Atypon Training Program 2023

AN OPTIMIZED 'KAREL THE ROBOT' ATYPON TRAINING PROGRAM 2023 USING STRATEGIC PROBLEM-SOLVING

BY: YUSSIF ABDALLA, The University of Jordan

I. ABSTRACT

This report illustrates the solution of the Aypon's 2023 assignment. The assignment is based on the famous Karel robot programming environment. Karel can divide a given map into 4 + 4, where the following constraints were observed:

- The inner chambers should be the biggest possible equal squares
- The outer chambers should be equal in size, and they should be L-shaped
- Optimized number of double lines of beepers may be used
- Karel should achieve his task with the lowest number of moves

An optimized code is successfully achieved, which maintains effectiveness, minimum moves and used beepers and minimal used commands. Five optimization strategies were successfully used to satisfy the needed assignment constraints. The minimum code requirement was accomplished by establishing advanced Karel commands, while multi-tasking was used to tackle the minimum moves constraints. Beepers placement was done intelligently by checking existing beepers before moving. Finally, exceptional routines were established that pertain to selected map dimensions.

II. INTRODUCTION

Over five decades ago, a Stanford graduate student named Rich Pattis perceived an idea that would help novice programmers to think like professional programmers in solving problems. Rich designed an introductory programming environment in which

students teach a robot to solve simple problems, hence they will learn the logical sequencies in problem solving [1].

The Atypon assignment consists of drawing a given map into 4 + 4, with the inner chambers should be the biggest possible equals squares, while the outer chambers should be equal in size, and they should be L-shaped. Double lines of beepers are allowed [2].

The proposed solution strives to optimize the given task by minimizing the number of required robot's moves, reducing the number of beepers used, and enhancing code efficiency by creating reusable functions.

This report presents a detailed step-by-step problem-solving approach, outlining the thought process behind each decision made. It also highlights the optimizations implemented and the iterative evolution of Karel code during the program's development.

III. SOLUTION METHODOLOGY AND OPTIMIZATION

A code optimizing technique that can be used for solving this Karel assignment is the modular design approach. The proposed code is divided into smaller and self-contained modules. Each module is responsible for a specific function and can be developed and tested independently.

A flow chart representation of the solution program conceptual design will be established. This flow chart will give the logical needed steps in achieving Atypon's Karl assignment. Basic Karel commands will be used to build more advanced commands that helps in executing the proposed flowchart and enable easier debugging and troubleshooting in a simpler way.

2.1 Karel Novel Advance Commands

To simplify the solution of the given problem I propose using the following advance commands.

3.1.1 faceNorth(), faceEast(), faceSouth(), faceWest()

After analyzing the problem requirements, I formulated a thoughtful and effective approach to tackle the problem assignment. However, before diving into coding, I realized that the 'turnLeft()' command provided in Karel's reference card posed challenges in terms of convenience, readability, and debugging. Understanding 'turnLeft()' required determining Karel's initial direction and performing multiple 90-degree rotations based on the number of consecutive turnLeft() commands which proved to be cumbersome.

To illustrate this difficulty, the following hypothetical task codes demonstrate the inconvenient use of the turnLeft command.

```
// Inconvenience of using turn left command

public void run() {
 move();
 pickBeeper();
 move();
 turnLeft();
 move();
 turnLeft();
 turnLeft();
 turnLeft();
 move();
 move();
 putBeeper();
 move();
 }
```

So, I came up with an idea: why not to introduce new commands that allow specifying the desired direction for Karel to face? Hence, the faceNorth, faceEast, faceSouth, faceWest methods were created that can help to command Karel to face a specified direction in an easy and convenient way. **The detailed codes of these procedures maybe found in Appendix A.**

3.1.2 moveNoSteps(int noSteps, boolean withBeepers)

A newly developed method, which is called 'moveNoSteps', enables Karel to move a specified number of steps. Instead of repeatedly rewriting the 'move()' operation in the code, one can simply invoke the 'moveNoSteps' method, making the code much easier to manage and understand.

Initially, two methods were used; they were called 'moveNumStepsBeepers' and 'moveNumStepsNoBeepers'. However, I realized that it would be more convenient to include beepers as a parameter in a general 'moveNoSteps' method. Now, the 'moveNoSteps' method accepts an additional parameter called 'withBeepers', which allows us to specify whether you want beepers to be dropped while moving.

This method optimizes the usage of beepers by ensuring that no beepers are present before dropping a new one. By doing so, it guarantees that Karel only has one beeper per corner, preventing any overlapping or unnecessary additional beepers in the solution.

The 'withBeepers' also ensures that Karel is carrying beepers, which shouldn't be a problem as we are provided enough number of beepers in the problem. **The detailed code of this procedure may be found in Appendix B.**

3.1.3 moveTillEnd(boolean withBeepers)

The moveTillEnd method performs the same action as the previously discussed moveNoSteps method, with one difference that you do not need to specify the number of steps. In moveTillEnd, Karel continues moving until it reaches a wall, ensuring continuous movement until an obstacle is encountered.

