci când un tanc a tras și, prin urmare, Game trebuie să verifice coliziunile.

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 esențiala părți ale proiectului, cum ar fi mișcarea glonțului 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 glonțului:

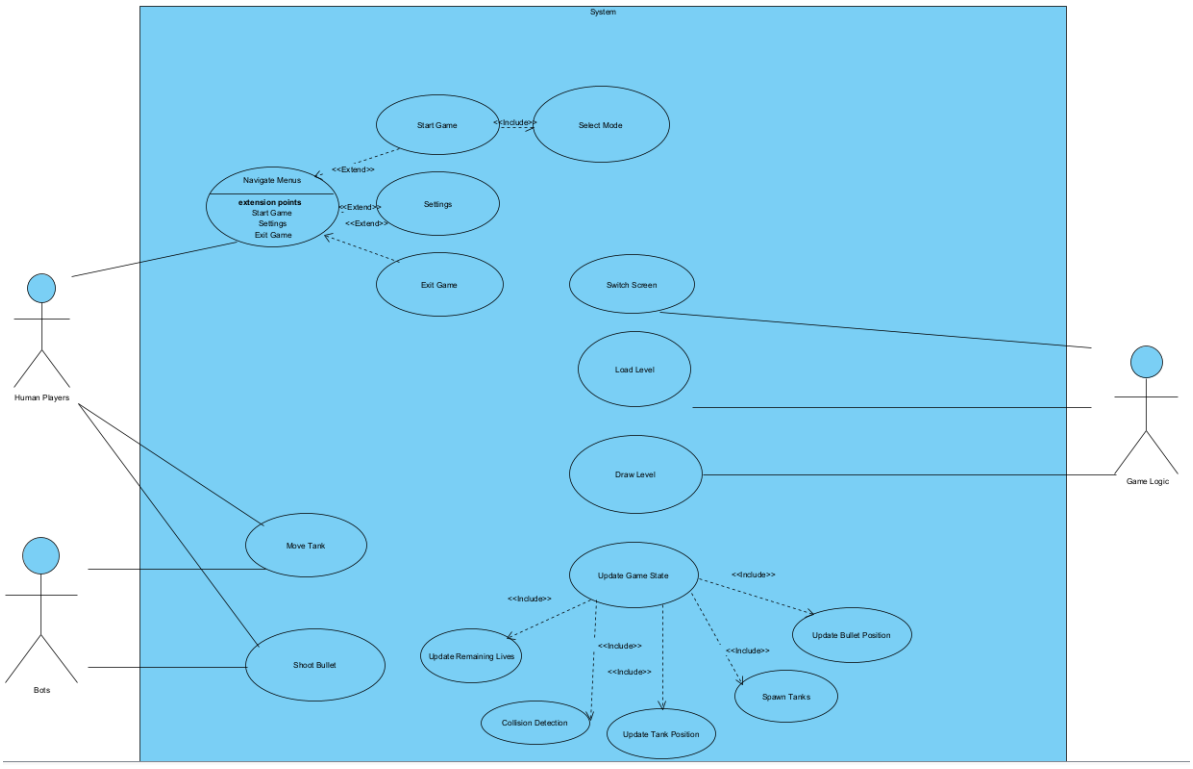
```cpp
TEST(BulletTest, BulletMoveMultiple) {
    auto originalPosition = Position{ 20, 30 };
    Bullet bullet{ originalPosition, GameConfig::MoveDirection::DOWN, Bullet::BulletType::PlayerBullet

    bullet.move();
    bullet.move();
    bullet.move();

    auto bulletSpeed = bullet.getSpeed();
    auto expectedPosition = Position{ originalPosition.x, originalPosition.y + 3 * bulletSpeed };

    EXPECT_EQ(bullet.getPosition().x, expectedPosition.x);
    EXPECT_EQ(bullet.getPosition().y, expectedPosition.y);
}

TEST(BulletTest, BulletDirection) {
    Bullet bullet{ Position{0, 0}, GameConfig::MoveDirection::UP, Bullet::BulletType::PlayerBullet };

    EXPECT_EQ(bullet.getDirection(), GameConfig::MoveDirection::UP);
}
```

Mocking pentru a veri ca harta:

```cpp
class MockFileHandler : public BattleCity::FileHandler {
public:
    MOCK_METHOD(BattleCity::Matrix<int>, getMapData, ());
};

TEST(MapTest, MockedFileHandler) {
    MockFileHandler fileHandler;

    ON_CALL(fileHandler, getMapData())
        .WillByDefault(Return(std::vector{ std::vector{0 , 0}, std::vector{1 , 2}, std::vector{1 , 2}, }));

    BattleCity::GameLogic::Map map{ fileHandler.getMapData() };

    EXPECT_EQ(map.getWidth(), 2);
    EXPECT_EQ(map.getHeight(), 3);
    EXPECT_EQ(map.at(0, 1), 0);
}
```

