



**Universitatea Tehnică "Gheorghe Asachi" din Iași**  
**Facultatea de Automatică și Calculatoare**  
**Domeniul Calculatoare și Tehnologia Informației**  
**Specializarea Tehnologia Informației**

## **Proiect Evaluarea Performanțelor**

Studenti:

Batalan Vlad, Grupa 1410A

Petrișor Iosif-Marcelin, Grupa 1410A

An universitar 2021-2022

# Cuprins

<b>Capitolul 1. Tema proiectului</b>	<b>3</b>
1.1 Enunțarea temei proiectului	3
1.2 Motivul alegerii temei	3
<b>Capitolul 2. Arhitectura generală</b>	
1.1 Arhitectura orientată eveniment	3
1.2 Arhitectura orientată obiect	4
<b>Capitolul 3. Funcționalitatea</b>	<b>4</b>
1.1 Descriere interacțiune	4
1.2 Controale	5
<b>Capitolul 4. Rolul fiecărui membru al echipei</b>	<b>6</b>
1.1 Implementare	6
1.2 Documentație	6
<b>Capitolul 5. Complexitatea în cazul cel mai favorabil, cel mai defavorabil, în cazul mediu</b>	
<b>.Demonstrația corectitudinii pentru funcția</b> <code>public static int[,] LeeAlgorithm(int[,] mapMatrix, Point tankIndexes, Point enemyIndexes)</code>	<b>6</b>
5.1 Scopul funcției	6
5.2 Complexitatea în cazul cel mai favorabil	6
5.3 Complexitatea în cazul cel mai defavorabil	6
5.4 Complexitatea în cazul mediu	6
5.5 Codul modulului analizat	7
5.6 Demonstrația corectitudinii secvenței	8
<b>Capitolul 6. Teste automate pentru a demonstra buna funcționare a programului.</b>	<b>11</b>
6.1 Modulul de Testare unitară	11
6.2 Rezultate	12
<b>Capitolul 7. Explicații suplimentare</b>	<b>12</b>
7.1 Utilizarea algoritmului lui Lee	12
7.2 Șabloane de proiectare	13
<b>Capitolul 8. Bibliografie</b>	<b>15</b>

# **Capitolul 1. Tema proiectului**

## **1.1 Enunțarea temei proiectului**

Tema proiectului constă în elaborarea unei aplicații de tip desktop care reprezintă un joc cu tancuri.

## **1.2 Motivul alegerii temei**

Unul dintre principalele motive ale alegerii acestei teme este faptul că problematica realizării unui joc de asemenea amploare atinge concepte din domenii variate, majoritatea fiind învățate în cadrul subiectelor abordate la facultate. Printre acestea se numără: Programarea Orientată Obiect, Sistemele de Prelucrare Grafică, Ingineria Programării, multiple domenii din matematică și multe altele.

Pentru a menține acțiunea, fereastra trebuie redesenată în fiecare cadru al jocului, lăsând un timp relativ scurt de procesare a tuturor evenimentelor din acesta. Astfel, performanța algoritmilor utilizați trebuie să respecte anumite cerințe care impun restricții de timp și spațiu. Din acest punct de vedere, aplicația se potrivește cu subiectul abordat la materia Evaluarea Performanțelor.

# **Capitolul 2. Arhitectura generală**

## **1.1 Arhitectura orientată eveniment**

Această paradigmă de programare promovează producția, detecția și consumul de evenimente. Datorită faptului că interacțiunea cu utilizatorul reprezintă o componentă foarte importantă în cadrul aplicației, acțiunile acestuia (apăsare de taste, de butoane, mișcarea mouse-ului, etc.) sunt încapsulate în evenimente și tratate fiecare în mod particular.

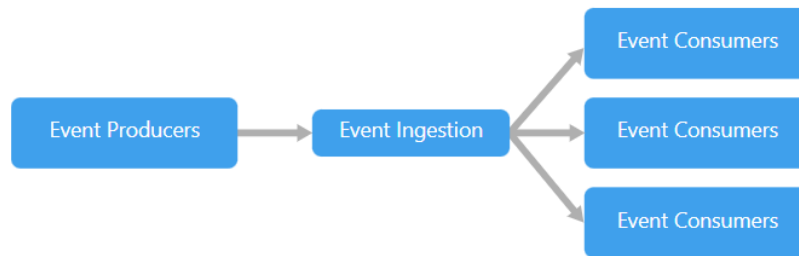


Fig 1. Paradigma orientată eveniment

Sursă imagine: <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven>

Evenimentele sunt generate de către cei doi utilizatori ai aplicației și sunt transferate către interfața aplicației, urmând a fi procesate de către unitatea de procesare.

