Richochet Robots

Jonas Kramer, Lauritz Andersen

February 21, 2024

1 Introduction

In this assignment, we were to create a game based on the board game *Ricochet Robots*. Our implementation of the game is about controlling a couple of robots, trying to make one them reach a goal on the game board. However there are obstacles along the way, so one needs to think carefully in order to succeed!

We were to create an object-oriented design of the game that would make it easy to expand it with new rules and elements. There were a few mandatory requirements for what should be included for the basis of the game to work. The basis game only includes the board, the robots, walls and goals. Where as in our implementation, we have added the expansions in form of mirrors and trampolines, making the game even more fun to play.

The assignment was divided into several parts. The first sub-task was to show a board in the terminal. We discussed how to best achieve this, and we chose to represent the board with 2D-array of strings. The second was to create a class hierarchy to represent different game elements, e.g. the robots. We created an UML-diagram to get a decent overlook of our program and gave a lot of thought into which classes should inherit from which and which were to associate to each other. Finally, in the third sub-task, we had to put the different parts together into a functioning game. We did this by implementing a Board and a Game class, wherein the first sets up the game board, keeps track of all elements and moves the robots, whereas the other handles all interaction with the player and allows for control of robots with the arrow keys. Then as the last sub-task, we had to implement the aforementioned extensions, where we ended up settling for a "Mirror"-extension that would throw the robot to different position against its benefit, and a "Trampoline"-extension, that as a strategic advantage, would allow to robot to e.g. jump over walls, resulting in distinctly fun gameplay.

2 Problem analysis and design

2.1 UML and design

To create the game, we knew we would have to have some kind of class hierarchy to represent different game elements.

The game takes place on a a board with r $\times$ c fields, i.e. r rows and c columns, where a number of robots can be moved around. We knew this would mean we needed to have a Board with some rows and some columns and some robots appearing on it. This is heavily suggesting we need a Robot class and a Board class, so we noted that.

Each robot starts in a separate field on the plate, and can then be moved by sliding in one of the four directions north, south, east, or west. This is quite obviously suggesting that the robot needs a "slide"/"move" method.

The goal of the game is to get one of the robots moved to a goal field in the lowest number of moves possible. The main challenge of the game is that when a robot starts sliding in one direction, then it continues to slide in that direction until it hits either a wall or another robot. This is suggesting that we are going to need to define some walls on the board as a class, and that we would somehow in the code need to tell the robot to stop when hitting either one of these wall or another robot. To complete a game, a robot must stop in a goal field. It does not count as a solution if a robot simply slides through the Goal

