

Bomberman

Echipă:

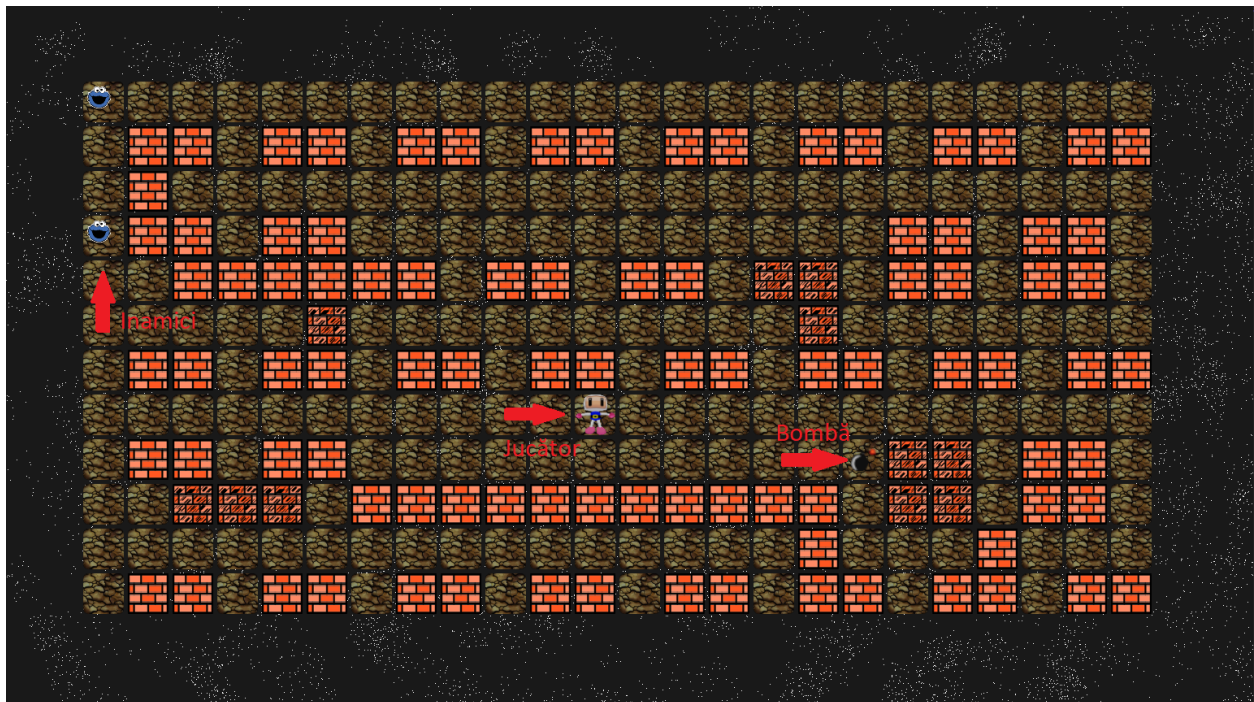
- Bardaş Denis Adelin
- Brănaci Șerban Mihai
- Chindea Cosmin Mihai

GitHub:

<https://github.com/Serban8/Bomberman>

Descriere

Echipa noastră a ales să realizeze jocul Bomberman. Scopul jocului este ca jucătorul să elimine inamicii prezenți pe hartă și să avanseze la nivelul următor. El folosește bombele pentru a sparge blocurile care stau în calea sa, cât și pentru a distruge adversarii.




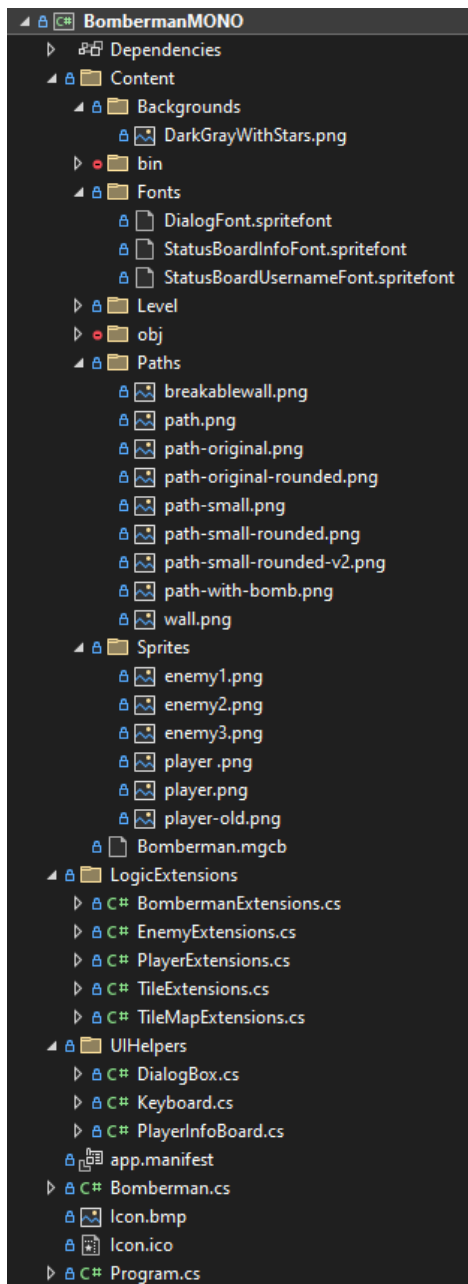
Harta unui nivel al jocului. Am evidențiat principalele elemente: jucătorul, bomba și inamicii.