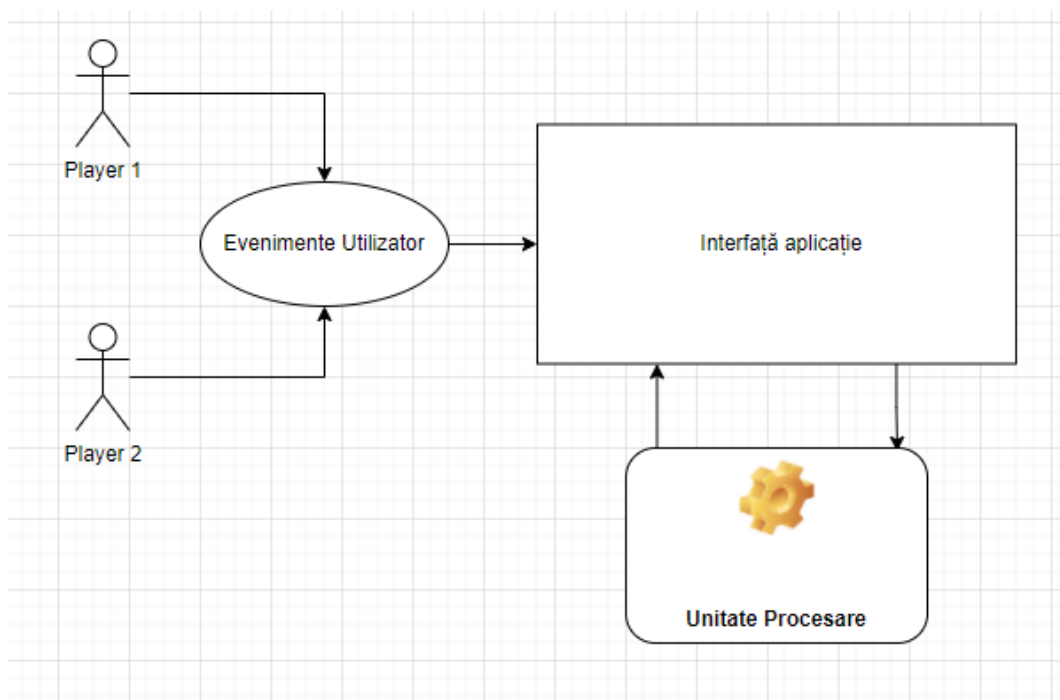


Fig. 2. Diagrama generală interacțiune useri - aplicație

## 1.2 Arhitectura orientată obiect

Arhitectura orientată obiect reprezintă un model programatic de organizare a componentelor software în obiecte pentru a fi manipulate cu ușurință, dar și pentru a facilita reutilizarea codului.

# Capitolul 3. Funcționalitatea

## 1.1 Descriere interacțiune

În această aplicație utilizatorul poate alege dacă vrea să joace împotriva computerului sau împotriva altui jucător. Deoarece la executarea aplicației jocul începe direct împotriva computerului, pentru a începe alt joc se apasă butonul **Restart Level**.

Va apărea o fereastră în care jucătorul este întrebat dacă dorește să joace tot împotriva computerului sau a unui al doilea jucător.