To ensure that Karel can move safely, one must always check that the front is clear before making a move. This is why I used the 'frontIsClear' condition depicted in the **code of this procedure in Appendix C.**

An additional feature of the moveTillEnd method, which sets it apart from moveNoSteps, is that it keeps track of the number of steps Karel moves until encountering an obstacle then return it. This feature is particularly valuable and essential for my solution.

2.2 Problem Solution Approach

A flow chart is the best structured way to organize ideas and thoughts in any process without getting into details. It provides a clear and intuitive visualization of the sequence of steps and the logic behind decisions. It really helped me a lot for the Karel assignment. Figure 1 depicts the program conceptual design flow chart that was used in solving the given assignment.

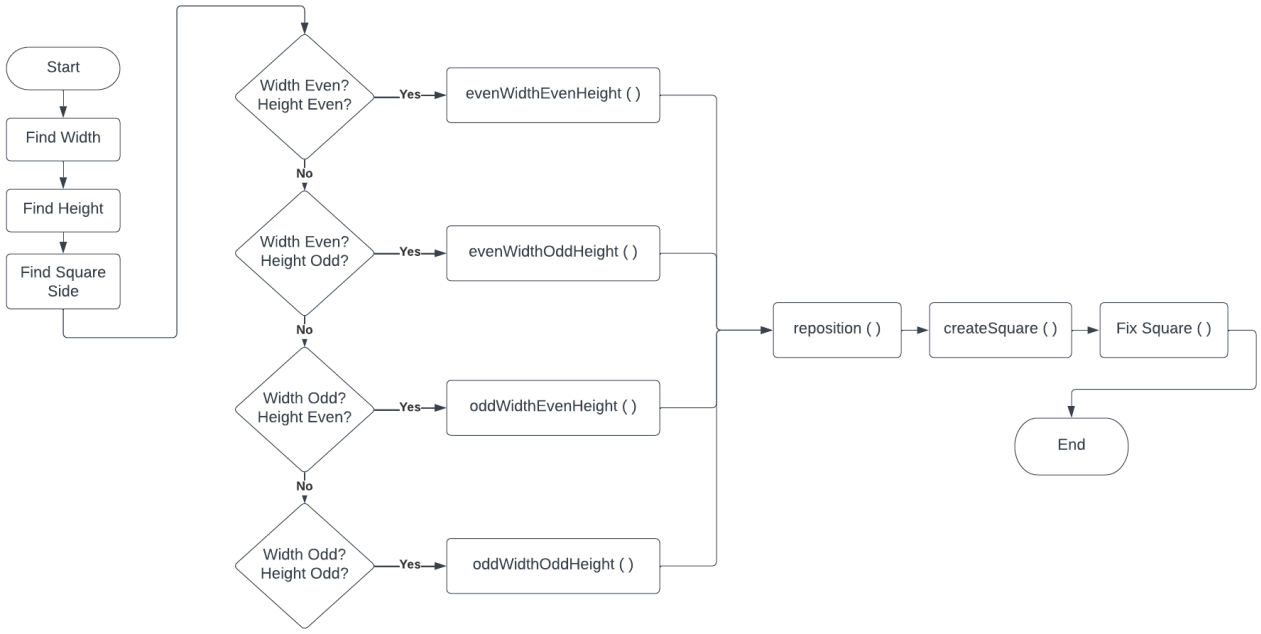


Figure 1: Program conceptual design flow chart

Find Width

To begin solving the problem, we need to determine the width of the map. Initially, Karel is facing east and starts from the corner (0, 0). By calling the moveTillEnd method, Karel can move until reaching the end of the x-axis, thus allowing us to determine the width of the map. We store this value for further use.

```
width = moveTillEnd(false);
```

Find Height

To find the height of the map, one should instruct Karel to face north and call the moveTillEnd method. This enables Karel to move until reaching the end of the Y-axis, thus determining the height of the map.

This step has been optimized to lower the number of steps Karel makes by finding the height and adding the center vertical line of beepers at the same time.

```
turnAround();
moveNoSteps(width/2, false);
faceNorth();
height = moveTillEnd(true);
```

Karel turns to face the opposite direction and then moves width/2 steps. This places Karel at the center of the x-axis. Now that Karel is positioned in the center, he turns to face north and continues moving while placing beepers until he reaches the end.

Find Square Side

To ensure that the inner chamber contains the largest equal squares while maintaining an L shape for the outer chamber, we need to determine the maximum size of the square containing the inner chamber. It is crucial to emphasize that the square should not occupy the entire space as it would adversely affect the L shape of the outer chamber.