Codul proiectului a fost scris în limbajul C#, folosind IDE-ul Visual Studio.

Soluția de Visual Studio cuprinde 3 proiecte distincte, pentru fiecare componentă a jocului: BombermanBase pentru logică, BombermanMONO pentru partea de UI și BombermanTests pentru partea de testare.

UI

Pentru realizarea părții de UI am folosit framework-ul MonoGame  Acesta are diverse avantaje, ușurând organizarea și afișarea elementelor grafice care constituie mediul de joc pentru utilizator.



Proiectul responsabil cu afișarea interfeței este de tip MonoGame Desktop Application și are o structură specifică MonoGame:

În folderul Content se afla texturile, fonturile și celelalte asset-uri folosite pentru interfața grafică

LogicExtensions conține metode de extensie ale interfețelor din DLL-ul cu logica aplicației, pentru a putea afișa fiecare element și update logica aferentă UI-ului

UIHelpers conține diverse clase specifice MonoGame create pentru a respecta principiile clean code și pentru a face cod reutilizabil și mentenabil

Clasa Bomberman.cs este "managerul" părții de UI, centralizând informațiile și instrucțiunile specifice UI.



Sprite-urile folosite pentru afișarea jocului.

Framework-ul Monogame dispune de funcții și clase speciale pentru afișarea pe ecran a resurselor. În proiect, aceste clase și funcții sunt utilizate în principal în clasa Bomberman.cs, care administrează interfața grafică a jocului, dar și în alte locuri. Această clasă se găsește în cadrul proiectului BombermanMONO, care este proiectul de front-end al jocului. În acesta am folosit clase specifice precum **GraphicsDeviceManager**, **SpriteBatch** și **Vector2**.

La deschiderea jocului, resursele sunt încărcate de pe disk. Apoi, folosind funcțiile specifice MonoGame, elementele sunt prelucrate și afișate pe ecran.

```
private GraphicsDeviceManager _graphics;
private SpriteBatch _spriteBatch;

private readonly Vector2 _windowSize;
private Texture2D _backgroundTexture;
```

```
protected void LoadLevels()
{
    List<string> levels = new()
    {
        ".\\Content\\Level\\Level1.txt",
        ".\\Content\\Level\\Level2.txt",
        ".\\Content\\Level\\Level3.txt"
    };

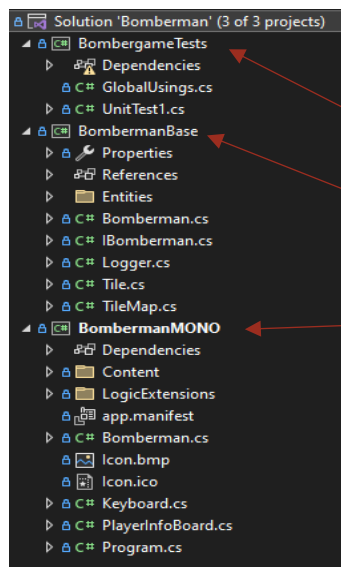
    //load the levels
    foreach (var levelPath in levels)
    {
        var level = TileMapExtensions.CreateMap(_windowBorderSize, _windowSize, levelPath);
        _game.AddLevel(level);
    }
}
```

Funcțiile pentru încărcarea resurselor.

Back-End

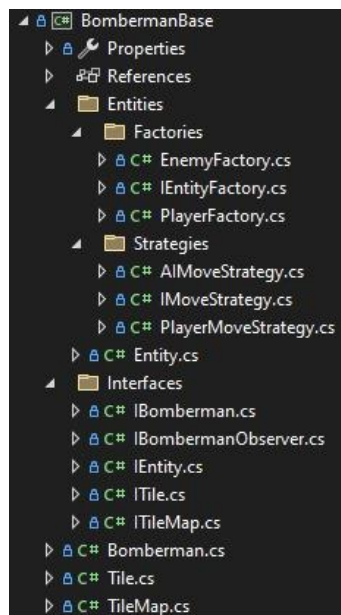
Proiectul respecta normele OOP și diversele convenții specifice C#, utilizând denumiri de variabile specifice, interfețe, polimorfism și moștenire, printre altele. De asemenea, în realizarea proiectului am ținut cont de principiile Clean Code, pentru a păstra codul ușor de citit și mentenabil.