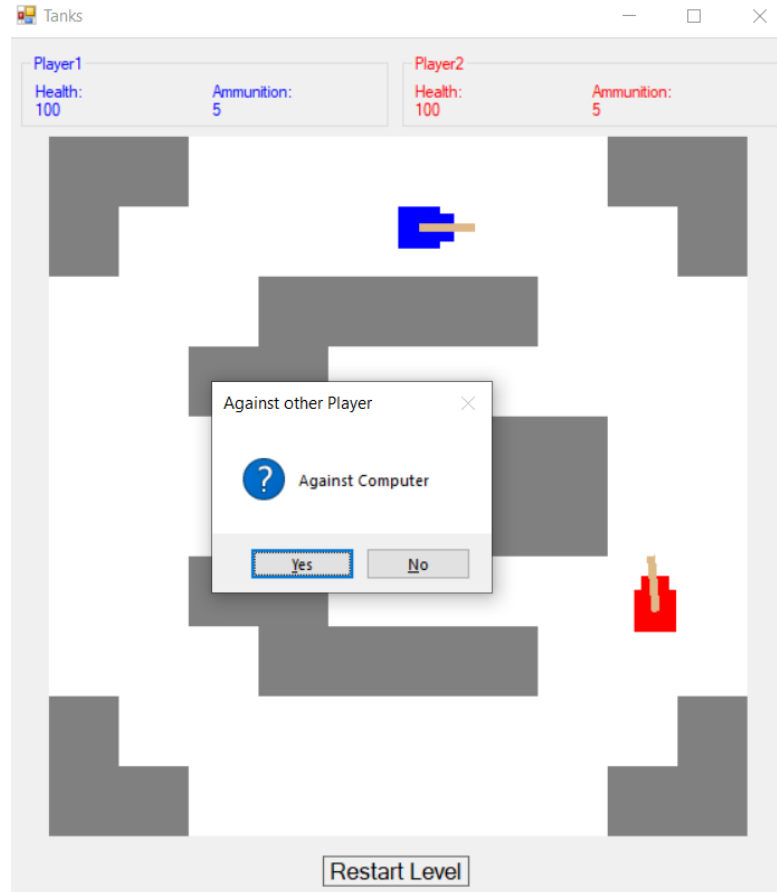


Fig. 3. Pop-up la apăsarea butonului de restart

## 1.2 Controale

Fiecare jucător controlează câte un tanc, scopul lui este să distrugă tancul inamic. Ambele tancuri au la început Health 100 și 5 proiectile. Muniția se reîncarcă odată la 1.5 secunde.

Controalele pentru Player 1:

Mișcare tanc: Săgeți up, down, left, right

Mișcare tun: Spre stânga - Z  
Spre dreapta - C

Trage: X

Controalele pentru Player 2:

Mișcare tanc: Tastele WASD  
Mișcare tun: Spre stânga - I  
Spre dreapta - P  
Trage: O

## Capitolul 4. Rolul fiecărui membru al echipei

### 1.1 Implementare

Clasele Map și MovingFlags au fost implementate de Petrișor Iosif-Marcelin  
Restul programului a fost implementat de Batalan Vlad.

### 1.2 Documentație

Capitolele 2, 5.6- Invariantul din bucla 1 (inv1) Partea formală pentru bucla 1 și  
Demonstrația pentru inv1,6 și 7.2 au fost scrise de Batalan Vlad  
Restul documentației a fost realizată de Petrișor Iosif-Marcelin

## Capitolul 5. Complexitatea în cazul cel mai favorabil, cel mai defavorabil, în cazul mediu. Demonstrația corectitudinii pentru

**funcția** `public static int[,] LeeAlgorithm(int[,] mapMatrix, Point tankIndexes, Point enemyIndexes)`

### 5.1 Scopul funcției

Această funcție recalculează la fiecare frame poziția inamicului și caută drumul cel mai scurt spre acesta și returnează poziția următorului tile în care acesta trebuie să se deplaseze.

Pentru calculul complexităților nu vom lua în considerare partea de ajustare a marginii matricei (construirea matricei *border*).

**Complexitatea spațiu** pentru funcția analizată este reprezentată de coada `toBeVisited`. Memoria maximă necesară este de ordin  $O(m+n)$ .

### 5.2 Complexitatea în cazul cel mai favorabil

Cazul cel mai favorabil este întâlnit atunci când inamicul se află în același tile cu tancul controlat de AI. În acesta caz, tancul computerului nu trebuie să se mai deplaseze. Astfel funcția va returna punctul (0, 0).

Complexitatea timp a funcției:  $O(1)$

### 5.3 Complexitatea în cazul cel mai defavorabil

Cazul cel mai defavorabil este întâlnit atunci când inamicul se află la distanța maximă față de tancul controlat de AI. Aici, complexitatea depinde de dimensiunea hărții.

Complexitatea timp:  $\Omega(\text{height} * \text{width})$ , unde height și width reprezintă dimensiunile hărții.

## 5.4 Complexitatea în cazul mediu

Cazul mediu este întâlnit atunci când inamicul se află la o distanță arbitrară față de tancul controlat de AI.

Probabilitatea ca inamicul să se afle într-un tile, pentru o mapă dată, este  $1/(\text{height} * \text{width} - n)$ , unde n reprezintă numărul de tile-uri ce construiesc obstacole.