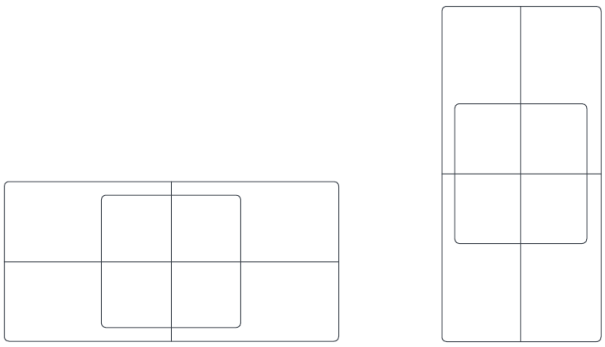


Figure 2: Possible square and map orientations

The size of the square should be determined by the smallest dimension of our map, whether it is in portrait or landscape orientation.

```
squareSide = (width <= height) ? width-2 : height-2;
```

To ensure that the square does not affect the L shape of the outer chamber, I subtracted two units from each side of the square. This removes one line of corners above the square and one line below it.

Check the widths and height's divisibility by 2

The next actions of my program depend on the dimensions of the map. Each scenario invokes a specific method, and while each method executes different commands, I ensured that regardless of the method invoked, Karel will ultimately reach the same corner in order to optimize the number of moves.

Here are all the possibilities:

a. Even width even height
b. Even width odd height
c. Odd width even height
d. Odd width odd height

```
if(width%2 == 0 && height%2 == 0){
        evenWidthEvenHeight(squareSide, width, height);
} else if(width%2 != 0 && height%2 == 0){
        oddWidthEvenHeight(squareSide, width, height);
} else if(width%2 == 0 && height%2 != 0){
        evenWidthOddHeight(squareSide, width, height);
} else {
        oddWidthOddHeight(squareSide, width, height);
}
```

The goal in this step after invoking one of the four functions, is to have the horizontal line drawn and no matter what scenario Karel has encountered he should land on the same the corner shown in the image. Figure 3 illustrates all the possible cases.

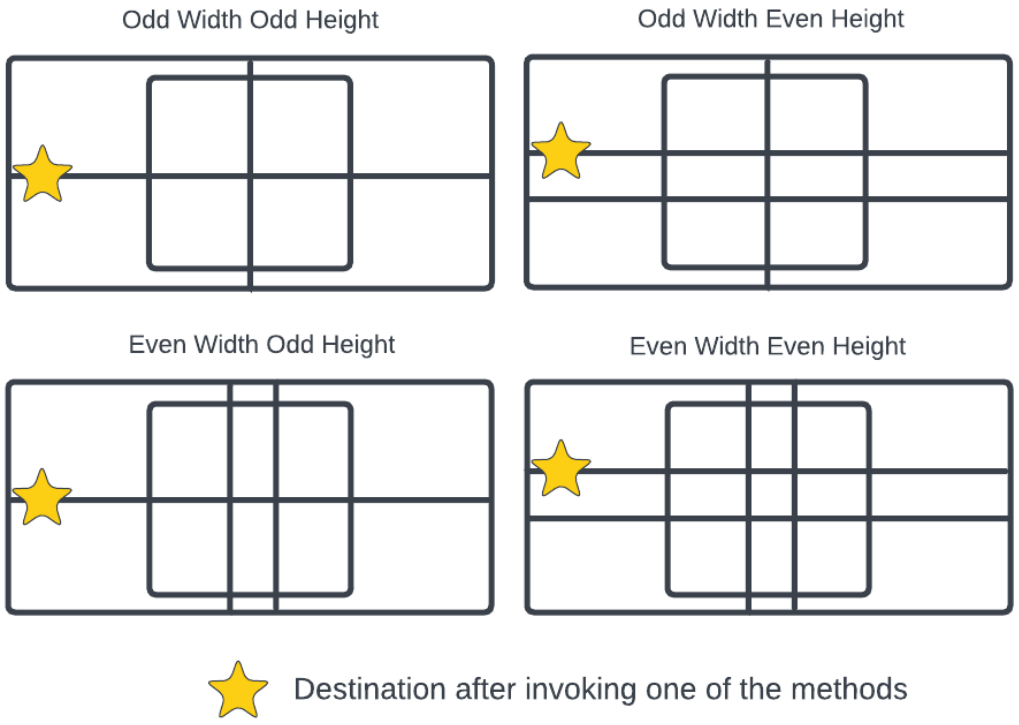


Figure 3: Common positions for all Karel scenarios

After determining the width and height of Karel's word and drawing the initial vertical line, Figure 4 illustrates how Karel's map should appear.