Din punct de vedere al design pattern-urilor, Observer, Factory și Strategy ne-au fost de ajutor în structurarea codului.



Ierarhia proiectului cu cele 3 componente:

*BombermanTests,
BombermanBase
și BombermanMONO*



Partea de logica a aplicatiei este conținută într-un proiect de tip DLL. Pe scurt, in acesta avem:

Clasele principale IEntity, ITile, ITileMap, IBomberman, care sunt elementele de baza ale aplicatiei. Aceste interfete sunt publice, iar ele sunt concretizate in clase interne.

Entitatile pot fi de doua tipuri: Player sau Enemy. De aici reiese si nevoia aplicarii design pattern-urilor Factory si Strategy. Factory initializeaza Entity-ul in functie de tip, Player sau Enemy, in timp ce strategy selecteaza cum se actioneaza la apelarea metodei Move

Pe langa acestea, se pot remarca diverse principii si implementari specifice OOP, cum ar fi polimorfismul, interfetele sau incapsularea

IBomberman.cs este clasa principala, concretizata prin Bomberman.cs. Aceasta trebuie instantiata si utilizata pentru a crea un joc.

Design Patterns

Design pattern-urile folosite în proiectul nostru sunt:

- Observer
- Factory
- Strategy

Factory - acest design pattern a fost folosit la crearea entităților de Player și Enemy. Deși vizual aceste entități sunt complet diferite, în cod este folosită doar clasa Entity, care este configurată cu ajutorul design pattern-ului Factory în funcție de tipul real al entității.

```
public interface IEntityFactory
{
    7 references
    IEntity CreateEntity(string username, (int, int) position, int? noOfBombs = null, int? noOfLives = null);
}
```

```
public class EnemyFactory : IEntityFactory
{
    4 references
    public IEntity CreateEntity(string username, (int, int) position, int? noOfBombs = null, int? noOfLives = null)
    {
        return new Entity(username, EntityDefaults.NoOfBombs, EntityDefaults.NoOfLives, position, new AIMoveStrategy());
    }
}
```

Factory Design Pattern

```
var ef = new EnemyFactory();
foreach (var spawnPoint in _crtLevel.EnemySpawnPoints)
{
    _enemies.Add(ef.CreateEntity("mrEnemy", spawnPoint));
}
```

```
_player = new PlayerFactory().CreateEntity(username, (0, 0));
```

Inițializarea player și enemy

Strategy — acest design pattern a fost folosit pentru a putea utiliza metoda Move atât pentru a mișca un Player, cât și pentru a realiza mișcarea automată a unui Enemy.

```
3 references
public interface IMoveStrategy
{
    3 references
    (int, int) Move(TileMap tileMap, (int X, int Y) crtPos, int xMove, int yMove);
}
```

Implementarea interfeței IMoveStrategy

```

public class PlayerMoveStrategy : IMoveStrategy
{
    2 references
    public (int, int) Move(TileMap tileMap, (int X, int Y) crtPos, int xMove, int yMove)
    {
        try
        {
            Tile nextTile = tileMap.GetTile((crtPos.X + xMove, crtPos.Y + yMove));
            if (nextTile.IsWalkable())
                return (crtPos.X + xMove, crtPos.Y + yMove);
        }
        catch
        {
            return crtPos;
        }

        return crtPos;
    }
}

```

Implementarea strategiei de mișcare a player-ului

```

public IMoveStrategy MoveStrategy;

public void SetMoveStrategy(IMoveStrategy moveStrategy)
{
    MoveStrategy = moveStrategy;
}

6 references
public void Move(TileMap tileMap, int x = 0, int y = 0)
{
    Position = MoveStrategy.Move(tileMap, Position, x, y);
}

```

Utilizarea strategiei de mișcare

Observer - folosit pentru a notifica proiectul MonoGame asupra diverselor evenimente care apar odată cu derularea jocului. Conform definiției acestui design pattern, definim un Observer si un Observable.