Diagrame

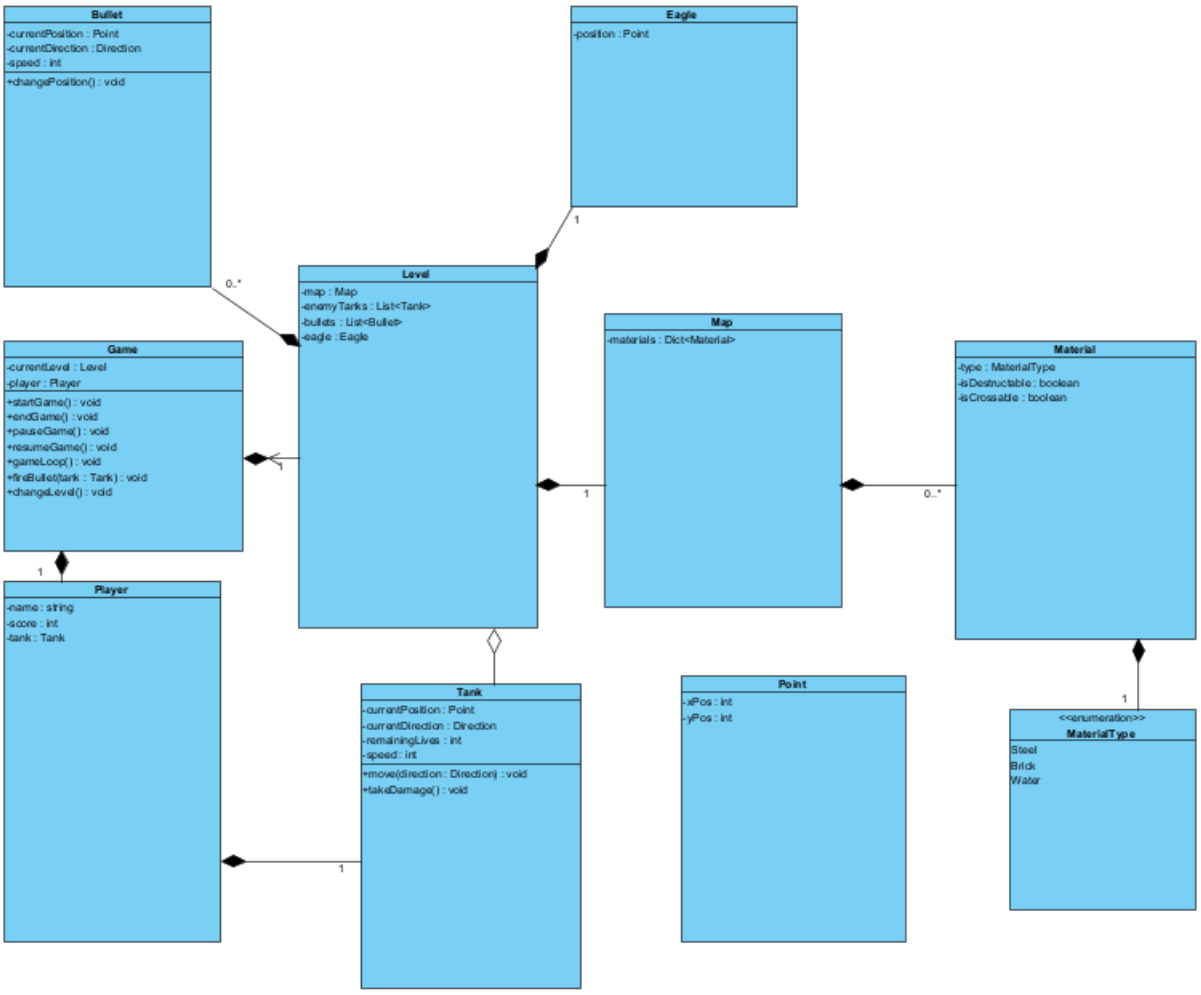
Usecase Diagram



O diagramă usecase reprezintă funcționalitatea unui sistem folosind actori și cazuri