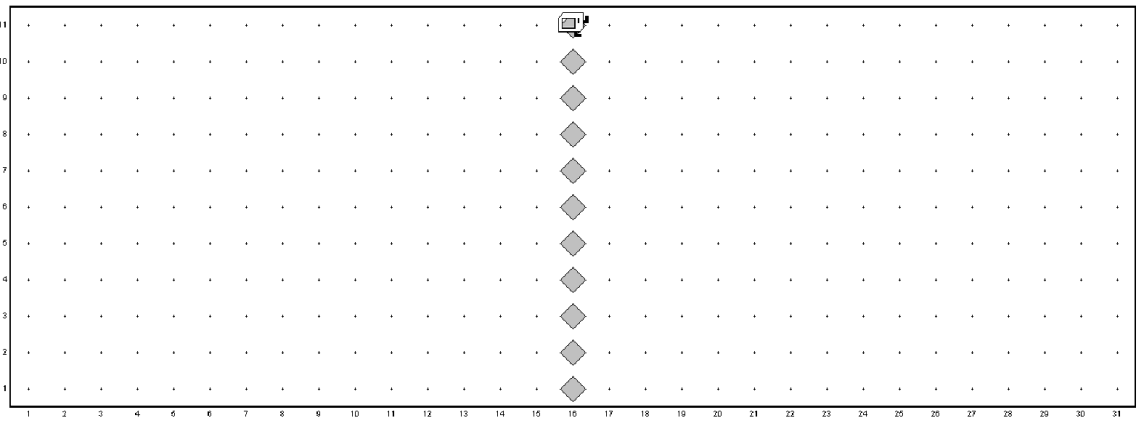


Figure 4: Optimizing finding the height

Depending on the width, we have two cases:

1. If the width is odd:
   In this case, we only need one vertical line to divide the chambers. Once the line is drawn, we are ready to proceed to the new commands.
2. If the width is even:
   In this case, we must create an additional vertical line to correctly divide the chambers and maintain symmetry for that I created uTurn1 method.

```
public void uTurn1(){
        faceEast();
        move();
        faceSouth();
        moveTillEnd(true);
}
```

This method should be invoked whenever the width is even. It instructs Karel to make a U-turn and continue moving towards the end of the map while placing beepers along the way. After returning from this method, the resulting Karel's map should end up looking like the one shown in Figure 5.

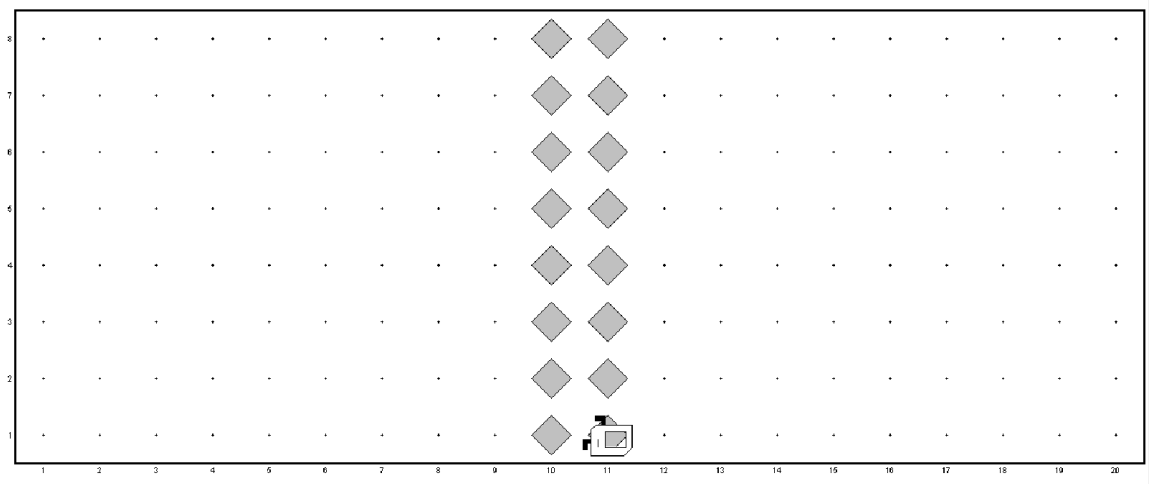


Figure 5: Karel after calling the uTurn1 method

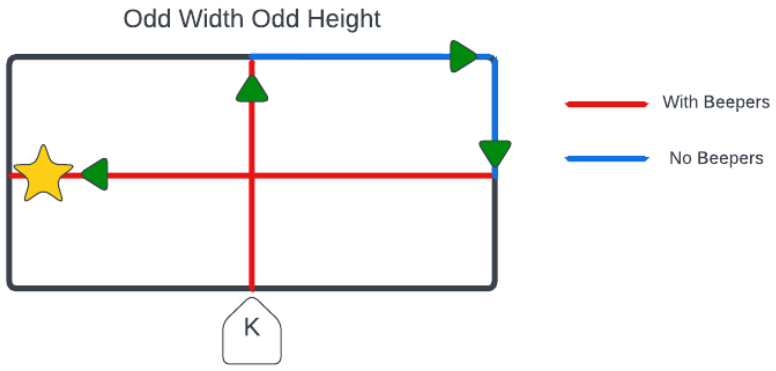
Now that the uTurn1 method is fully explained, one can see the oddWidthOddHeight and odd evenWidthOddHeight functions

Odd width odd height

```
public void oddWidthOddHeight(int height){
        faceEast();
        moveTillEnd(false);
        faceSouth();
        moveNoSteps(height/2, false);
        faceWest();
        moveTillEnd(true);
}
```

The oddWidthOddHeight is the simplest method. After drawing the vertical line, it directs Karel to move to the rightmost position. Then, Karel moves towards the south by height/2 steps, where Karel can begin drawing the horizontal center line of beepers.