Complexitatea timp:  $\Theta(\text{height} * \text{width})$

## 5.5 Codul secvenței analizate

### 5.5.1 Funcția de analizat

```
public static int[,] LeeAlgoritm(int[,] mapMatrix, Point tankIndexes,
Point enemyIndexes)
```

```
int[,] border = new int[height + 2, width + 2];

// Adjust the borders of the matrix
for (int i=1; i <= height; i++)
    for (int j=1; j <= width; j++)
        border[i, j] = mapMatrix[i - 1, j - 1] == 1 ? -1 : 0;

for(int i=0; i <= height + 1; i++)
    border[i, 0] = border[i, width + 1] = -1;
for(int j=0; j<=width + 1; j++)
    border[0, j] = border[height + 1, j] = -1;

// Get the queue
Queue<Point> toBeVisited = new Queue<Point>();
//P0
// Push the starting point
toBeVisited.Enqueue(tankIndexes); // A0
//P1
// Set the initial distance as 1
border[tankIndexes.X, tankIndexes.Y] = 1; // A1
//P2
// Bucla 1
while(toBeVisited.Count != 0) // C1
{
//P2&c1
// Get the first in the queue
Point currentPoint = toBeVisited.Dequeue(); // A2
//P3
// If it is the enemy, we found the path, we can stop
if (currentPoint.Equals(enemyIndexes)) // C2
//P3&c2
    Break; // Exit
//P3&!c2 ⇔ P4
// Bucla 2
```

```

        // Foreach neighbour, check if it can be visited
//P4
        foreach(Point delta in directions) // C3
        {
//P4&c3
            // Get the neighbour by incrementing the position with delta
            Point neighbour = new Point(currentPoint.X + delta.X,
currentPoint.Y + delta.Y); //A3
//P5

            // If it can be visited
            // - it has border = 0 (it was'n visited before and it
is not margin)
            // - there is no obstacle on the map
            if(border[neighbour.X, neighbour.Y] == 0 &&
mapMatrix[neighbour.X - 1, neighbour.Y -1] != 1) //C4
            {
//P5&c4
                // Update the border
                border[neighbour.X, neighbour.Y] =
border[currentPoint.X, currentPoint.Y] + 1; //A4

                // Add the neighbour to the queue
                toBeVisited.Enqueue(neighbour); // A4

//P6
            }
//P5&!c4
        }
//P4&!C3 ⇔ P7
    }

```

## 5.6 Demonstrația corectitudinii funcției

Precondiții:

- enemyIndexes și tankIndexes sunt puncte valide pe hartă;
- există cel puțin un element în coada toBeVisited;
- border este inițializat corect;

Postcondiții :

- Matricea border este completată parțial cu valori din mulțimea  $\{-1, 0, 1, \dots, \text{height} * \text{width}\}$ , unde -1 reprezintă obstacol, 0 spațiu nevizitat, iar restul valorilor reprezintă distanța Manhattan dintre poziția  $i, j$  și poziția tancului la care se adaugă o unitate.
  - Prin completare parțială se înțelege până la evaluarea elementului cu coordonatele tancului inamic.

Invariantul din bucla 1:

- Denumire: inv1
- Semnificație: Pozițiile evaluate până acum din matricea border au ca valoare distanța Manhattan până la poziția tancului desemnată de punctul tankIndexes la care se mai adaugă o valoare.

Invariantul din bucla 2:



- Notăție: inv2
- Semnificație: Presupunând că punctul curent are indecșii i și j, vecinii valizi (nu sunt margine, obstacol și au fost vizitați) ai acestuia au valoarea  $border[i, j] + 1$  și sunt introduși în coada toBeVisited.

### Demonstrație bucla 1 bine definită:

Din P1, coada toBeVisited conține cel puțin un element. Condiția C1:  
 $toBeVisited.Count \neq 0$  este satisfăcută, deci bucla se execută cel puțin odată.

### Demonstrație bucla 2 bine definită:

Vectorul directions conține 4 valori codificate ca: up, down, left și right.  
 Orice punct are cei 4 vecini datorită bordării, rezultă că bucla este bine definită.

### Demonstrație inv1:

Verificăm prin inducție matematică faptul că la finalul buclei, invariantul este respectat.

#### Pasul 1:

Inițial, border conține la poziția 1 la poziția corespondentă tancului, iar nicio altă valoare nu a mai fost vizitată până acum. Această valoare reprezintă distanța Manhattan + 1 până la poziția tancului. Adevărat

#### Pasul k:

Presupunem că:

$\forall p \in M_{\text{puncte verificate}} = \{1, 2, \dots, t\}, border[i_p, j_p] = dist_{\text{Manhattan}}(tankIndexes, [i_p, j_p]) + 1$   
 Adevărat

#### Pasul k+1:

Având punctul  $[i_s, j_s] = \text{currentPoint}$ , vârful cozii de elemente de vizitat, toBeVisited, pentru care se cunoaște că  $border[\text{currentPoint}] = dist_{\text{Manhattan}}(tankIndexes, [i_s, j_s]) + 1$ , valorile corespunzătoare punctelor vecine valide care nu au fost vizitate (au  $border[i, j] = 0$ ) vor fi asigurate cu valoarea  $border[i_s, j_s] + 1 = dist_{\text{Manhattan}}(tankIndexes, [i_s, j_s]) + 2$ .

Deoarece se schimbă doar valorile vecinilor nevizitați, distanța Manhattan a acestora față de poziția tancului va fi cu 1 mai mare față de distanța punctului verificat curent.

Deci  $border[i_{\text{vecin}}, j_{\text{vecin}}] = dist_{\text{Manhattan}}(tankIndexes, [i_{\text{vecin}}, j_{\text{vecin}}]) + 1$ , ceea ce confirmă afirmația invariantului.

**Concluzia:** în toate situațiile, matricea border își păstrează proprietatea că pozițiile care au fost evaluate păstrează ca valoare distanța Manhattan +1 față de poziția tancului (tankIndexes).

