PoP - 11g

Hold 14 - Gruppe 04
Markus L. Bille
Ruimin Huang
Anders Elberling

11. januar 2022

Indhold

1 Introduction

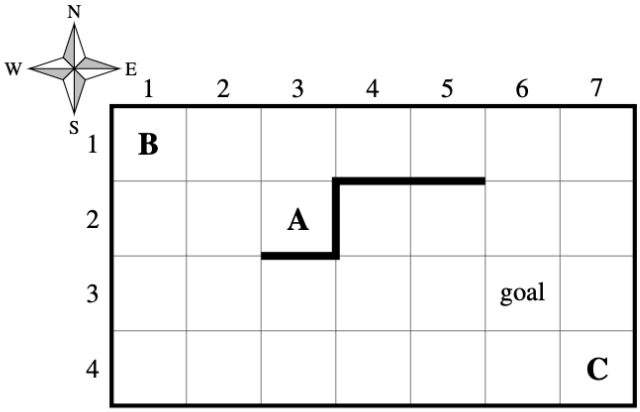
In this assignment we were assigned to recreate a variation of Ricochet Robots. We used OOP, by writing different game elements classes, such as Robot,BoardFrame and different walls, which inherit an abstract class called BoardElement. We found out that OOP is perfect for coding a game with many elements.

The final product works well, although there is many more we can do if we get more time. Anyway people can play this game smoothly,getting a lot fun, as long as they don't press wrong keys-We could not find time to handle I/O exceptions.The program meets all the demands in the assignment, the robots can slide on the board until it is blocked by a wall or an other robot.Every move is showed in the console.The game will stop when a robot arrives at the goal.We also changed the color of console.We want the game interface looks sample,so we put only one emoji in the game.It is a Yinyang fish represents a portal,which is also an extra figure.

We also thought about put on some sounds, make it like a old fashion computer game. But after some research we find it is too complicate and need more time.

2 How the game works

The game exists on a board with r x a spaces, where a number of robots can be moved around. Every robot starts in a separate space on the board and is moved by sliding in a direction and will keep gliding until it hits a wall, or other robot. The goal of the game is to get one of the robots to reach the goal in the fewest amount of moves.



To complete the game, the robot need to completely stop at the goal, gliding across the goal wont count. We also introduce a teleport portal in the game. Any robot passes it will be sent out the board. This makes the game a little more difficult.

3 Problem and design

As mentioned above,the game involved a lots of different elements, how to handle the relation of elements, how to make they interact each other is the core problem in this assignment.We write classes of the elements as templates to make different element objects:robots,frame,innerwalls, goals and portal.Robots will move and try to stop at the goal, and the other elements is used to change the direction of the robot.

The centre function of Robot is Step(). By calling this function,the robot can move a step, thus starts a round of game move. Below is the function.

```
member robot.Step dir =
     match dir with
           |North-> position <- ((fst position)-1,snd position)
           |South-> position <- ((fst position)+1,snd position)
           |East -> position <- (fst position,(snd position)+1)
           |West -> position <- (fst position,(snd position)-1)
```

Then in Board class there is Move function, which will call Step function to make a game move, like this:

```
member this.Move (robot:Robot) (dir:Direction) =
   let rec move (r:Robot) (d:Direction) lst =
      if List.forall (fun (x:BoardElement) ->x.Interact r d = Ignore) lst then
           r.Step d
           move r d lst
      else
          let e = List.find (fun (x:BoardElement)-> x.Interact r d <> Ignore) lst
          match (e.Interact r d) with
                  |Stop pos -> r.Position <- pos
                  |Continue (dirt,pos) -> r.Position <-pos

   move robot dir elements
```

Here we uses a helper function "rec move". if there will no board element in the robot's new position and stop it, the robot will continue to move step by step in the same direction until it is stopped. Sooner or later the robots will be stopped by something, there is a outer wall in every direction after all. So the robot moves one step, if it doesn't knock on something, it will make one more step, and call the recursive function. If it is step on something (we find it by go though the board elements list and let every element interact with the robot), then return the according kind of Action, like Stop or Continue.

At last in the Game class, there is a function Play, which will setup the game,then in a while loop it will call the Move function, function to make the game move, and show the result by calling BoardDisplay.Show(). As long as no robot arrive the goal, the game will going on and on. (In fact we should find a way to allow players to break the game, jump out the loop. But that need more time, because we could find something in f like keyword "break"or "stop"to break the loop easily).