Astfel, avem interfata IBombermanObserver care este implementată de clasa principala din Mono, Bomberman.cs.

```
public interface IBombermanObserver
{
    2 references
    void OnMoveMade(object sender, MoveEventArgs e);
    2 references
    void OnLevelOver(object sender, LevelOverEventArgs e);
    2 references
    void OnPlayerEnemyCollision(object sender);
}
```

Definirea interfeței IBombermanObserver

```
4 references
public class Bomberman : Game, IBombermanObserver
{
```

Moștenirea interfeței

```
2 references
public void OnMoveMade(object sender, MoveEventArgs e)...
2 references
public void OnLevelOver(object sender, LevelOverEventArgs e)...
2 references
public void OnPlayerEnemyCollision(object sender)...
```

Implementarea metodelor interfeței

Clasa Observable este Bomberman.cs, care implementează funcții specifice, precum adăugarea si scoaterea subscriberilor, respectiv notificarea subscriberilor in functie de eveniment

```
#region Observer-related methods
2 references
public void AddObserver(IBombermanObserver observer)...
1 reference
public void RemoveObserver(IBombermanObserver observer)...

2 references
private void NotifyMoveMade(IEntity entity)...
2 references
private void NotifyGameOver(IEntity entity)...
1 reference
private void NotifyPlayerEnemyCollision()...
#endregion
```

Unit Testing

Pentru proiectul de Unit Testing am folosit frameworkul MSTests.

În cadrul unit testelor am verificat corectitudinea codului scris din punct de vedere logic. Scopul principal este asigurarea ca fiecare metoda scrisă își îndeplinește scopul și nu are diverse efecte secundare. Am început prin a face teste simple, de exemplu, testarea că inițializarea entităților se face corect în cadrul testului `TestEntitiesInitialization()`.

În continuare am verificat ca funcționalitățile jucătorului să funcționeze corect în limita hărții definită la începutul jocului prin teste precum `TestPlayerMoveOutOfMap()`, `TestPlacingBomb()` și `LosingLife()`.

```
[TestMethod]
0 references
public void LosingLife()
{
    var game = BombermanFactory.CreateGame("Denis");

    int exNoOfLives = 2;

    while (game.Player.NoOfLives > 0)
    {
        game.Player.RemoveLife();
        Assert.AreEqual(exNoOfLives, game.Player.NoOfLives);
        exNoOfLives--;
    }
    game.Player.RemoveLife();
    Assert.AreEqual(0, game.Player.NoOfLives);
}
```

În următorii pași se testează funcționalitățile hărții și modul în care jucătorul interacționează cu ea. Testele cuprind și funcționalități mai complexe ale proiectului și modul de relaționare al proiectelor între ele, unde am folosit mai mult decât teste simple.

Diagramme

(Notă: pentru a vizualiza diagramele se poate da zoom – nu se pierde calitatea la zoom-in)