**Demonstrație inv2:**

Prin inducție demonstrăm ca orice vecin nevizitat, care nu este obstacol și nici margine are, în matricea `borders`, valoarea punctului curent + 1.

$P(\text{tankIndexes}): \text{border}[\text{tankIndexes.X}, \text{tankIndexes.Y}] = 1$  **Adevărat**

Presupunem

$P(\text{currentPoint}): \text{border}[\text{currentPoint.X}, \text{currentPoint.Y}] = x$  **Adevărat**

Demonstrăm ca  $P(\text{neighbour}): \text{border}[\text{neighbour.X}, \text{neighbour.Y}] = x+1$ , unde `neighbour` ia valoarea tuturor vecinilor lui `currentPoint`.

Având în vedere instrucțiunea  $\text{border}[\text{neighbour.X}, \text{neighbour.Y}] = \text{border}[\text{currentPoint.X}, \text{currentPoint.Y}] + 1$ ; rezultă că  $P(\text{neighbour})$  este adevărat.

**Partea formală înainte de bucle:**

$P0 \rightarrow A0 \rightarrow P1$

Coadă `toBeVisited` conține elementul `tankIndexes`.

Din precondiție, `borders` este bine inițializat.

$P1 \rightarrow A1 \rightarrow P2$

În matricea `borders`, este marcată cu 1 elementul corespunzător poziției tancului.

**Partea formală pentru bucla 1:**

$P2 \& c1 \rightarrow A2 \rightarrow P3$

Punctul următor de verificat `currentPoint` este scos din coadă.

$P3 \& c2 \rightarrow \text{Exit} \rightarrow Q$

Căutarea se termină odată cu găsirea punctului unde se afla tancul inamic. În acest moment, execuția buclei se termină.

**Partea formală pentru bucla 2:**

$P4 \& c3 \rightarrow A3 \rightarrow P5$

Pentru un element scos din coada nevidă se creează vecinii după direcțiile din vectorul `directions`.

$P5 \& c4 \rightarrow A4 \rightarrow P6$

Dacă acest vecin nu este obstacol, nu este margine și nici nu a fost vizitat atunci el este adăugat în coadă și vizitat cu valoarea corespunzătoare  $(\text{border}[\text{currentPoint.X}, \text{currentPoint.Y}] + 1)$

$P4 \& !c3 \rightarrow P7$

Când am prelucrat toți vecinii ajung în starea finală pentru acest punctul curent.  $P7$  stare finală pentru bucla 2.

**Concluzie:** Toți vecinii care nu sunt obstacole, margine sau nevizitați ai unui punct curent `currentPoint` cu valoarea `border[currentPoint.X, currentPoint.Y]=x` au valoarea `border[neighbour.X, neighbour.Y] = x+1`.

### Funcția de terminare pentru bucla 1:

Deoarece numărul de elemente din coadă nu crește sau scade monoton, el depinzând de numărul de vecini adăugați (ex:  $1 \rightarrow 0 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow \dots$ ), funcția de terminare pentru bucla 1 este

$$F(enemy) = \begin{cases} 0, & \text{dacă coada este vidă} \\ border[enemy.X, enemy.Y] - border[currentPoint.X, currentPoint.Y] & \text{în rest} \end{cases}$$

### Funcția de terminare pentru bucla 2:

Deoarece bucla 2 se termină atunci când am procesat toți vecinii unui punct curent și fiecare punct are 4 valori în vectorul `directions`.

$$F(n) = 3 - n, \forall n = \overline{0, 3}$$

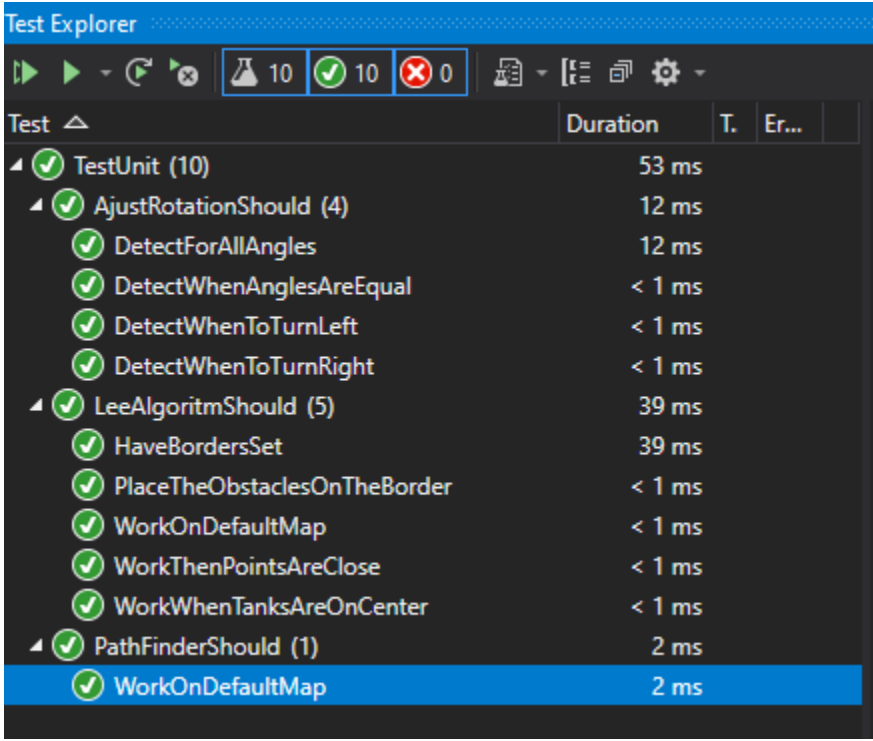
## Capitolul 6. Teste automate pentru a demonstra buna funcționare a programului.