Here's an image that illustrates Karel's movements.

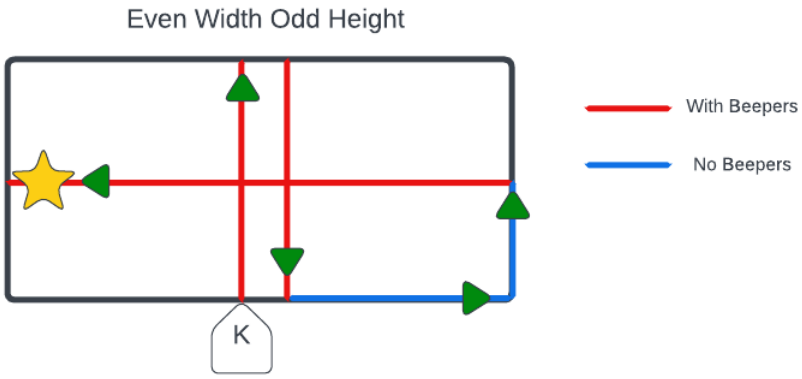


Even width odd height

```
public void evenWidthOddHeight(int height){
        uTurn1();
        faceEast();
        moveTillEnd(false);
        faceNorth();
        moveNoSteps(height/2, false);
        faceWest();
        moveTillEnd(true);
}
```

Since the width is even, Karel needs to make a U-turn and move towards the end of the map. Then, Karel faces east and proceeds to the leftmost position. Afterward, Karel moves north by height/2 steps and continues westward, placing beepers along the way.

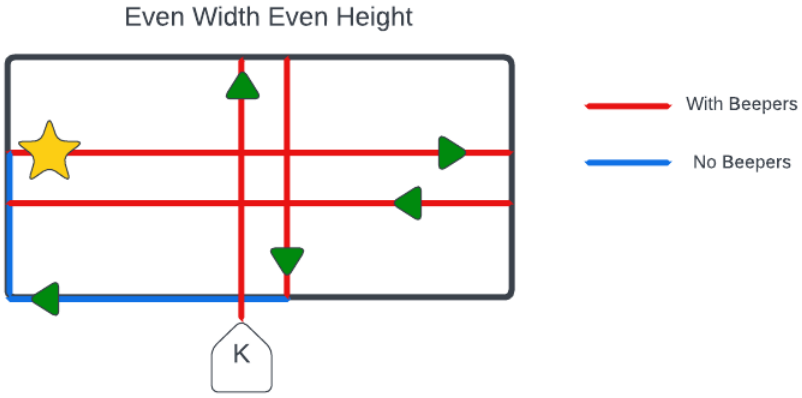
Here's an image that illustrates Karel's movements.



Now that we saw how the program deals with even width, let's see what happens when the height is even using the oddWidthEvenHeight and evenWidthEvenHeight methods.

Even width even height

```
public void evenWidthEvenHeight(int height){
        uTurn1();
        faceWest();
        moveTillEnd(false);
        faceNorth();
        moveNoSteps(height/2, false);
        faceEast();
        moveTillEnd(true);
        uTurn2();
        faceNorth();
        move();
}
```



Karel will make a U-turn and continue moving until reaching the wall. Then, Karel will proceed to move westward until reaching the leftmost side of the map. Afterward, Karel will move north by height/2 steps and then head east towards the rightmost side of the map, while placing beepers along the way.

The reason why Karel moves west in this method is that, since the height is even, he needs to create two lines of beepers. To ensure that Karel reaches the common point between all four methods, he must make a U-turn after completing the necessary movements.

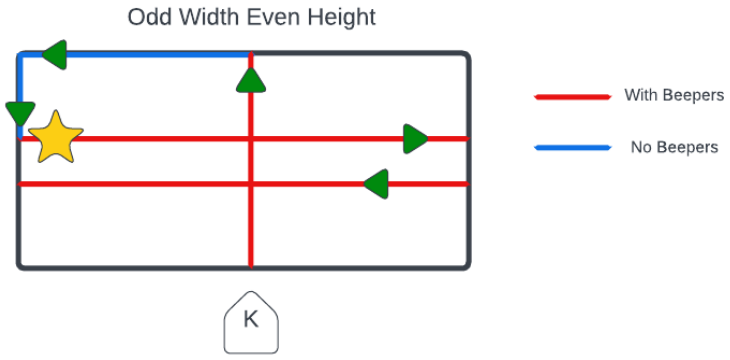
To make a U-turn for the horizontal line we have the uTurn2 method.

```
public void uTurn2(){
        faceSouth();
        move();
        faceWest();
        moveTillEnd(true);
}
```

uTurn2 does the exact same as uTurn1 just on a different orientation. Notice that Karel is a step below the common point that's why he moves a step to the north.