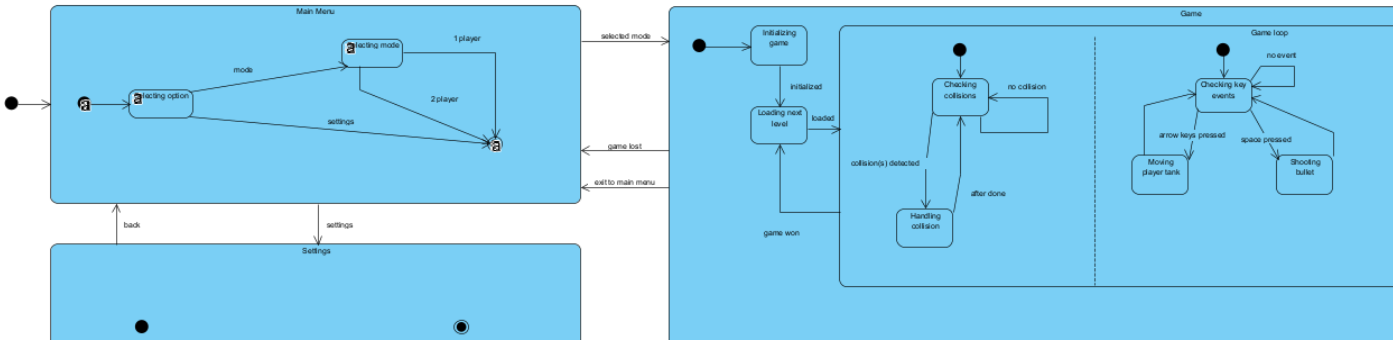
de utilizare. În contextul aplicației noastre, ea subliniază în mod distinct interacțiunile dintre jucătorul uman, AI și sistemul general de joc. Jucătorul este asociat cu cazuri de utilizare precum „Start Game", „Select Level" și „Exit Game", indicând acțiunile directe pe care le pot efectua în cadrul sistemului. În mod similar, implicarea AI este caracterizată de cazuri de utilizare precum „Control Enemies" și „Load Next Level", reflectând rolul său în provocarea jucătorului și progresul jocului. Sistemul în sine este implicat în sarcini fundamentale precum „Inițializare joc" și „Nivel de încărcare", care sunt esențiale pentru configurarea mediului de joc.

Class Diagram



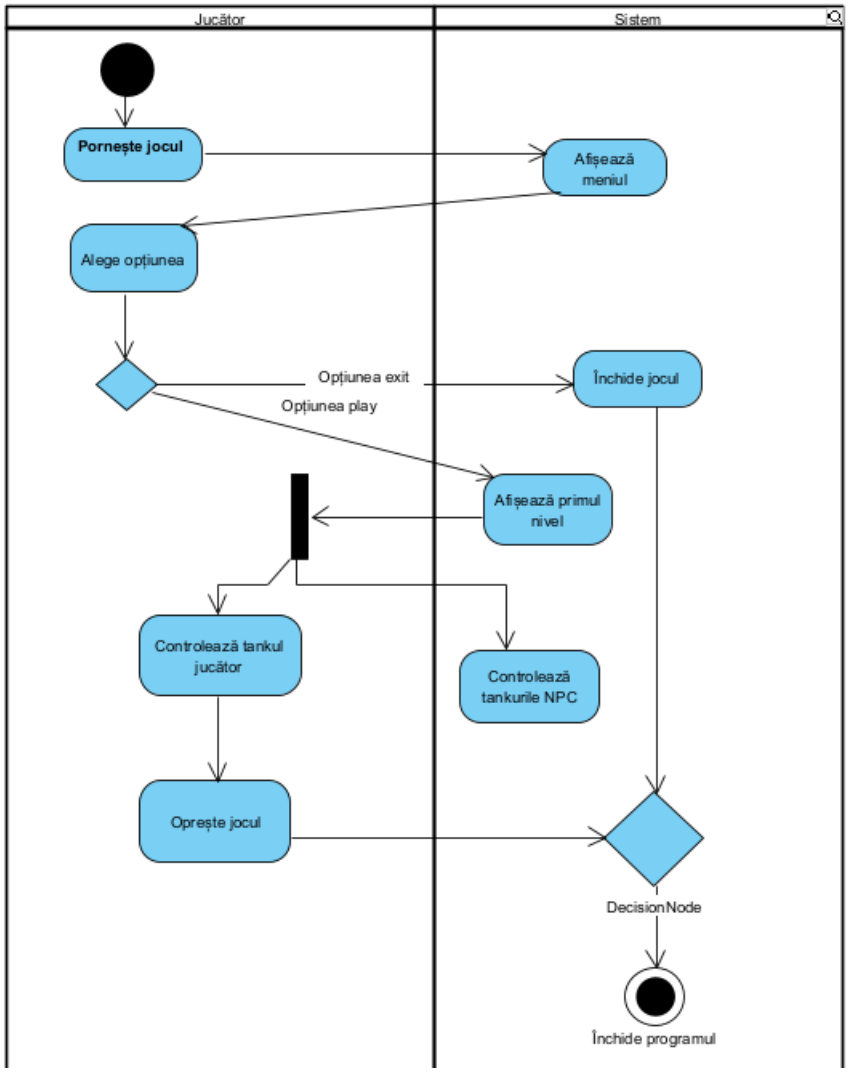
O diagramă de clase oferă o vedere statică, concentrându-se pe structura sistemului. În cazul aplicației noastre, conturează arhitectura jocului, delimitând relațiile și rolurile diferitelor entități din sistem. Clasele centrale precum Player, Game, Map, Playertanc și Enemytanc formează coloana vertebrală 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(). Playertanc și Enemytanc sunt subclase care moștenesc din clasa tanc, permițând comportamente polimorfe în joc.