### 6.1 Modulul de Testare unitară

Modulul de testare unitară conține o serie de clase de testare, fiecare responsabilă pentru o funcționalitate anume.

- 1) AjustRotationShould: Este utilizat pentru detectia direcției de adaptare a unui unghi referință pentru a atinge în cel mai eficient mod un unghi țintă. Deoarece sistemul de coordonate este amplasat în stânga-sus, unghiurile reprezentate sunt în oglindă. Din acest motiv, pentru teste, desi target-ul și reference-ul par a fi sugera o direcție, în sistemul de coordonate, aceasta trebuie invertita.
- 2) LeeAlgorithmShould: Este utilizat pentru testarea generării unei matrice în care toate elementele de la punctul de start până la punctul final au ca valoare distanța Manhattan până la punctul de start. Acesta este optimizat și odată ce va găsi inamicul, nu va mai verifica restul punctelor.
- 3) PathFinderShould: Este utilizat pentru testarea funcției care primește o matrice generată de algoritmul lui Lee și identifică direcția în care trebuie să se deplaseze punctul de start pentru a ajunge la cel de final.

## 6.2 Rezultate



Test	Duration	T.	Er...
TestUnit (10)	53 ms		
AjustRotationShould (4)	12 ms		
DetectForAllAngles	12 ms		
DetectWhenAnglesAreEqual	< 1 ms		
DetectWhenToTurnLeft	< 1 ms		
DetectWhenToTurnRight	< 1 ms		
LeeAlgorithmShould (5)	39 ms		
HaveBordersSet	39 ms		
PlaceTheObstaclesOnTheBorder	< 1 ms		
WorkOnDefaultMap	< 1 ms		
WorkThenPointsAreClose	< 1 ms		
WorkWhenTanksAreOnCenter	< 1 ms		
PathFinderShould (1)	2 ms		
WorkOnDefaultMap	2 ms		

## Capitolul 7. Explicații suplimentare

### 7.1 Utilizarea algoritmului lui Lee

Algoritmul lui Lee este foarte cunoscut și este asemănător cu parcurgerea în lățime (Breadth First Search), doar că se aplică pe o matrice și nu pe un graf.

Algoritmul lui Lee presupune doi pași importanți:

1. Primul și poate cel mai important pas este folosirea unei cozi, sub forma unui vector de structuri (de preferabil), care va menține toți pașii pe care o să-i facem de acum în colo. În această coadă se pun, pentru fiecare pas, locurile care s-au marcat la punctul anterior.
2. Se marchează cu numere consecutive toate locurile posibile prin care putem trece, parcurgând în ordine elementele cozii, până când nu mai putem marca, sau am ajuns la final.