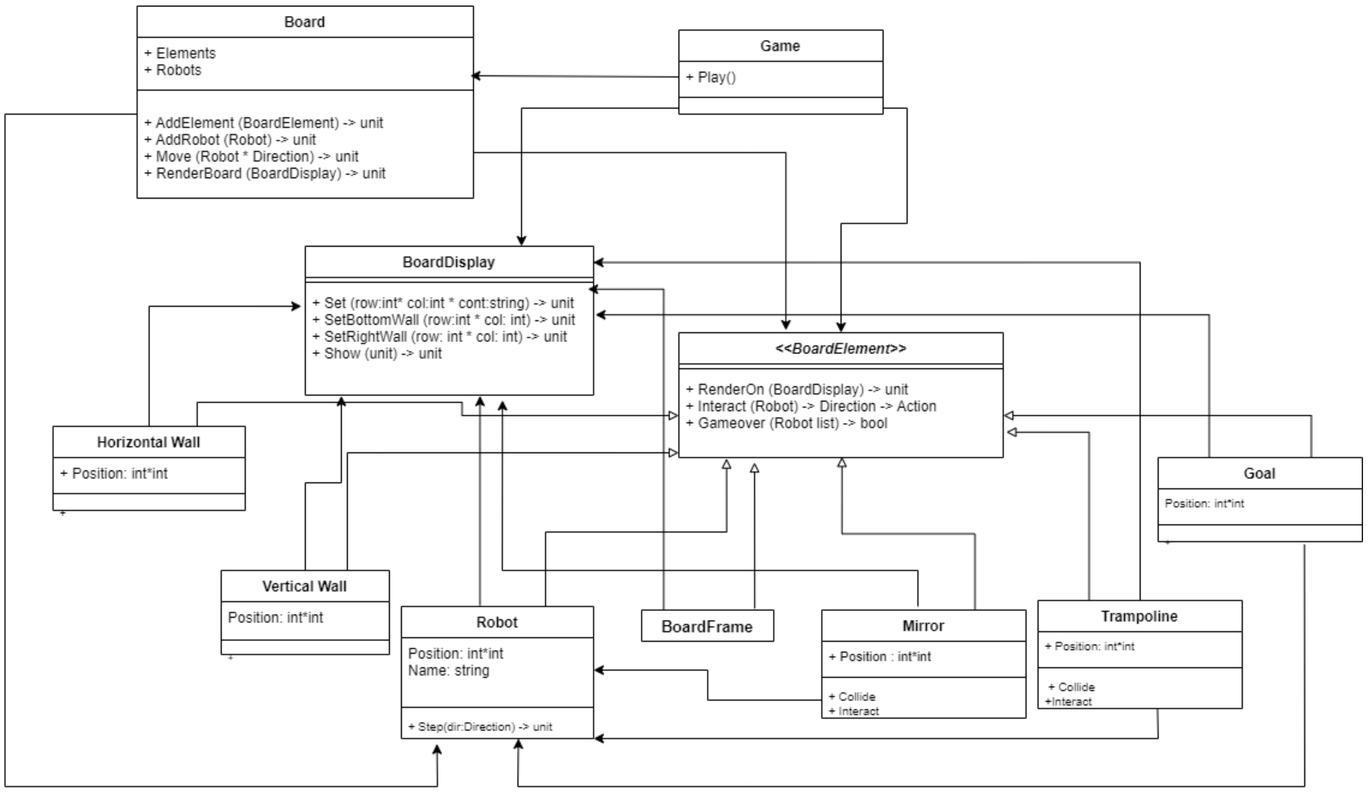
Board
+ Elements
+ Robots
+ AddElement (BoardElement) -> unit
+ AddRobot (Robot) -> unit
+ Move (Robot * Direction) -> unit
+ RenderBoard (BoardDisplay) -> unit
Game
+ Play()
BoardDisplay
+ Set (row:int* col:int * cont:string) -> unit
+ SetBottomWall (row:int * col: int) -> unit
+ SetRightWall (row: int * col: int) -> unit
+ Show (unit) -> unit
<<BoardElement>>
+ RenderOn (BoardDisplay) -> unit
+ Interact (Robot) -> Direction -> Action
+ Gameover (Robot list) -> bool
Horizontal Wall
+ Position: int*int
Goal
Position: int*int
Vertical Wall
Position: int*int
Robot
Position: int*int
Name: string
+ Step(dir:Direction) -> unit
BoardFrame
Mirror
+ Position : int*int
+ Collide
+ Interact
Trampoline
+ Position: int*int
+ Collide
+Interact

Figure 1: UML diagram displaying all of our classes with their methods and attributes.

field. This suggests that the robot's position must be the same as that of the target field for the game to be over and that we probably need a Goal-class in order to create this Goal-field for the robot to slide to.

In the assignment it was given that we needed to create a BoardDisplay class, a HorizontalWall and VerticalWall class, as well as a Robot, BoardFrame, BoardElement, Goal, Board and a Game class.

Our reasoning was that HorizontalWall, VerticalWall, Robot, BoardFrame and Goal were all Board Elements, so they would have to inherit from BoardElement. Apart from this "Goal" needed to associate with Robot, but Robot would not need to associate to Goal. This is because it is only relevant for Goal to know whether a Robot is inside it and it is not relevant for the Robot to know whether it is inside a Goal. Therefore Robot and Goal is connected in our UML-diagram with a unidirectional associate arrow. We applied the same logic for our extensions, Mirror and Trampoline, which both know Robot in order to determine whether they have collided with them.