Class

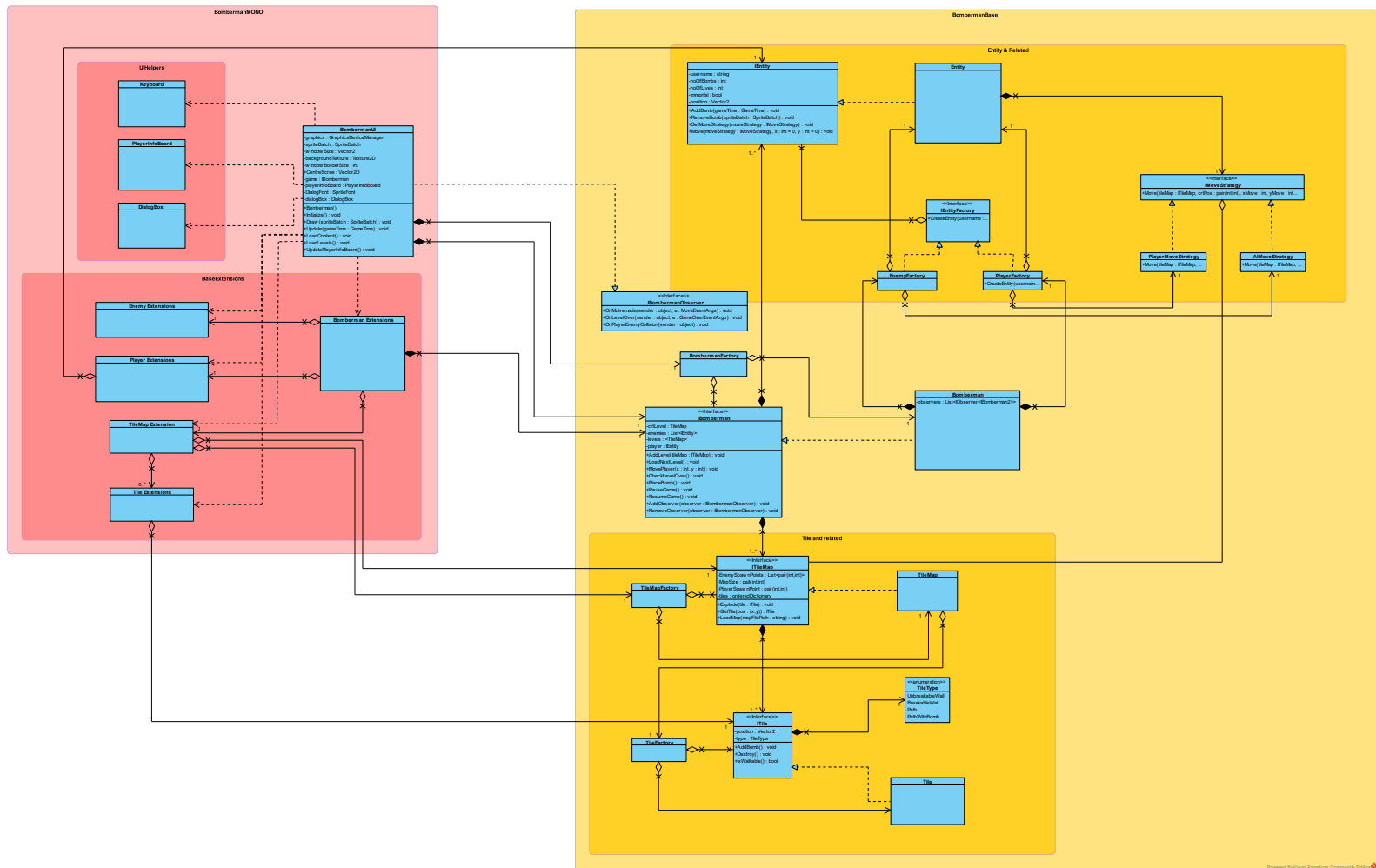


Diagrama arată relațiile dintre clasele proiectului de baza (galben), cele dintre clasele proiectului de UI (rosu-roz), cât și legăturile dintre cele doua proiecte.