Odd width even height

```
public void oddWidthEvenHeight(int height){
        faceWest();
        moveTillEnd(false);
        faceSouth();
        moveNoSteps(height/2-1, false);
        faceEast();
        moveTillEnd(true);
        uTurn2();
        faceNorth();
        move();
}
```



After drawing the vertical line, Karel turns to face west and moves towards the leftmost side of the map. It's important to note that Karel does not make a vertical U-turn since the width is odd.

Next, Karel faces north and proceeds to move height/2 - 1 steps. Then, Karel continues moving east towards the rightmost side. Since the height is even, Karel will have to make a horizontal U-turn to create a double line of beepers. After creating the double line, Karel takes a step upwards to reach the common point.

Repositioning

Now that we have drawn the vertical and horizontal lines, we need to create our square for the inner chambers to appear. However, before that, we must position Karel at the desired corner to begin creating the square from.

Where should we position Karel?

After considering various approaches, I concluded that the optimal position for placing Karel would be at one of the corners along the perimeter of the square. This choice minimizes the number of moves Karel needs to make. On the other hand, if Karel starts from the center of the square, additional moves would be required.

```
public void reposition(int squareSide, int width){
        turnAround();
        if(squareSide%2 != 0 && width%2 == 0){
                moveNoSteps(width/2 - squareSide/2 - 1, false);
        } else {
                moveNoSteps(width/2 - squareSide/2, false);
        }
}
```

Before making a square Karel turns around and moves

`(width/2 - squareSide/2) steps without beepers`

This places Karel on the perimeter of the square. This is how Karel's map should look like as shown in Figure 6.

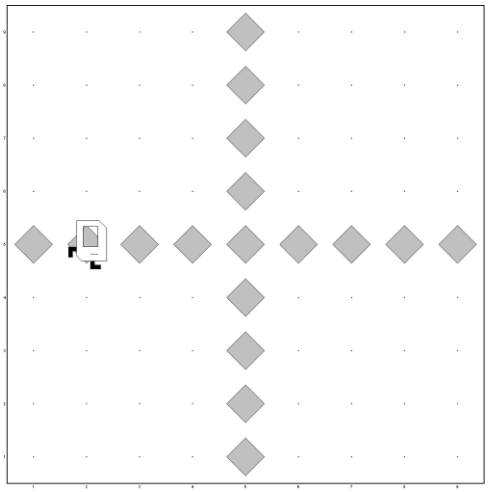


Figure 6: Positioning Karel 1

In this image Karel turned around and moved 1 step

width/2 – squareSide/2 =

9/2 – 7/2 =

1 step

An edge case occurs when the squareSide is odd but the width is even. This is handled by moving: width/2 – squareSide/2 . Figure 7 illustrates an image that shows such a case.

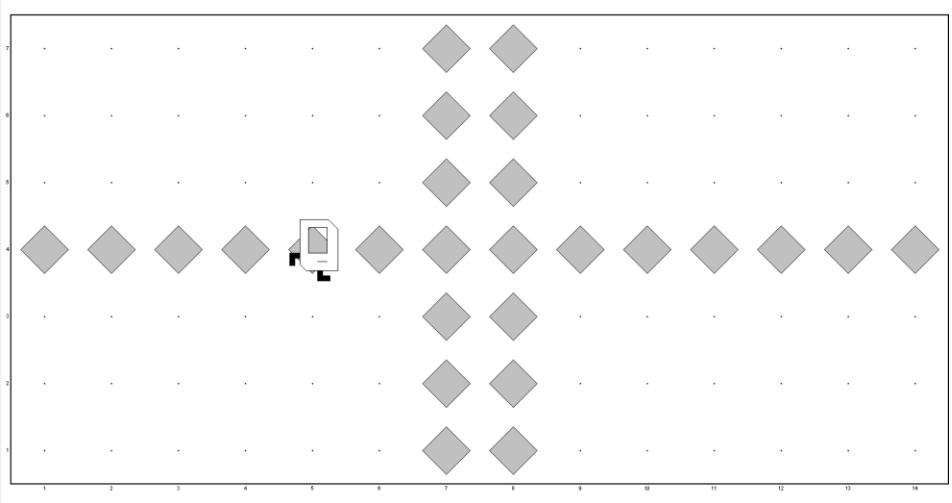


Figure 7: Positioning Karel 2

Karel moves

width/2 – squareSide/2 -1 =

14/2 - 5/2 - 1 =

4 steps

Create Square

Now that Karel has successfully repositioned, he can finally create the square. Creating the square is easy; it's just a sequence of moving south, east, north, west, and then south again.

```
public void createSquare(int squareSide){
        faceSouth();
        moveNoSteps(squareSide/2 ,true);

        faceEast();
        moveNoSteps(squareSide-1,true);

        faceNorth();
        moveNoSteps(squareSide-1, true);

        faceWest();
        moveNoSteps(squareSide-1, true);

        faceSouth();
        moveNoSteps(squareSide/2 , true);
}
```

Karel moves south squareSide/2 steps, then moves east squareSide – 1 steps, then moves north squareSide – 1 steps, then moves west squareSide – 1 steps, and finally returns to the initial point by moving squareSide/2 steps. Figure 8 depicts the results.

