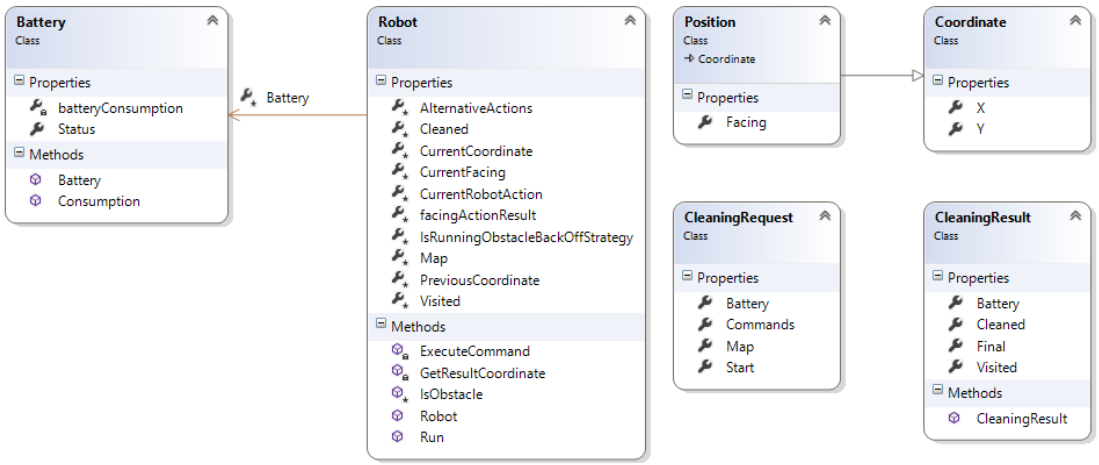
Technical Analysis for the Problem

First, the solution for this problem involves the design of a set of classes:



I separate the Robot Class and Battery Class to separate concerns and maintain its respective logic according to their objectives.

The classes CleaningRequest and CleaningResult are used by the Method **Run ()** located in **Robot** Class.

The class **Position** is a subclass from **Coordinate** Class. The Coordinate class stores the X and Y position where the Robot will clean. The **Position** Class additionally stores the **Facing** or orientation of the Robot i.e. North (N), South (S), East (E) or West (W).

Robot Class

It contains the method to execute the instructions supplied in **CleaningRequest** class which is supplied in the console program.

The following section describes all the properties used in this class:

```
/// <summary>
/// Contains the Map used by the Robot to Clean
/// </summary>
protected string[][] Map { get; set; }

/// <summary>
/// Contains the coordinates X, Y where the Robot walks
/// </summary>
protected List<Coordinate> Visited { get; set; } = new List<Coordinate>();
/// <summary>
/// Contains the coordinates X, Y where the cleans
/// </summary>
```

```
protected List<Coordinate> Cleaned { get; set; } = new List<Coordinate>();
/// <summary>
/// Contains a matrix to store the result facing when an command (Turn Left or Turn
Right) is applied knowing its initial facing
/// </summary>
protected string[][] FacingActionResult { get; set; } = new string[][] { new string[2] {
"W", "E" }, new string[2] { "E", "W" }, new string[2] { "N", "S" }, new string[2] { "S",
"N" } };
/// <summary>
/// Battery object that stores Status and Gets the Comsumption of Battery for each action
/// </summary>
protected Battery Battery { get; set; }
/// <summary>
/// Current Coordinate resulting of executing the Current Command
/// </summary>
protected Coordinate CurrentCoordinate { get; set; }
/// <summary>
/// Previous Coordinate used as backup when the current coordinate has an invalid value
or state
/// </summary>
protected Coordinate PreviousCoordinate { get; set; }
/// <summary>
/// Current Facing where the Robot is looking
/// </summary>
protected Facing CurrentFacing { get; set; }
/// <summary>
/// Current command that is being executed
/// </summary>
protected RobotAction CurrentRobotAction { get; set; }
/// <summary>
/// Boolean that indicates if the Robot is running a Obstacle Backoff Strategy
/// </summary>
protected bool IsRunningObstacleBackOffStrategy { get; set; }
/// <summary>
/// Matrix to map the strategies commands if there are an obstacle
/// </summary>
protected string[][] AlternativeActions { get; set; } = new string[][] { new string[] {
"TR", "A" }, new string[] { "TL", "B", "TR", "A" }, new string[] { "TL", "TL", "A" }, new
string[] { "TR", "B", "TR", "A" }, new string[] { "TL", "TL", "A" } };
```

The `protected string[][] FacingActionResult` is the representation of the following table:

Initial Facing	Applied Action	Resulting Facing
N	TL	W
N	TR	E
S	TL	E
S	TR	W
E	TL	N
E	TR	S
W	TL	S
W	TR	N

And the **AlternativeActions** are a collection of enumerated actions to be executed in case there is any obstacle.