Use Case

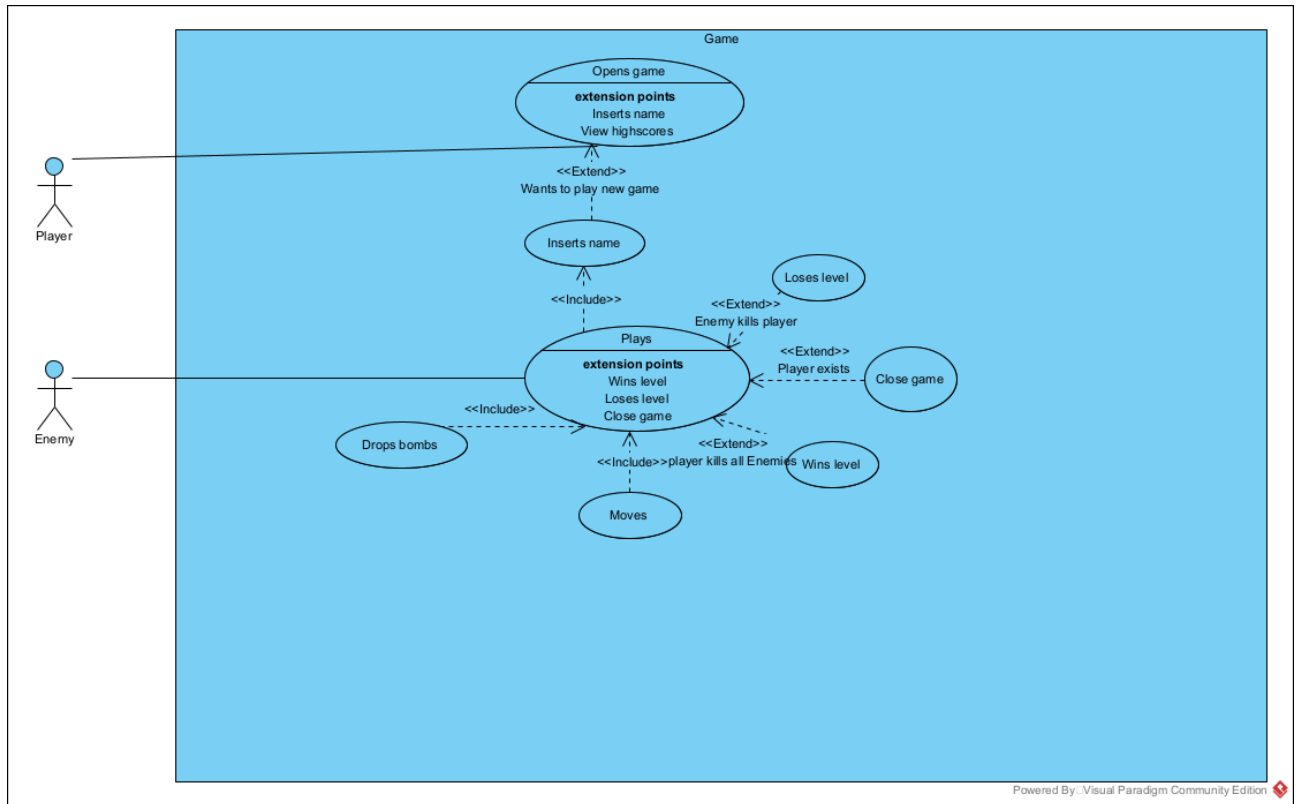


Diagrama prezintă cum jucătorul interacționează prima dată cu meniul programului și apoi cu partea de joc. Adversarul este un actor virtual de aceea el poate interacționa doar cu partea de joc.

Activity

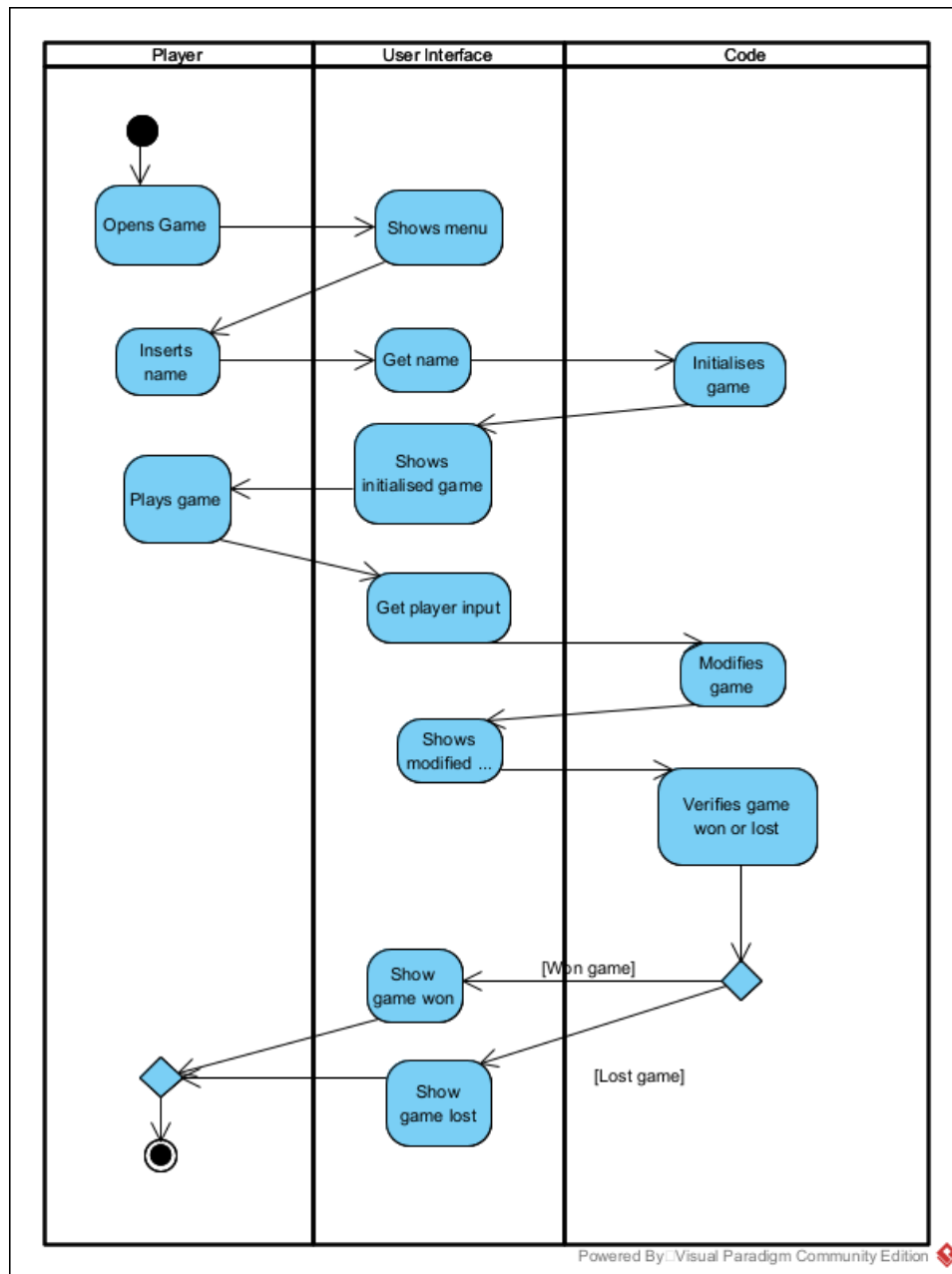


Diagrama de activități prezintă la scară largă toate stările prin care trece jucatorul, interfața grafică și jocul. Prima dată jucatorul deschide jocul. Interfața afișează meniul de start unde jucatorul introduce numele. Cu acea informație se inițializează jocul. Jocul trece prin schimburi de informație cu jucatorul iar la final există 2 stări posibile, cea de câștig a jocului și de pierdere a jocului.