Statechart Diagram



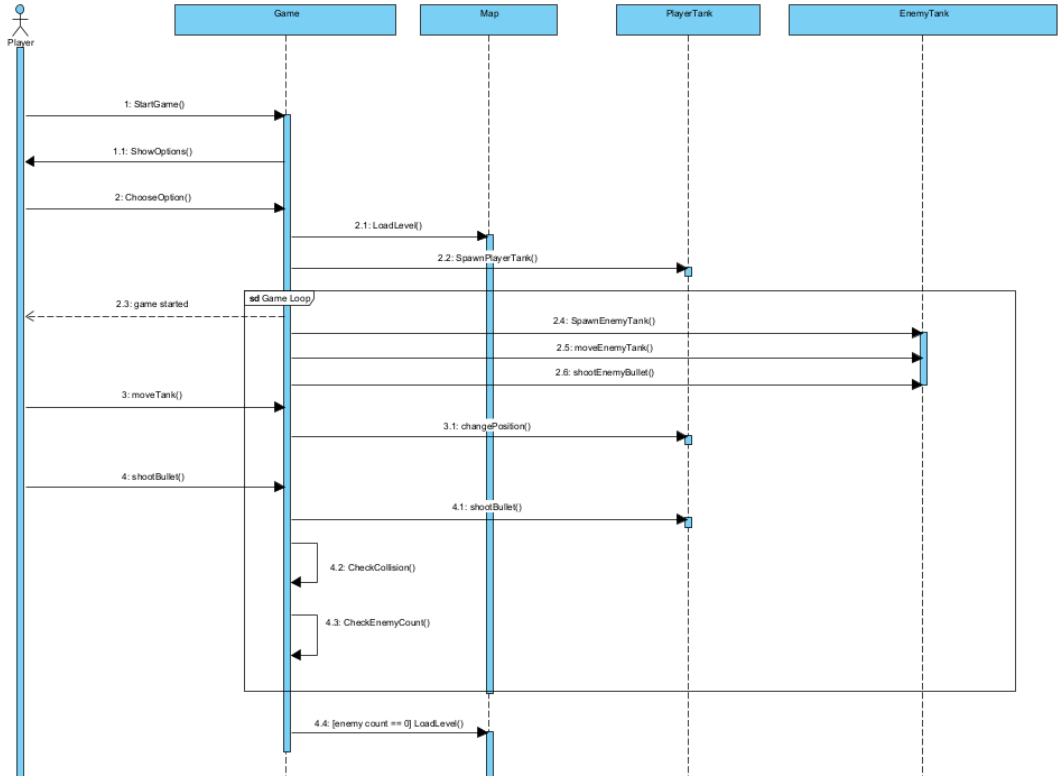
O diagramă de stări descrie schimbările de stare ale unui sistem sau ale unui obiect ca răspuns la evenimente. În cadrul aplicației noastre, prima stare este cea de "Main Menu", unde inputul jucătorului determină fluxul, fie prin începerea jocului propriu-zis, fie prin modificarea setărilor. Trecerea 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 tanc" și "Running Enemy tanc AI" reflectă natura interactivă și paralelă a jocului.

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 tancului său, în timp ce sistemul controlează tancurile inamice. Alegerea de a părăsi jocul în orice moment duce la închiderea aplicației.