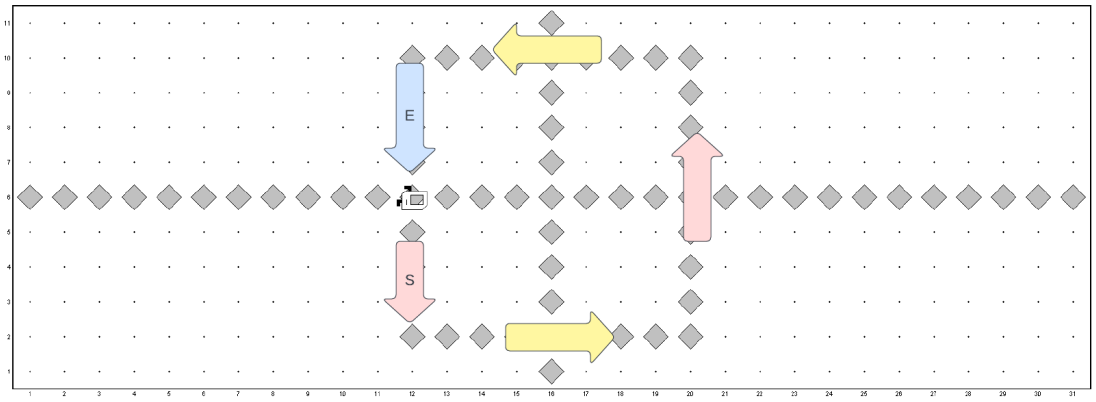


Figure 8: Square drawing

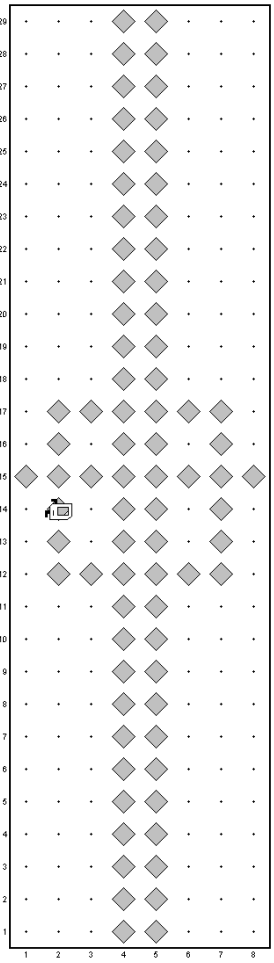
Fixing the square

All these steps work with most cases, but there are a few special cases where the inner chamber sizes won't match. To address this, the fixSquare method is added to resolve the issue.

All of these cases can be tested by Karel by moving, but I found that the approach of examining the square side, width, and height is more optimized and requires fewer steps for Karel to make.

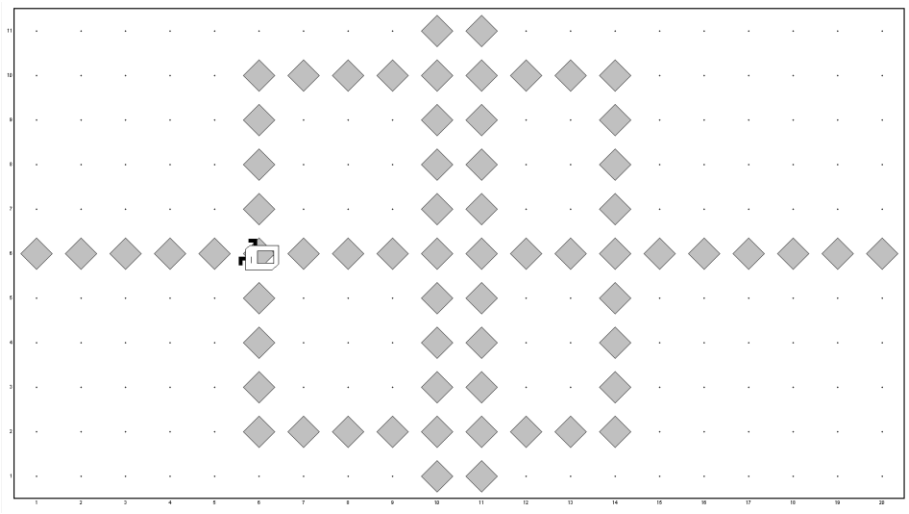
```
public void fixSquare(int squareSide, int width, int height){
        if(squareSide%2 == 0 && width%2 == 0 && height%2 != 0){
                faceEast();
                moveNoSteps(squareSide-1, true);
        } else if(squareSide%2 != 0 && width%2 == 0 && height%2 != 0){
                moveNoSteps(squareSide/2, false);
                uTurn3(squareSide);
        } else if (squareSide%2 == 0 && width%2 != 0 && height%2 == 0){
                moveNoSteps(squareSide/2-1, false);
                uTurn3(squareSide);
        }
}
```