State Diagram

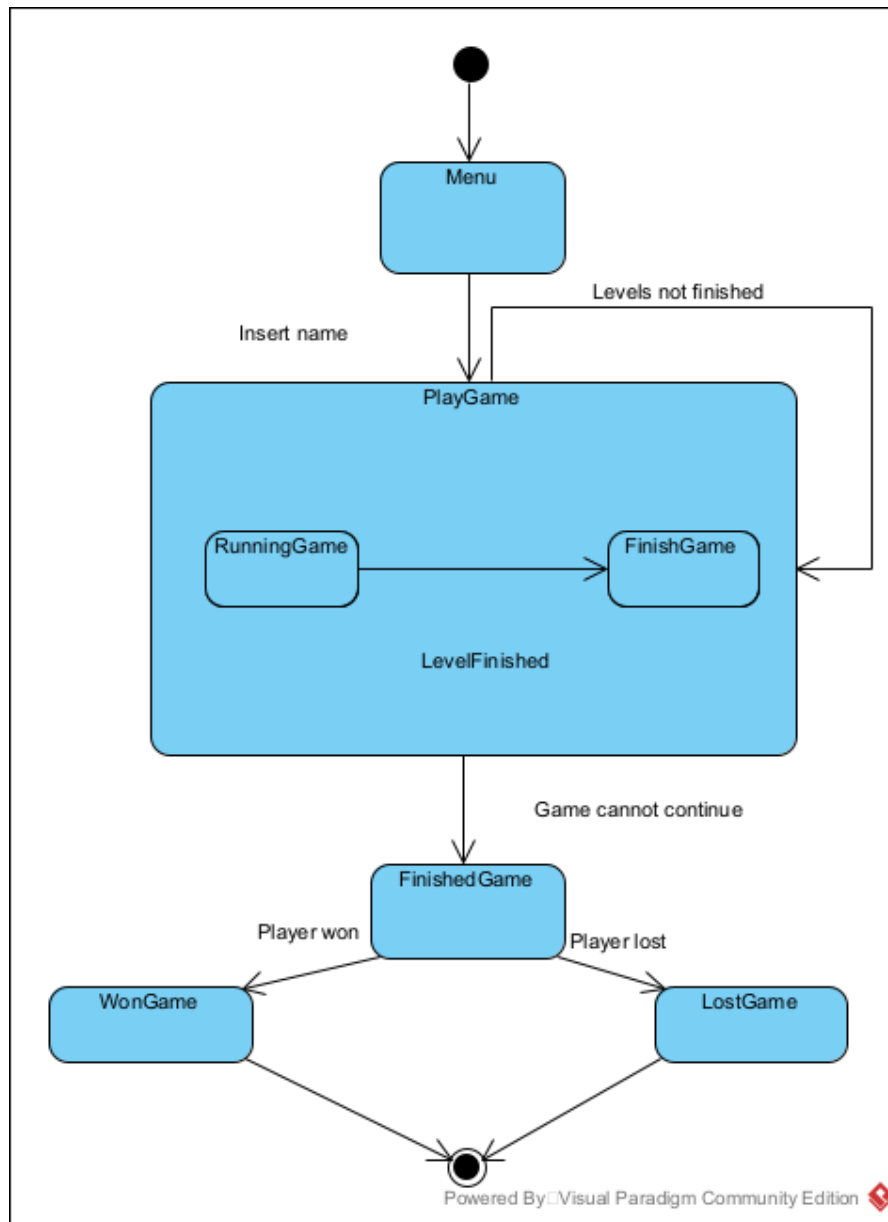
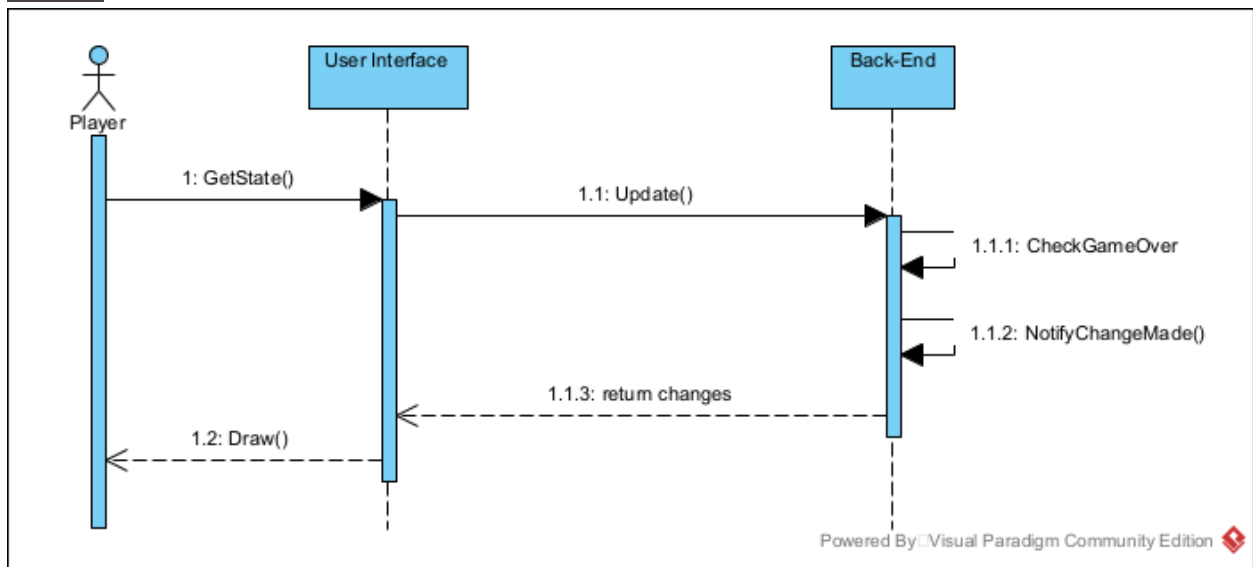