Methods from BoardDisplay is applied in almost all of our classes. Especially the "Set" method is used quite frequently, so we thought that all of our classes would be unidirectionally associated to Board-Display, since none of our classes actually inherits from BoardDisplay and that BoardDisplay does not know any of our classes. Furthermore, Board knows Robot and BoardElement and uses Robot to add it to a list, and BoardElement as x in the List.Filter and List.Map-functions. Lastly, we have our Game-class that knows BoardFrame, Board and BoardDisplay in order to keep track of board and the Game process.

A UML diagram with all of our classes, as well as their methods and attributes are shown in Figure 1.

2.2 Representing the board

Initially we considered which data structures were fitting for both the representation of the board, but also the content within it. Our initial intuition was that a 2D Array, that can store both rows and columns, would be fitting for this task. When discussing how to implement the best internal representation of the board, we considered whether to create an array of strings, array of objects we have defined ourselves or

```
(* Board Display Snippet *)
    member this.Set (row: int, col: int, cont: string) =
        f.[row,col] <- [cont]
```

Figure 2: Snippet from BoardDisplay

```
(* 11g0 - BoardDisplay *)
type BoardDisplay (r: int, c:int) =
    let rows = 2*r+1
    let cols = 2*c+1
    let size = (rows, cols)
    let f = Array2D.create (fst size) (snd size) ("␣␣")
    member this.Set (row: int, col: int, cont: string) =
        f.[row, col] <- cont
    member this.SetBottomWall (row: int, col:int) =
        f.[row,col] <- "--"
    member this.SetRightWall (row:int, col:int) =
        f.[row,col] <- "|␣"
    member this.Show () =
        for i in 0 .. rows-1 do
            printfn ""
            for j in 0 .. cols-1 do
                if (i%2 = 0) && (j%2 = 0) then
                        this.Set (i,j, "+␣")
                printf "%s" <| f.[i,j]
```

Figure 3: The full function BoardDisplay, which displays the board with its right and bottom walls.

coordinates for relevant objects as well as making a field class. At first, we discussed that we could create a unique field class, but discarded the idea, as it felt like it would be too much unnecessary work and we felt like we were moving in the wrong direction. We then agreed to settle for representing the board as an array of strings, where we would choose to represent the walls as strings. Thus having a 2DArray of strings. We conversed that the solution of implementing arrays results in constant look-up time, which is advantageous, but may as a consequence use up a lot of space. Aware of this, however, we still thought this representation seemed like a good solution.

3 Program Description

3.1 11g0

3.1.1 Displaying the board

We made the BoardDisplay class with an Array2D. We created our first member "Set", which takes a row, a column and some content and sets the content on the given row and column. The code was simple to implement, since what we have just described in natural language is very straight-forward in code. As seen in figure 2

The coding of the rest of the members, SetBottomWall and SetRightWall, followed a similar approach. All we had to write out was that the given row and column had to be updated with a string-representation of the walls. The last member this.Show(), had to print the board, which we implemented with a printfn-statement of the BoardDisplay class, such that it will be printed to the screen. See Figure 3 for the final solution.

```
(* Robot Snippet 1 *)
override this.Interact other dir =
        match dir with
        | North when (other.Position = (((fst pos)+2), snd pos)) -> Stop (other.Position
        | South when (other.Position = (((fst pos)-2), snd pos)) -> Stop (other.Position
        | East when (other.Position = (fst pos, ((snd pos)-2))) -> Stop (other.Position)
        | West when (other.Position = (fst pos, ((snd pos)+2))) -> Stop (other.Position)
```

Figure 4: Snippet of the "Robot"-class. The member interact is a method that makes sure to stop another robot that is attempted to move into the robot.

```
(* Robot Snippet 2 *)
member this.Step dir =
        match dir with
        | North -> this.Position <- ((fst pos-1), snd pos)
        | South -> this.Position <- ((fst pos+1), snd pos)
        | West -> this.Position <- (fst pos,(snd pos-1))
        | East -> this.Position <-(fst pos, (snd pos+1))
```

Figure 5: Snippet of the "Robot"-class. The member "Step" is a method that moves the robot a field in the direction dir. It is an auxiliary method to change the Position, as Step does not have to take into account other game elements..