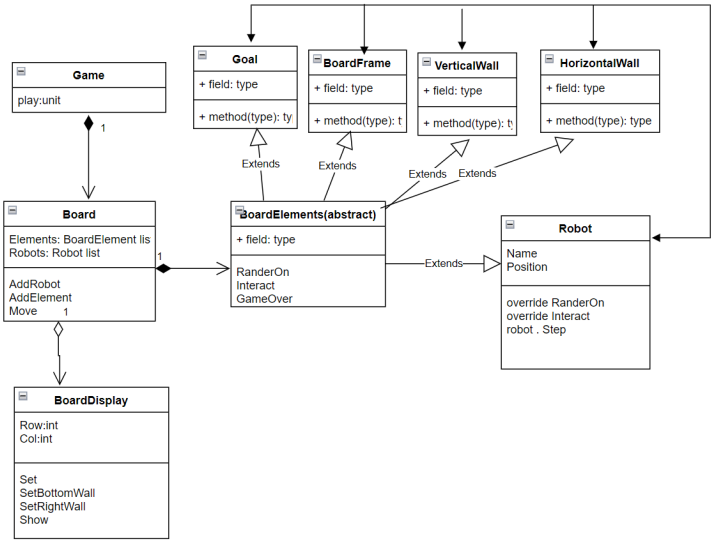
```
 while not (goal.GameOver board.Robots) do
     printfn "%s" "Choose the robot you want move:"
     let n = (System.Console.ReadLine())
     let r = List.find (fun (x:Robot) -> x.Name = n) board.Robots
     printfn "%s" "Press Arrow keys to move the robot:"
     let k = System.Console.ReadKey(true)
     let mutable dir = East
     match k.Key with
        |System.ConsoleKey.UpArrow -> dir <- North
        |System.ConsoleKey.DownArrow -> dir <- South
        |System.ConsoleKey.LeftArrow -> dir <- West
        |System.ConsoleKey.RightArrow -> dir <- East
        |_-> printfn "%A" k.Key
     board.Move r dir
     counter <- counter + 1
     System.Console.Clear()
     let bd = new BoardDisplay (rows,cols)
     List.iter (fun (x:BoardElement) -> x.RenderOn bd) board.Elements
     bd.Show()
 printfn "You reach the goal! It takes you %d steps" counter
```

This is the basic design of the program. We want to start by handling as less element as possible. So we didn't totally follow the steps described in the assignment.At the beginning, we only write one element: robot. Use robot we can test many game functions.By this way we make a simplified version game. In the process we learned how to make the elements interact correctly. Later on we added elements one by one, write more complicated Interact function, for example the VerticalWall.Here is its Interaction function:

```
override this.Interact robot dir:Action =
  if (snd robot.Position = c) && (fst robot.Position > (r-1)) && (fst robot.Position
  < (r+n)) then
        match dir with
          |East -> Stop (fst robot.Position,c)
          |West -> Stop (fst robot.Position,(c+1))
          |_ -> Ignore
  else
       Ignore
```

4 Program and classes

It can be tough to know how every class relates and extends each other, the UML flowchart below describes the frame of the program. First we have a Game class, which is composed of Board, the Board is in turn composed of BoardElement. There are five different elements: robot,goal,boardFrame,vertical wall, horizontal wall. All of them inherit from the abstract class BoardElement. There can override some functions from the base, like RenderOn and Interact, and more importantly, all the elements can upcast to BoardElement, so that we can put them in one list, which is essential for the game to run.From the diagram we can see the inherit relationship by the extends arrows.



Furthermore, all the five elements are associated with each other.For example the robots will interact with all the other elements.The board don't know all this elements, on the contrary,the elements keep all proprieties such as position in their self. An other class only connect with board, BoardDisplay is aggregated by the board.After every move, it will generate a new display to show the board. BoardDisplay functs like a black box, objects of the elements classes only use RenderOn function to let the BoardDisplay to know where there are.
Here we would introduce some important classes in detail.

4.1 BoardElement

Beside the two important abstract functions, RenderOn and Interact,the abstract class for elements on the board also include some very handy default values, like Interact's default value is Ignore. By this way, when we write the program, we only need to think about the elements that do affect the robots' movement. The other elements will Ignore the robot, and we can Ignore them in turn. Also the gameover function has default value false, so in the game loop,as long as the value hasn't been set true, the game will play on.

4.2 Robot

It is worth to mention,that the Robot class and BoardElement class are mutually recursive,because they use each other. Therefor we use the keyword "and"rather than "type"in front of Robot constructor. When a robot interacts with an other element,the result (Action) could change the robot's position, so it is important to write a setter for the property Position. The code below shows how BoardElement and Robot connect each other:

```
    [< AbstractClass >]
type BoardElement () =
  abstract member RenderOn : BoardDisplay -> unit
  abstract member Interact : Robot -> Direction -> Action
```

```
  default __.Interact _ _ = Ignore
  abstract member GameOver : Robot list -> bool
  default __.GameOver _ = false
and Robot ( row : int , col : int , name : string ) =
  inherit BoardElement ()
```

4.3 Goal

The goal ends the game if a robot ends their turn on it. So among all the elements only Goal overrides function GameOver.

```
override this.GameOver (robots:Robot list) =
      let mutable gameover = false
      for elm in robots do
          if (fst elm.Position) = r && (snd elm.Position) = c then
             gameover <- true
          else ()
      gameover
```

The value of this function becomes the condition of the while-loop in the game.Play(). Before every game move,it goes though the robots list and compares their position with itself one by one, if one robot has the same position with the goal, the game is over.