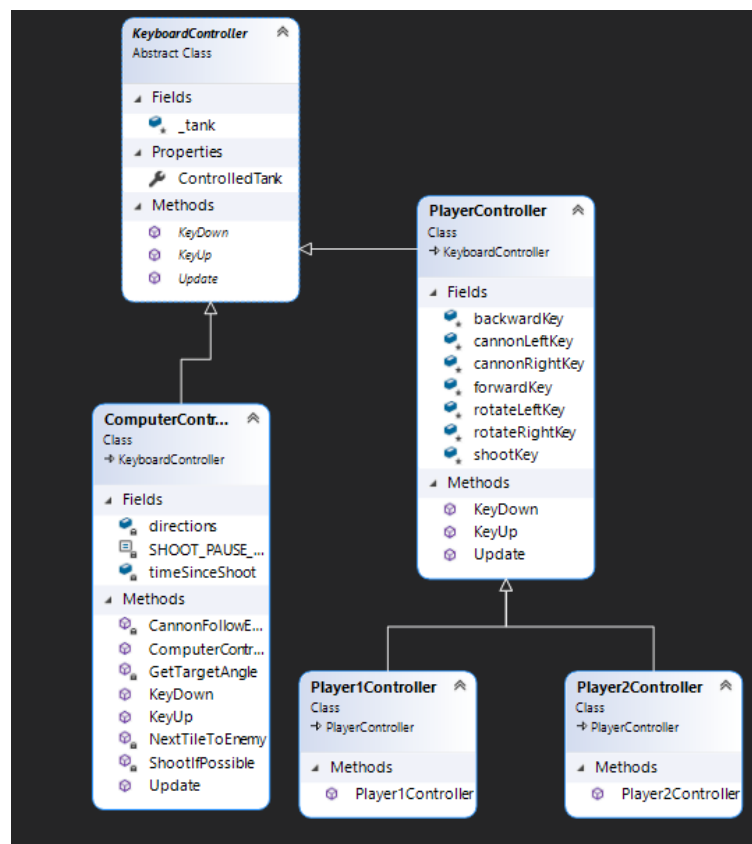
În cazul acestei funcții, coada conține tile-uri și algoritmul reprezintă o variantă îmbunătățită față de cea a lui Lee prin faptul că adaugă o condiție în plus de ieșire: dacă se

ajunge în poziția finală, având în vedere că dorim construcția drumului minim, se va forța ieșirea din buclă.

Optimizarea algoritmului duce la micșorarea complexității de timp, dar, de asemenea, face ca singura cale de ieșire din program, cu precondiția că există cale între cele două tancuri, este descoperirea poziției inamicului, făcând condiția din while să fie adevărată tot timpul.

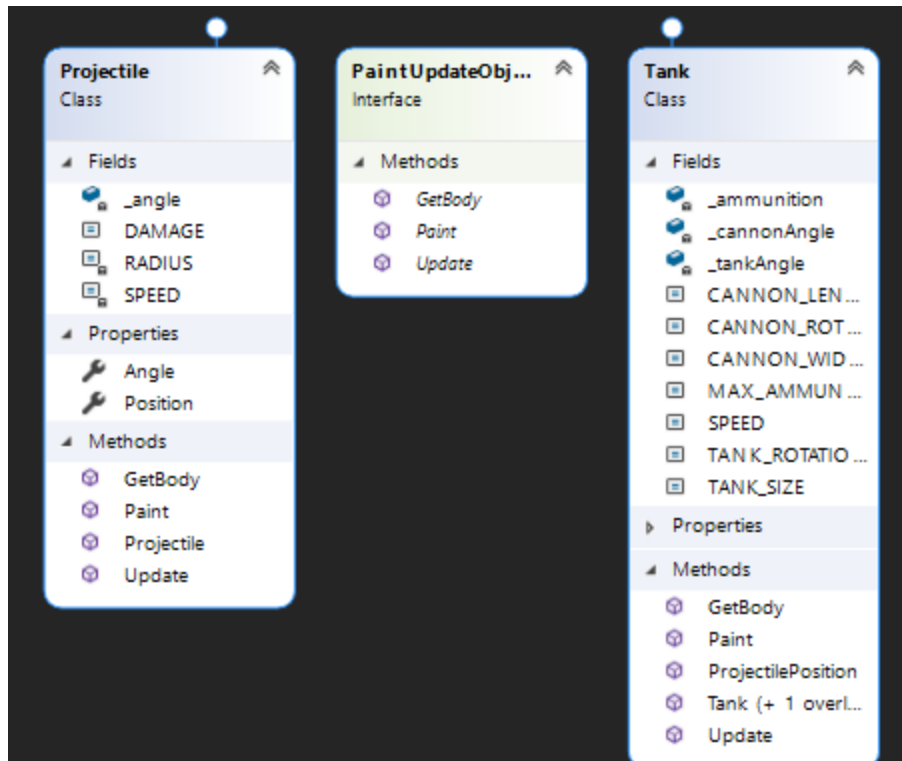
## 7.2 Șabloane de proiectare

Pentru a facilita reutilizarea codului, a fost utilizat polimorfismul pentru utilizarea generală obiectelor de tip Controller pentru tancurile existente în joc. Acest tip de obiect are rolul de a schimba flag-urile interne ale unui tanc, determinându-l să se miște. PlayerController utilizează ca evenimente apăsarea anumitor taste, iar ComputerController analizează pozițiile relative ale tancurilor și decide ce mișcări să facă.