3.2 11g1

3.2.1 Modifying the Robot class

We were given some code for a class called Robot, but had to expand the class with more features. Firstly, we created a variable that took care of the conversion between the inner and outer representation of the array. From here, we had to implement two properties, Position and Name, indicating the name of the robot that would be used when rendering it on the BoardDisplay, and the position of the robot on the board. We also had to create two methods. The first is called "Step", which should, in a nutshell, move the robot in a given direction. And the second, "Interact", which was used to stop another robot trying to move into the robot.

Initially, when coding the the Robot-class, we wrote that this.Position had to be equal to a (row,col) pair, the initial input, which the robot was given. We would then try to make a match-case where we would, e.g. with the direction North, refer to these rows and columns. This ended up being problematic, as the program would not compile, etc. We then tried to reference to "Robot" in front of these rows and columns, but we quickly realized this would only create a new robot, and we would still have our initial robot, thus causing us to have two robots (of the same name) in two different positions. We deleted and started over. We realized we needed some mutable variables. We then made a member this.Position, with a get'er and a set'er. Get refers to the position it initially gets, and the set'er receives an input a and mutates it from a position to a. We would then make a match-case on North, South, East, West on how the rows and columns would be subtracted respectively, in order to move in the right direction. We then had a successfully functional "Step"-method. Furthermore, naturally, the robot should have a name. This was easily implemented by creating a member called name, which takes the name given to the robot when it is instantiated as an object. Lastly, we had to make the method "Interact", which we implemented with a match-case. If, for example, the other robot was moving north, it would have to step in the row below (in the outer representation of the array) the position of the robot that called this.Interact. As such, stop had to be called in other.Position. By applying this same line of reasoning, we implemented a case for south, east and west as well. In all other cases, this.Interact should return Ignore, allowing the robot to move on. Thus, we used a wildcard for this last pattern. See figure 4 for a snippet of interact and figure 5 for a snippet of Step.

```
(* Snippet of the Goal class *)
override this.RenderOn display =
            display.Set (row, col, "gg")
override this.GameOver (r: Robot list) = not (List.forall (fun (x: Robot) ->
                                                 x.Position <> this.Position) r)
```

Figure 6: Snippet of the "Goal"-class with focus on the "GameOver"-member that checks if a robot is in the goalfield.

3.2.2 Creating a Goal class

We created the Goal-class, where we inherited BoardElement and defined a member This.Position, so we could refer to a row and a column. Then we created a variable that took care of the conversion between the inner and outer representation of the array. We would render on the display with the string "GG" representing the goal, with the position that the goal should have. Apart from just representing the goal, we also needed to implement the GameOver-method, which was to return true, if the robot was to stop on the Goal. We created GameOver as a member, which takes a list of Roots as an input. We figured that we needed to somehow iterate over or somehow check the list for robots who would be in the same position as the goal. If this condition is true, the game would be over. To solve this in practice, we initially attempted to use the higher order function List.filter in order to apply a function to elements in the list, but figured out that List.forall was a better solution, since it allowed us to check all the elements for our condition and return true if they met the condition, instead of returning a new list, which List.filter would do, where we would have to check this new list for our condition. We got a more concise code using the List.forall function, that directly returns true if the game is over, essentially by checking if any of the robots in the list of robots are in the same position as the goal. If this condition is true, then the game is over. If it is false, then the game continues. See 6 for some of this code.

3.2.3 Creating a BoardFrame Class

We created the BoardFrame class, which should represent the outer walls of the game board. The class should take two integers as arguments, r and c, that specifies the size of the BoardFrame.

As per design, we assume that BoardDisplay and BoardFrame must always have the same input, because we consider "row" and "col", which BoardFrame receives as input as the same "r" and "c" in BoardDisplay. Aware that it is not the same thing internally in the computer, it is what makes the most sense in terms of design that the four variables in terms of the two pairs must always be the same. From here, we created a variable that took care of the conversion between the inner and outer representation of the array

Due to a misunderstanding, we initially had some of the "BoardFrame"-code in our "Show"-function. However, we changed the design along the way, by taking some of our code from "Show" and implementing it in "BoardFrame". Thus solely using "Show" as to print the board, and thereby thinking of BoardFrame only as an internal representation and *not* the GUI that should be printed in the console.