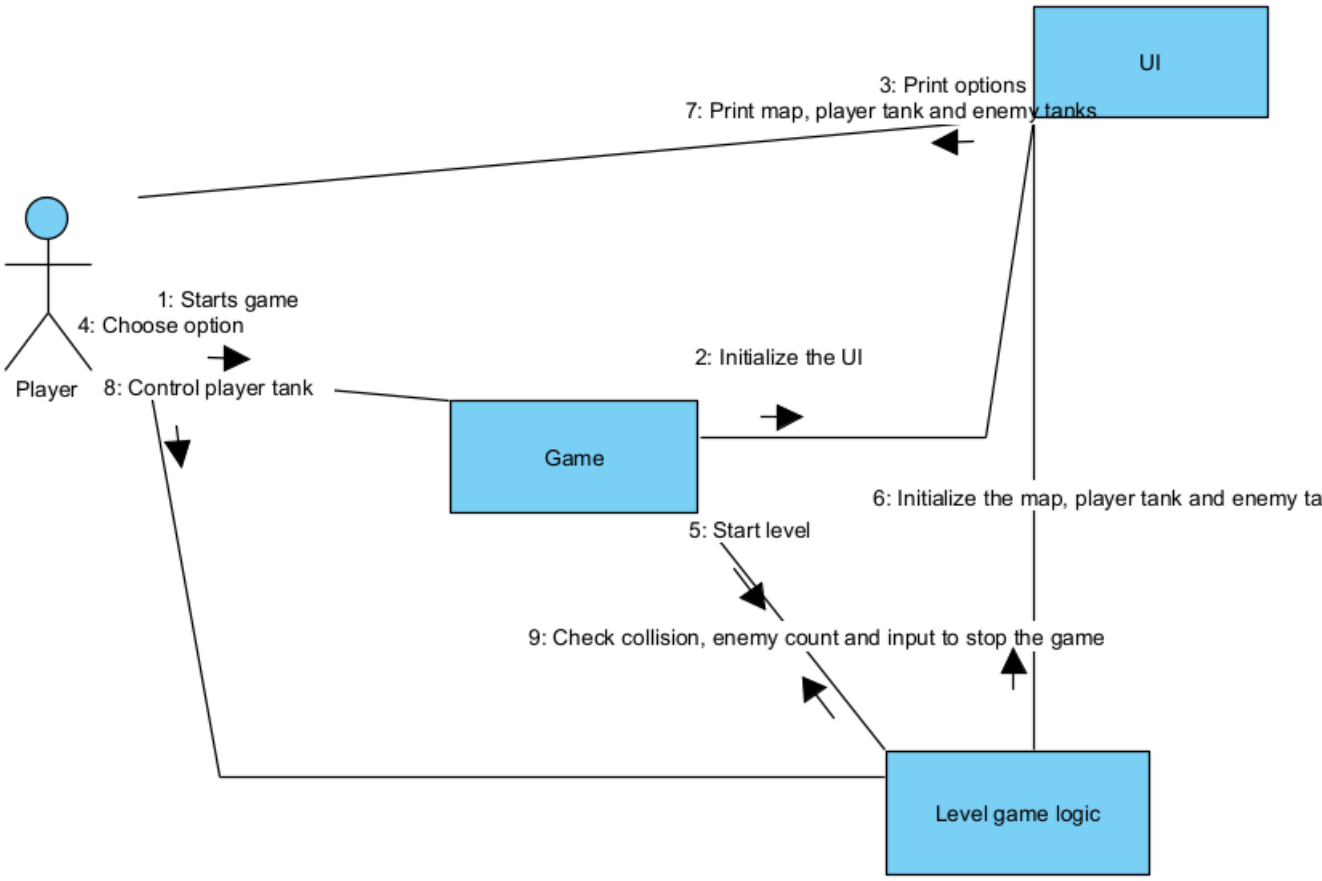
Sequence Diagram



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 SpawnEnemytanc, MoveEnemytanc, ShootEnemytanc, CheckCollision și CheckEnemyCount. Dacă numărul tancurilor inamice scade la 0, înseamnă că jucătorul a câștigat nivelul respectiv și poate trece la următorul nivel.

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.



Concluzii

Scrierea acestui proiect ne-a ajutat să ne dezvoltăm ca programatori, fiindcă proiectul ne-a ridicat numeroase provocări. De exemplu: am folosit CMake pentru a face aplicația cross-platform, am învățat folosul diagramelor și a design pattern-urilor în cadrul ingineriei software, am învățat cum se scrie o funcție pentru detectarea coliziunii. Cel mai important lucru, însă, pe care l-am învățat este cât de importantă este proiectarea înainte de a începe dezvoltarea propriu-zisă a aplicației.