4.4 BoardDisplay

It is responsible for printing out the board.Here we use a hiding propriety "fields", which is a 2D array, from its index we can find every field on the board.For the value of the fields, we use a tupler (bool*bool*string) to give information of elements on every field, the first bool value is about vertical wall, the second is about horinzontal wall, and the third is a string, which can be a robot's name or "gg"to mark the goal. Every element will uses RenderOn function to call BoardDisplay's Set,SetBottomWall or setRightWall to change the value of the fields.By this way, the board don't need to know the elements, but the elements talk control of their position. After the game setup and every game move, the program will go through the elements list and make every elements render their self on the board. According this, boardDisplay class will create a big string (by overriding ToString function) and by calling Show() function, print it in the console, so that player can see what happens on the board.

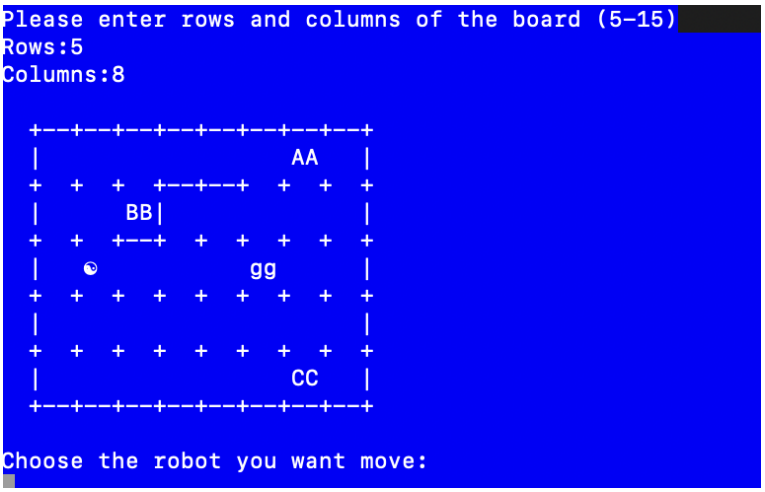
Here is the some pieces of code:

```
  let fields:Status [,] = Array2D.create (rows+1) (cols+1) (false,false,None)
  member this.Set (row:int) (col:int) (str:string) =
       let (a,b,c) = fields.[row,col]
       fields.[row,col] <-(a,b,Some str)
  member this.Show() =
       printfn "%s" (this.ToString())
```

It is worth to mention that we didn't write an override RenderOn function for the board frame, because it is always on the board. In stead we write an private member in BoardDisplay class:

```
     member private this.SetFrame() =
       for j = 1 to cols do
           this.SetBottomWall 0 j
           this.SetBottomWall rows j
       for i = 1 to rows do
           this.SetRightWall i 0
           this.SetRightWall i cols
```

The final display is simple but effective like below:



4.5 Game

It is the Game class assemble all the elements and function together. There is only one function in Game: Play(). This function will setup the board firstly and then start a while loop in which player can make moves until the game is over.

Setting up the game is to instantiate classes and to make different of objects, including 3 robots, walls, board frame,the goal and the portal.It is difficult for us to decide how to arrange these elements on board.Because player will decide the size of the board, we could not just set all the elements in the fixed field. But we don't want its totally random, because then we need to write more code to make sure two robots will not be set in the same position of one robot on the goal at start. So we make its partly random by give robot row number but random column number.

In the while loop we make player input instruction to move which robot on which direction. We add an hiding field called "counter"to count how many moves it talk and will print the number when the game is over.

5 Possible expansions

We have made some extra feature, like a teleport portal. We changed the console's color to blue not just because of laziness, it is also because this color remind people the ole days computer screen suddenly go all blue.We put a Yinyang fish in the game, and we didn't forget to set a counter.The final product is quite satisfying, but because time limit, there are still some place to be improved. For example, we should handle exception when asking player to input instructions. We should allow player quit the game half way. The portal still have bugs, if it send an other robot out the game, it will be the same position as the first one. And We really want to make some sound for every game move, because it happens too fast, player sometimes will fail to notice the move has done.The same will happens if player choose a robot move to a direction, in which there are something stop the robot to move.Then after the instruction nothing will change, the system just ask the player choose another robot to move.If there is a sound, player will know the robot tried to move.

More expansions could be:

- tiles that push the robots once
- buttons and doors
- movable walls via switches or buttons
- coins to be collected before the game can be completed
- one-way walls
- a level creator: unsolved questions: why match pattern don't work well?

6 Conclusion

This project have quite high degree of completion, and the program's performance is really satisfying, without many bugs.It is interesting to develop the game,using all kinds of knowledge we learn in this course. From working for this assignment, we tried OOP and get much better understanding on class and relationship between classes.