3.2.4 Creating the VerticalWall Class

VerticalWall is used to represent an inner vertical wall. It is given the starting fields for the wall and the length of the wall. If the length is positive, the wall runs from north to south, otherwise, it runs from south to north, and the wall will always be on the east side of the starting field. Firstly, we created a variable that took care of the conversion between the inner and outer representation of the array

Our initial thought was to implement an "System.IndexOutOfRangeException"- exception. Should the wall not fit on the board, this exception would be handled. This would create only part of the wall (the part that could fit on the board), and print a message to the player informing him/her of this. Apart from this we would have two cases.

1) The length of the wall is positive. This would lead us to a couple of sub-cases.

1a) The wall is length 1. Here we would simply have to display.SetRightWall on the first and sec-

```
(* Vertical Wall snippet *)
"display.SetRightWall((fst_pos)+(i+1),(snd_pos))".
```

Figure 7: The Vertical Wall going from North to South

```
(* Vertical Wall snippet 2 *)
"display.SetRightWall((fst_pos)+(i-1),(snd_pos))".
```

Figure 8: The vertical wall going from South to North.

ond position.

1b) The length of the wall is bigger than 1. Then we would have to go through $1..2..(2 \cdot (n-1))$ and then set the wall as in Example 7. The "$i+1$" addition means that wall goes from north to south.
Now we can look at the second case:

2) The length of the wall is negative.

Here we followed a similar approach. In fact the code is almost entirely identical, except that we checked for the case of the length being -1 instead of 1 and that when in the case of the length being less than -1, we still used the summation formula, except changing the positive values into negative ones, as in: $-1.. -2..(2 \cdot (n-1))$. We also had to change the The "$i+1$" addition into "$i-1$, which would make the wall go from south to north. Apart from this, everything else is the same as seen in the prior example. Example 8.

3.2.5 Creating the HorizontalWall Class

The HorizontalWall class should represent an inner horizontal wall. Like VerticalWall, it is given the starting fields for the wall and the length of the wall. If the length is positive, the wall runs from west to east, otherwise it runs from east to west. The wall is on the south side of the starting field. Initially, we created a variable that took care of the conversion between the inner and outer representation of the array.

Creating the HorizontalWall class was very similar to VerticalWall. It is almost completely identical, except for the fact that we used BottomWalls instead of RightWalls, and that to make it go from west to east, we had to add "i" to the second position coordinate. And to make the wall go from east to west we had to add "$i-2$" to the second position coordinate. We would still handle the same exception as before, that the wall did not fit on the board, use the same summation formulas etc.

3.3 11g2 - Interaction:

3.3.1 Implementing the "Board"-class

The purpose of the Board class is threefold. Firstly, it should allow us to add new instances of robots and board elements, while adding these objects to their corresponding list. Secondly, the board should all the objects that exists in board to render themselves on a board display. Lastly, it should allow the robot to move around the board.

In order to implement the "Board"-class, we use the Method AddElement to set up a game board. The Property "Elements" is used to get a list of all game elements (robots inclusive), and another property "Robots" is used to get a list of all robots. A board will always have a BoardFrame element, which is quite obvious, since there can't be a game with the board frame. We apply a method "Move" in order to move a robot. Before each step the robot takes, all game elements must be checked except for the robot itself, and the method "Interact" must be called for each element. We implemented this by higher-order functions List.filter and List.map. It seemed quite intuitive with these two higher order functions to first we make a temporary list of all the board elements that are not the robot itself by applying the higher order functions List.filter, and then applying List.map to call Interact for each element in our temporary list. We get a new list out of this that we call our "action list". Lastly, we make a recursive helper

```
(* Board snippet *)
member this.Move (r: Robot) (dir: Direction) =
        let tempList = List.filter (fun (x:BoardElement) -> x <> (r:>BoardElement)) this
        let actionList = List.map (fun (x:BoardElement) -> x.Interact r dir) tempList
        let rec moveHelper (actions: Action List) =
            match actions with
            | Ignore:: xs -> moveHelper xs
            | (Continue (condir, pos))::_ ->
                r.Position <- pos
                this.Move r condir
            | (Stop pos)::_-> ()
            | [] ->
                r.Step dir
                this.Move r dir
        moveHelper actionList
```