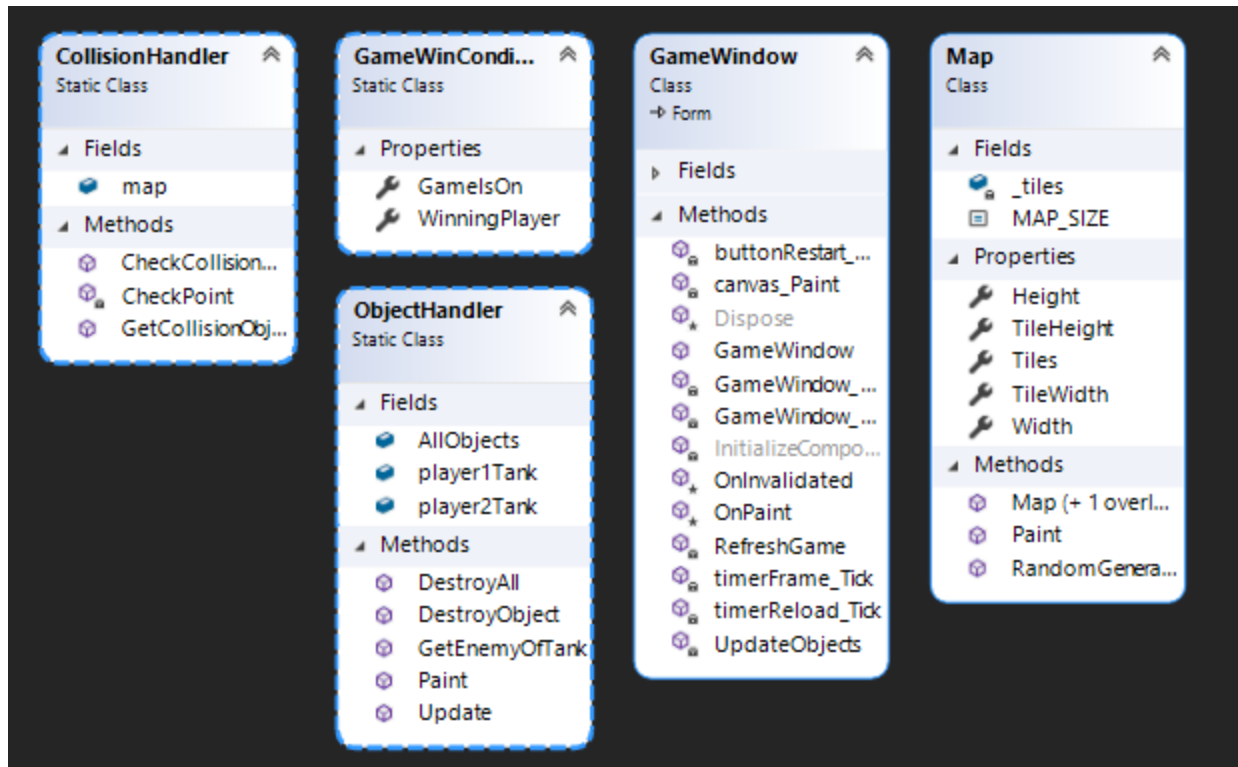
Jocul utilizează două entități: tancurile și proiectilele tancurilor.

Ambele sunt abstractizate în clasele `Tank`, respectiv `Projectile`. Aceste clase implementează interfața `PaintUpdateObject` pentru a facilita polimorfismul.



Pentru gestionarea jocului, a fost necesară utilizarea unor obiecte statice, vizibile de orice clasă, care conțin informații necesare pentru realizarea diferitelor operații.

- CollisionHandler are ca responsabilitate verificarea tuturor coliziunilor din joc, atât cu obstacolele de pe hartă, cât și între obiecte.
- GameWinConditionsHandler conține o serie de steaguri care descriu dacă jocul este activ sau nu, dar și jucătorul câștigător.
- ObjectHandler conține toate obiectele de tip PaintUpdateObject și la fiecare frame, se asigură ca acestea să fie updatate și desenate.
- Map nu este o clasă statică și poate fi instanțiată cu date citite dintr-un fișier.
- GameWindow este fereastra principală a jocului și este responsabilă de tratarea evenimentelor primite de la user.



## Capitolul 8. Bibliografie

- <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven>
- <https://searcharchitecture.techtarget.com/definition/object-oriented-programming-OOP>
- [https://en.wikipedia.org/wiki/Software\\_architecture#Software\\_architecture\\_description](https://en.wikipedia.org/wiki/Software_architecture#Software_architecture_description)
- <https://docs.microsoft.com/en-us/dotnet/desktop/winforms/?view=netdesktop-6.0>
- <https://infoarena.ro/algorithmul-lee>
- D. Zaharie, Algorithms and Data Structures - note de curs.
- D. Lucanu, M. Craus; Proiectarea algoritmilor, Ed. Polirom, 2008.
- S. Skiena – The Algorithm Design Manual, Springer, second edition, 2008.
- D. Zaharie, Introducere în proiectarea și analiza algoritmilor, Edit. Eubeea, Timisoara 2008