The Boolean method **IsObstacle** indicates if there is an obstacle when it is being executed the current command. An obstacle depends of Current Coordinate, Map Position (X and Y) and Battery Status:

```
protected bool IsObstacle() => (CurrentCoordinate.X < 0 || CurrentCoordinate.Y < 0 ||
CurrentCoordinate.X >= Map.GetLength(0) ||CurrentCoordinate.Y >= Map.GetLength(0) ||
Map[CurrentCoordinate.Y][CurrentCoordinate.X] == "C" ||
Map[CurrentCoordinate.Y][CurrentCoordinate.X] == "null" || Battery.Status -
Battery.Consumption(CurrentRobotAction) < 0);
```

Method: `public CleaningResult Run(CleaningRequest request)`

This method contains the logic to resolve every position where the Robot will be located and the process to execute the commands indicated in the Request Object.

Every command in **CleaningRequest.Commands** changes the Battery and Position represented by the object property **Battery** and the **CurrentCoordinate** Property.

The Algorithm for this Method is a very simple implementation which by means of iterating over **CleaningRequest.Commands** and using the method **ExecuteCommand** where the State is Changed (Battery and Position). In this iteration, it is evaluated if there is an Obstacle using the Boolean function **IsObstacle().**

If there is an obstacle, the **AlternativeActions** is iterated and resolved according the collections of commands supplied using the same Method **ExecuteCommand**.

At the end, all the state of the Robot class is returned.

```
var result = new CleaningResult();
Map = request.Map;
Battery = new Battery { Status = request.Battery };
CurrentCoordinate = request.Start;
PreviousCoordinate = CurrentCoordinate;
CurrentFacing = (Facing)Enum.Parse(typeof(Facing), request.Start.Facing);
Visited.Add(new Coordinate { X = CurrentCoordinate.X, Y = CurrentCoordinate.Y });

foreach (var command in request.Commands)
{
 CurrentRobotAction = (RobotAction)Enum.Parse(typeof(RobotAction), command);
 ExecuteCommand();

if (IsObstacle())
{
  CurrentCoordinate = PreviousCoordinate;
  IsRunningObstacleBackOffStrategy = true;
      var alternativeIndex = 0;
      do
        {
           var altCommands = AlternativeActions[alternativeIndex];
           foreach (var altCommand in altCommands)
             {
```

```
                CurrentRobotAction = (RobotAction)Enum.Parse(typeof(RobotAction),
altCommand);
                    ExecuteCommand();
              }

             if (!IsObstacle())
             {
                  break;
              }
             else
              {
                 CurrentCoordinate = PreviousCoordinate;
                 IsRunningObstacleBackOffStrategy = true;
               }

              alternativeIndex++;
         }
         while (IsRunningObstacleBackOffStrategy || alternativeIndex <=
AlternativeActions.GetLength(0));

                      IsRunningObstacleBackOffStrategy = false;
                  }
             }

result.Visited = Visited.OrderBy(m => m.X).ThenBy(m => m.Y).ToList();
result.Cleaned = Cleaned.OrderBy(m => m.X).ThenBy(m => m.Y).ToList();
result.Final = new Position { X = CurrentCoordinate.X, Y = CurrentCoordinate.Y, Facing =
CurrentFacing.ToString() };
result.Battery = Battery.Status;

return result;
```

The above code shows part of the implemented code for the Method **Run**. The yellow part shows the normal execution and the green part shows the alternative execution when the Robot finds an Obstacle.

The Method **ExecuteCommand** process and changes the State of the Robot:

```
private void ExecuteCommand()
        {
if (Battery.Status - Battery.Consumption(CurrentRobotAction) < 0)
{
    return;
}

Battery.Status = Battery.Status - Battery.Consumption(CurrentRobotAction);

if (CurrentRobotAction == RobotAction.A || CurrentRobotAction == RobotAction.B)
 {
    PreviousCoordinate = CurrentCoordinate;
    CurrentCoordinate = GetResultCoordinate(CurrentFacing,
                          CurrentRobotAction, CurrentCoordinate);
}
else if (CurrentRobotAction == RobotAction.TL ||
         CurrentRobotAction == RobotAction.TR)
{
   CurrentFacing =
```

```
        (Facing)Enum.Parse(typeof(Facing),

        FacingActionResult[(int)CurrentFacing][(int)CurrentRobotAction]);
            }

if (!IsObstacle())
{
   if ((CurrentRobotAction == RobotAction.A ||
        CurrentRobotAction == RobotAction.B) &&
       (!Visited.Any(m => m.X == CurrentCoordinate.X && m.Y == CurrentCoordinate.Y)))
   {
       Visited.Add(CurrentCoordinate);
   }
   else if ((CurrentRobotAction == RobotAction.C) &&
       (!Cleaned.Any(m => m.X == CurrentCoordinate.X && m.Y == CurrentCoordinate.Y)))

   {
        Cleaned.Add(CurrentCoordinate);
   }
}


}
```

The method **ExecuteCommand** uses the method **GetResultCoordinate** to resolve the next position when there is executed a command action (Advance or Back) according the **CurrentFacing** (North, South, East or West) and passing the **Current Coordinate**.