Figure 9: The Move Method that uses lists to filter elements out and map all elements to interact, ultimately to make the robot move. Worth to know to understand the code better: Before Move is defined in Board, Board receives a list of elements and a list of drones. It also has AddRobot and AddElement methods, as well as a This.Robots member.

function that takes as input an action list. We pattern match on this input, and if all elements in the list return Ignore, we make a call to this.Step and recursively call this.Move (thus allowing us to check if we can move from the new position). If an element returns Continue, we update the position of the robot being moved to the position in which Continue was output, and recursively call this.Move on the same robot with the direction given by Continue. If even one element returns Stop, we simply return (), thus causing the robot to stop moving. See some of the most important code for the "Board"-class, specifically with focus on the "Move"-method in figure 9.

3.3.2 Implementing the "Game"-class

To make it possible to play the game, we created a member "Play" in the Game class. We implemented a counter that counts the number of moves. We used some print statements to print out information to the user, such as "Press enter to choose a new robot or use the arrow keys to control current robot." We made a recursive playHelper function that takes a board as an input and checks whether "GameOver" appears in the list of elements, in this case we would print out a statement to the user, implying that the game is over and display the number of moves used to complete it. If not, the game is not over yet and we continue to our else-case. In here, we make a variable ,key, which reads the key pressed by the user. We then made a pattern match that matches on this key. Depending on the input by the user, playHelper would behave accordingly. For example, if the user presses the UpArrow-key, playHelper would, among other things, make a call to move in the North direction and then recursively call itself again. If the user pressed any other button than the arrow-keys or ENTER, the program displays the message: "Invalid input. Press an arrow key to move or press enter to choose a new robot." on the screen, and recursively calls itself again to keep the player in the game-loop.

3.3.3 Implementing extensions

When we first started discussing extensions, we had actually intended to implement the extension "dynamite". The idea was that whenever a robot would collide with dynamite, the robot would blow up and die/be eliminated from the game. However, due to problems with our design, we could not implement this. Instead, we thought that adding the concept of "Mirror" as an extension would be a fun concept, such that if the Robot moves into a Mirror field, it would start to move in a different direction. As an example, if a robot going north moved into a mirror, it would start moving east instead. Secondly, we added a "Trampoline", which would allow the robot to jump over the closest spot, giving it the chance to avoid obstacles that would otherwise be in its way. Together, both of these elements added a new and fun twist to the game, and allowed for even more creative ways for a player to get to the goal in the least amount of moves. See the code of Mirror in Figure 10. The code for Trampoline is a bit too comprehensive to include here, so we refer to the .fs file instead.

```
(* The full code of the Mirror-class *)
and Mirror (row: int, col: int) =
    inherit BoardElement ()
    let pos = (((2*row)-1), (2*col)-1)
    member this.Position = pos
    override this.RenderOn display =
        display.Set (fst pos, snd pos, "//")
    member this.Impact (r: Robot) = this.Position = r.Position
    override this.Interact other dir =
        if this.Impact other then
            match dir with
            | North -> Continue (East, (fst pos, ((snd pos)+1)))
            | South -> Continue (West, (fst pos, ((snd pos)-1)))
            | East -> Continue (North, (((fst pos)-1), snd pos))
            | West -> Continue (South, (((fst pos)+1), snd pos))
        else
            Ignore
```

Figure 10: Snippet of the "Mirror"-class. The class inherits from BoardElement and defines positions privately. It utilizes the RenderOn method to render on "//" the Display, which represents a mirror. It also has a member Impact to check if its position is equal to that of the robot and a interact method, which checks for impact and matches directions in North, South, East and West with Continue and which positions to manipulate.