Diagrama de stări prezintă stările jocului: în meniu, desfășurarea jocului, joc terminat și joc câștigat sau pierdut.

Sequence



Aici evidențiem cum din punct de vedere temporal se desfășoară acțiunea în program. Jucătorul trimite un input interfeței. Această îl transmite mai departe către partea de logică. Aici se fac modificările necesare iar apoi ele sunt trimise către interfață pentru a fi afișate utilizatorului.

Communication

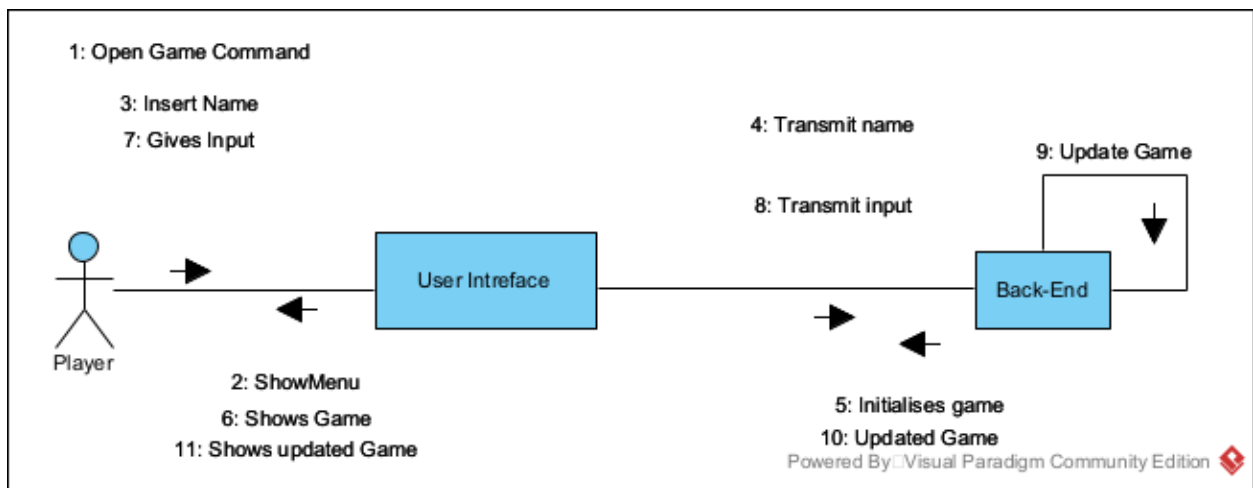


Diagrama de comunicare arată schimbul de informație de la jucător la cele 2 componente ale programului.

Concluzii

În concluzie, putem spune ca în realizarea acestui proiect am acumulat cunoștințe noi în domeniul ingineriei software și, în general, al informaticii. Am învățat cum să structuăm un proiect care să utilizeze design patterns și am pus în valoare diagramele de modelare. De asemenea, am învățat diverse practici folositoare în dezvoltarea software, precum separarea logică-UI, lucrul cu interfețe sau utilizare texturilor.