Case1: when square side is even, width is even, and height is odd



This can be easily fixed by facing east and moving (squareSide – 1) steps.

Case2: when square side is odd, width is even, and height is odd



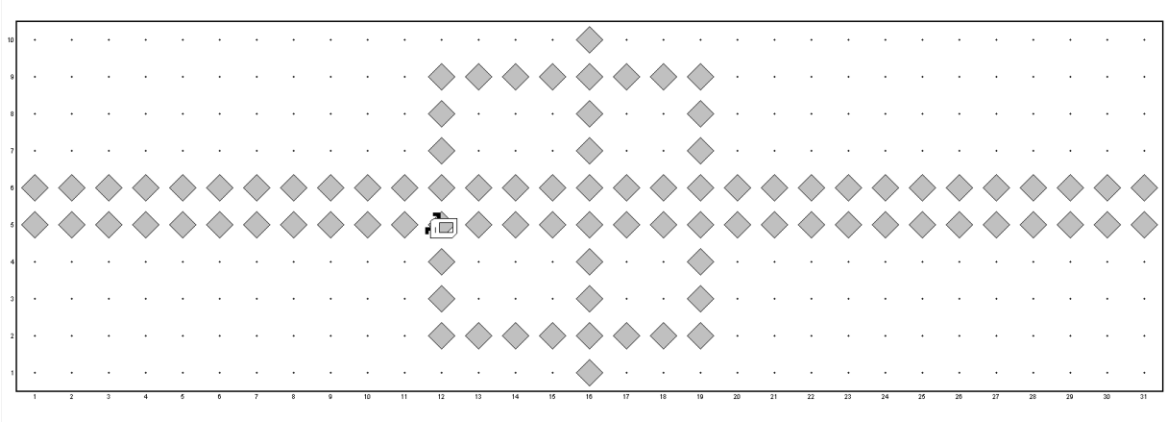
This case can be solved by moving to the bottom left corner of the square and making a U-turn while placing beepers. However, instead of allowing Karel to keep moving until he reaches a wall, you only want him to move until he reaches the perimeter of the square.

Here's the new U-turn method we will use to solve such case.

```
public void uTurn3(int squareSide){
        faceEast();
        move();
        faceNorth();
        moveNoSteps(squareSide-1, true);
}
```

Note that we used moveNoSteps method with (squareSide - 1) steps instead of moveTillEnd.

Case3: when square side is even, width is odd, and height is even.



This is a very similar case as case 2 the only difference in the solution is that Karel moves (squareSide/2 - 1) moves instead of (squareSide/2) because moving (squareSide/2) steps gets us out of the square's perimeter.

IV. CONCLUSIONS

In this work diversity of Map scenarios were successfully executed with the Karel robot, which meets all the assignment constraints. The minimum number of Karel moves were achieved by implementing optimization strategies.

Exceptional maps are identified and handled by the program and the Lowest possible number of lines were achieved by implementing new reusable functions.

Finally, the solution is accomplished with minimum number of placed beepers using lookahead strategies.

V. REFERENCES

[1] Eric Roberts, Karel the Robot – Learn Java, Stanford University, September 2005.

[2] Motasem Diab, Karel Assignment, Atypon Training Program 2023.

VI. APPENDICES

Appendix A

```
public void faceEast(){
   while(notFacingEast()){
      turnLeft();
   }
}

public void faceWest(){
   while(notFacingWest()){
      turnLeft();
   }
}

public void faceNorth(){
   while(notFacingNorth()){
      turnLeft();
   }
}

public void faceSouth(){
   while(notFacingSouth()){
      turnLeft();
   }
}
```

Appendix B

```
public void moveNoSteps(int noSteps, boolean withBeepers){
        for(int i = 1; i<=noSteps ;i++){
                if(noBeepersPresent() && withBeepers){
                        putBeeper();
                }
                move();
        }
        if(noBeepersPresent() && withBeepers){
                putBeeper();
        }
}
```

Appendix C

```
public int moveTillEnd(Boolean withBeeper){
        int counter = 1;
        while(frontIsClear()){
        counter++;
        if(noBeepersPresent() && withBeeper) putBeeper();
                move();
        }
        if(noBeepersPresent() && withBeeper) putBeeper();
        return counter;
}
```