3.3.4 Problems with implementation

As with most, if not all, other implementations of a program, we ran into a few problems with ours, the first of which caused us the most trouble. In our board, when looking at the inner representation, every even row and column is essentially filler space. What we mean by this is that the robot will never touch these spaces, and instead they serve as space for the inner walls to be placed. As such, the inner and outer representation of the array is different. Hence, if a player puts a robot on position (10,2), said robot will, in the outer representation, look to be placed on exactly that spot. In the inner representation, however, the robot is placed at position (19, 3).

What this meant for us was that we had to convert between the inner and outer representation, such that it wouldn't cause trouble for the player. Moreover, it was done in order for the game to function properly. Had we not done the conversion, a robot placed in position (2,2) would be placed in an even row and column, causing the robot to not only be placed in an "illegal" space, but also move in said space. Obviously, this would be detrimental to both the user experience and the game play. Our somewhat naive approach to this was that we would define a variable in each class taking care of it. As an example, in our robot class, we would create a variable, called pos, that did the following: $(((2 \cdot row) - 1), (2 \cdot col) - 1)$. If we were to instantiate a robot with position (2,2), this variable would convert it to position (3,3) which does in fact correspond to position (2,2) in the outer representation. The problem with doing this on a class-by-class basis, however, is that every time we had to write a new piece of code that, in some way, dealt with the position of an object (and this was more or less all the time), we would have to keep in mind, and abide by, each variable from every individual class. Furthermore, by adding these variables to every class, we gave the classes a responsibility they never should have had in the first place. A more clever solution would have been to let BoardDisplay take care of the conversion one time, and then letting each class use these conversion, thus avoiding the trouble of having to keep track of many individual variables. Safe to say this caused us a lot of problems and headache, and if we were to do the same assignment over again, this particular issue would be the first thing we would fix.

Our second problem came when implementing the Trampoline-extension. This opened up the possibility of two robots occupying the same space. By default, robots can not occupy the same space, and they will never move directly into one another. However, if a robot were to interact with a trampoline, this robot would jump over the closest spot and go straight to the next closest. This is obviously not possible in a physical implementation of the game, and so we had to decide how this edge case would be treated. Our decision was that, if an event like the above were to occur, the original occupant of the spot would be thrown away for a second, giving the robot who used the trampoline a chance to make its next move, only for the original occupant to then return back to its position afterwards.

3.3.5 How to use the program

The program is run from the Mono Command Prompt. When in the Command Prompt, the user needs to type "fsharpc robots.fs robots-game.fsx" followed by "mono robots-game.exe" to run the application.

Now the game can start and the program will ask the user to click "Enter" to choose a robot or to use the arrow keys to move the default chosen robot. If the user wants to chose a robot, he has to choose between robot 1, 2 or 3 and can select one of them by pressing down the "1", "2" or "3"-key. When a robot has been chosen it can be moved around the board. The player's objective is then to move one of the robots into the goal, so that the player can win the game. The game will tell you have many moves you used in order to complete the game.

Left arrow key = move left. Right arrow key = move right. Up arrow key = move up. Down arrow key = move down.

4 Testing and experiments

The following is a list of the testing/experiment we did for our implementation.

1) We did a test with differently sized boards,. eg. 5x5, 10x10 and 15x15 (before conversions), to check if our board would be printed correctly at different sizes. No problem founds in this test.

2) We did a test with both a horizontal and vertical wall with its length set to 100 such that it wouldn't fit on the board. This correctly throws an exception, and display a message to the user explaining that only some of the wall has been drawn. This was tested to make sure we wouldn't encounter a runtime error and the whole program would crash.

3) We tested whether a robot sliding through the goal would trigger the if-statement in this.Play, causing the game to be complete and the program to terminate. As expected, this does not happen. Conversely, if the robot does in fact stop on the goal, the program correctly terminates. Both of these were tested to make sure the game works as intended.

4) We tested whether the mirror-extension worked as intended. If, for example, the mirror is located in row 1 and the robot is coming towards it from the east, the mirror obviously cannot change the robots direction to north, as that would cause the robot to go outside the board, thus entering an infinite loop and bugging the game. As such, in these rare, edge cases, the direction assigned to the robot will be "flipped", i.e. in the aforementioned case, the robots direction would change to south. In all other cases apart from these, the mirror functions regularly. Be aware, though, that the solution to the above program only works as long as the board is 15x15. However, since our game is played only on a 15x15 board, the solution suffices. As such, the mirror-extension is working exactly as expected.

