Analyse

Licence RGI - Groupe ERP:

Alexis TATARKOVIC

Florent LELIEVRE

Analyse

FP1 Initialisation	
Valeur Ajoutée Choix de la carte (tableau 2D)	
INPUT MAP	**OUTPUT** …..
Logigramme Fonctionnel  	
Service Fonctionnel `// FP1 Initialisation`	
Commentaire ----------	

Analyse

FP2 findShortestPath	15/12/2022

Valeur Ajoutée

Initialise les tableaux distances, visiter et previous avec des valeurs par défaut.

INPUT	**OUTPUT**
----------	----------

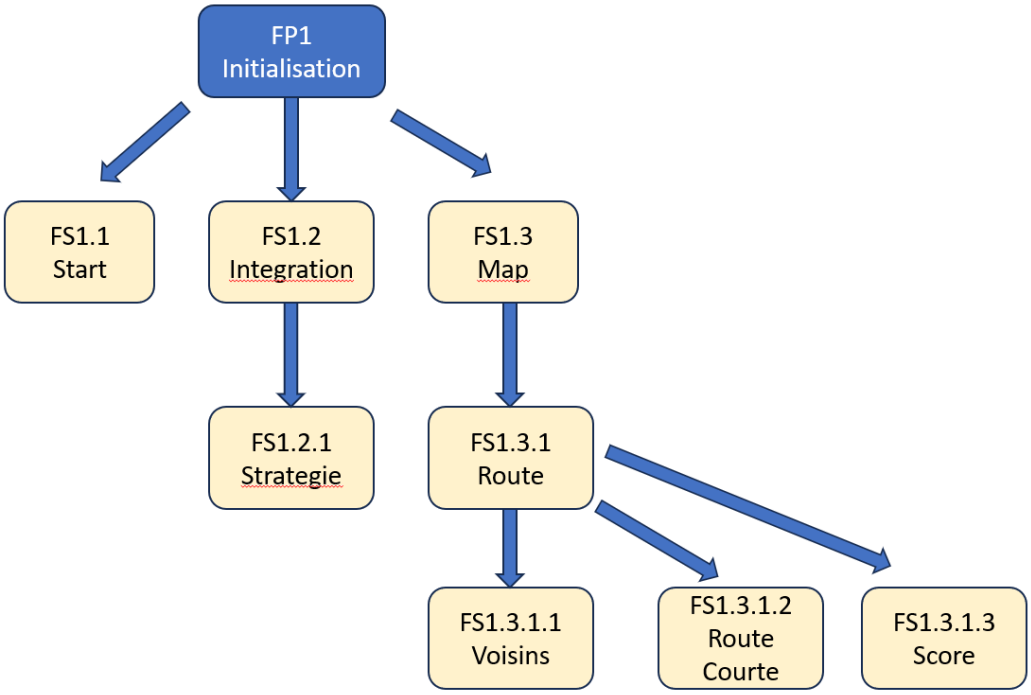
Logigramme Fonctionnel

Service Fonctionnel

```
// FP2
}
```

Commentaire

Analyse Arbre Hiérarchique



Analyse Arbre

- package fr.alexis
  - import java.io.*
  - import java.util.*
  - public class Main
    - static Integer[][] map = new Integer[20][20]
    - static ArrayList<int[]> strategiques = new ArrayList<>()
    - static HashMap<int[], Integer> interets = new HashMap<>()
    - static List<Route> dataset = new ArrayList<>()
    - static int[] start = {18, 10}
    - public static void main(String[] args)
      - getMap()
      - getSpeciaux()
      - for (int[] pointStrategique : strategiques)
        - for (Map.Entry<int[], Integer> entry : interets.entrySet())
          - List<int[]> shortestPath = findShortestPath(pointStrategique, pointInteret)
          - int cost = calculateCost(shortestPath) - entry.getValue()
          - dataset.add(new Route(shortestPath, cost))
      - System.out.println("Pour visualiser : le départ est en [18, 10] (forme [y, x]) et donc on commence en 0 jusqu'à 19 au lieu de 1 jusqu'à 20")
      - System.out.println("Dataset : Nombre de routes : " + dataset.size())
      - List<int[]> highestScoringPath = findHighestScoringPath()
      - System.out.println("Chemin le plus rentable : " + Arrays.deepToString(highestScoringPath.toArray()))
      - System.out.println("Score : " + calculateScore(highestScoringPath))
    - public static List<int[]> findShortestPath(int[] start, int[] end)
      - int[][] distances = new int[map.length][map[0].length]
      - boolean[][] visited = new boolean[map.length][map[0].length]
      - int[][][] previous = new int[map.length][map[0].length][2]
      - for (int i = 0; i < distances.length; i++)
        - Arrays.fill(distances[i], Integer.MAX_VALUE)
        - for (int j = 0; j < distances[i].length; j++)
          - previous[i][j] = new int[]{-1, -1}
      - distances[start[0]][start[1]] = 0

- PriorityQueue<int[]> queue = new PriorityQueue<>(Comparator.comparingInt(o -> distances[o[0]][o[1]]))
- queue.offer(start)
- while (!queue.isEmpty())
  - int[] current = queue.poll()
  - if (Arrays.equals(current, end))
    - List<int[]> path = new ArrayList<>()
    - int[] temp = end
    - while (!Arrays.equals(temp, start))
      - path.add(temp)
      - temp = previous[temp[0]][temp[1]]
    - path.add(start)
    - Collections.reverse(path)
    - return path
  - visited[current[0]][current[1]] = true
  - List<int[]> neighbors = getNeighbors(current)
  - for (int[] neighbor : neighbors)
    - if (!visited[neighbor[0]][neighbor[1]])
      - int newDistance = distances[current[0]][current[1]] + map[neighbor[0]][neighbor[1]]
      - if (newDistance < distances[neighbor[0]][neighbor[1]])
        - distances[neighbor[0]][neighbor[1]] = newDistance
        - previous[neighbor[0]][neighbor[1]] = current.clone()
        - queue.offer(neighbor)
  - return new ArrayList<>()
- public static List<int[]> getNeighbors(int[] point)
    - List<int[]> neighbors = new ArrayList<>()
  - int x = point[1]
  - int y = point[0]
  - if (x > 0)
    - neighbors.add(new int[]{y, x - 1})
  - if (x < map[0].length - 1)
    - neighbors.add(new int[]{y, x + 1})
  - if (y > 0)
    - neighbors.add(new int[]{y - 1, x})
  - if (y < map.length - 1)

```
            - neighbors.add(new int[]{y + 1, x})
        - return neighbors
    - public static int calculateCost(List<int[]> path)
        - int cost = 0
        - for (int[] point : path)
            - cost += map[point[0]][point[1]]
        - return cost
    - public static List<int[]> findHighestScoringPath()
        - Route bestRoute = null
        - int maxScore = Integer.MIN_VALUE
        - for (Route route : dataset)
            - int score = calculateScore(route.getPath())
            - if (score > maxScore)
                - maxScore = score
                - bestRoute = route
        - return bestRoute.getPath()
    - public static int calculateScore(List<int[]> path)
        - int score = 0
        - for (int[] point : path)
            - score += interets.get(point)
        - return score
    - public static void getMap()
        - // Code pour récupérer la carte
    - public static void getSpeciaux()
        - // Code pour récupérer les points stratégiques et leurs intérêts
- class Route
    - private List<int[]> path
    - private int cost
    - public Route(List<int[]> path, int cost)
        - this.path = path
        - this.cost = cost
    - public List<int[]> getPath()
        - return path
    - public int getCost()
```

- return cost