5) We tested whether the trampoline-extension worked as intended. This works as intended. In the case that, for example, the trampoline is positioned at row 1, and the robot is coming towards it from the north, the robot will stop correctly stop in front of the robot. This is to prevent the robot from jumping outside the board, thus making the game go into an infinite loop. In all other instances, except for these few edge cases, the trampoline causes the robot to jump. Be aware, though, that the solution to the above program only works as long as the board is 15x15. However, since our game is played only on a 15x15 board, the solution suffices. It will also correctly jump over a wall. Hence, our trampoline-extension is working exactly as expected.

6) When running the program through OnlineTA, we got a lot of errors. For example, when OnlineTA tested .Interact on a HorizontalWall(2,1,10) and a Robot(3,1), it expected a Stop(2,1), but our program gave a Stop(3,1). This, however, is not due to a fault in our implementation, but rather the fact that our representation of the board (explained thoroughly in section 3.3.4) is different to the way OnlineTA

represents it. Hence, our program works as intended.

As of this writing, there are no known bugs in the program.

5 Conclusion

The purpose of the assignment, to create a functioning game with extensions, was overall fulfilled. However, certainly not without a couple of challenges and changes underway. First we discussed how to represent the elements of the game. We ran into some problems in our implementation of the BoardDisplay-class, problems which ultimately happened to be irreversible. In our discussion of the representation of the game, we should have had thought more thoroughly about the role of the BoardDisplay class, and how it relates to other classes, since we had significant problems with writing code relating to position in our classes. We lost the good overview of things, since we had to keep track of too many individual variables. We realized we should have used BoardDisplay take of the conversion a single time, so each class could use these conversion, since each class inherits from it. If we had this in mind in our implementation, we would have saved ourselves from a lot of trouble. Next, we made UML-diagram of our program, which worked well to actually get a better overview of our program and see how everything relates to each other. We ended up creating a Board that was displayed successfully. However, we did not succeed before we figured out BoardDisplay's real purpose of printing the Board to the console. Initially we confused the internal and external representation and had a lot of code we had to move around to get it to work. In a nutshell, our mistake was not to read the assignment description properly, and we fixed the problem after we had spend a sufficient time of reflecting on the problem and adjusting our code.

We created a successful Robot class that had the intended Position and Name as well as the two methods "Step" and "Interact", which were used to stop another robot trying to move into it. We had some trouble underway, since we didn't include mutable variables in the beginning. However, after a bit of reflection on the issue, we figured out how to make the positions modifiable and got our methods working as intended.

Our Goal, BoardFrame and Wall classes were implemented without too much fuss. Something worth mentioning however was that our BoardFrame and BoardDisplay classes got a bit mixed together at first, but we fixed that after some adjustment at mentioned above. All in all, we had our basic BoardElement classes working successfully, with some trouble.

After creating the class hierarchy of Game Elements, our third sub-task was to put the different parts into a functioning game. To achieve this, we had to implement Board-class, that could set up the game board, keep track of all elements and robots on the board and can move a robot, and to create a Game-class that would let the player, play the game. This was achieved without a lot of fuss and we a game, where the player was able to interact as intended. With our last sub-task, where we had implement a few extensions, we ran into some problems. We initially made our trampoline extension, which worked well from the beginning. However, we then tried to create a second extension which was "Dynamite" that would blow up the robot when the robot collides with it. The implementation of this feature happened to be too difficult in terms of the design we had created. It was not easy to add "Dynamite" as a new feature without having to change a whole lot of things, and so we implemented the extension "Mirror" to replace it, which would just toss the robot around to a different position whenever it collides with it.

All in all, apart from a couple of beauty spots, we have a functioning game with fun extensions that works well. The final product is not entirely as we first imagined it. However, we are of the opinion that we have created a well-functioning game with solid extensions that satisfies the requirements of the assignment and is fun to play.