PoP 12i
An exploration of the electronic implementations of a Chaturanga-derived board game

pwn274

January 2022

Introduction

Ever since the dawn of man, games has been an integral part of the human condition, (*Homo Ludens*, Johan Huizinga (1938)) and except for a brief outlawing in most of Europe in 1254 (*Europe: mémoires emblèmes*, Michel Pastoureau (1990)) chess has always been at the heart of western game culture (*A history of chess*, Murray, H. J. R (1986)).
Chess took over from Chaturanga and has therefore existed since the at least ancient India (*Chess and playing cards*, Stewart Culin (1898)), so it can be seen as somewhat recently that computers has been powerful enough to beat humans. Even though it is very interesting how the mere act of Kasparov losing to a computer took the world by storm (*Deep Blue: An Artificial Intelligence Milestone*, Monty Newborn (2002)), this will not not be the focus of our thesis. In the following sections, we will use code-examples to try to both understand and somewhat recreate the roots of the human-computer interface and how it relates to chess.

12i0

In this first sub-question I was asked to create a class Player and a derived class Human. I will not be getting into the dire philosophical consequences behind having the human part be a derivation of a generic 'player', thereby placing us conscious animals on the same footing as a chess engine. What I will be getting into on the other hand is the implementation, which was quite trivial.

Creating the generic player class required only the following three lines:

```
[<AbstractClass>]
type Player () =
  abstract member nextMove : board -> string
```

The human was then created using inheritance in the following way:

```
type Human () =
  inherit Player ()
  override this.nextMove (Board : board) : string = //function body
```

1 12i1

The run method could have been implemented in a multitude of ways. Having grown quite fond of *F* and functional programming, I chose to create the recursive function `taketurn` which asks for a `Player`'s move and updates the board accordingly.

A summary of the function is shown below where some details have been removed or changed for clarity:

```
let rec taketurn (players : Player list) (gameboard: board) (ind : int) : int =
    let nextmove = players.[ind%2].nextMove gameboard
    match nextmove with
```

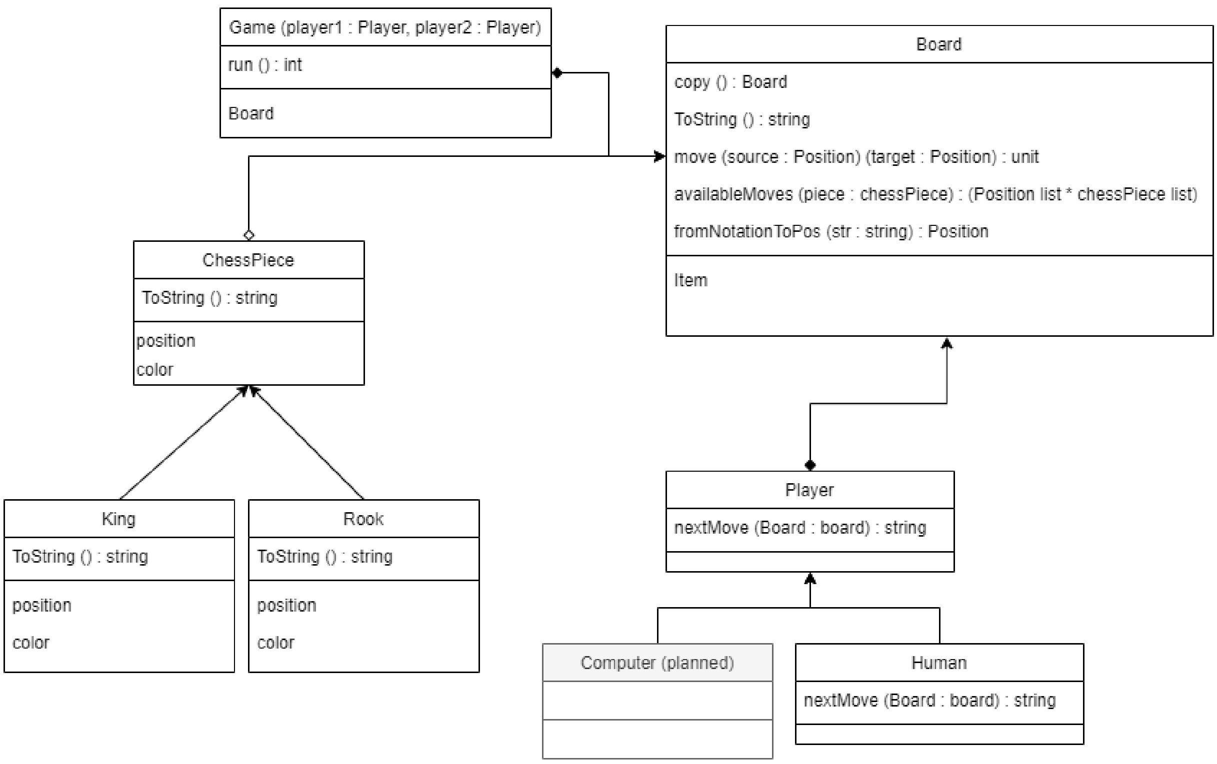
```
        | "quit" -> ind
        | "not a legal move" | "error" | "no piece selected" | "wrong formatting" | ""->
          printf "\%A: Unable to do move!\n" nextmove
          taketurn players gameboard ind
        | _ ->
          gameboard.move nextmove
          printf "\%A has been moved" nextmove
          taketurn players gameboard (ind+1)
```

`taketurn` was then started in the `run`-function in the following way:

```
member this.run () : int = taketurn [player1;player2] gameboard 0
```

12i2

Having read throught the Jon Sporring F-sharp-notes a couple of times, I am still unsure on the exact specifications of a good UML-diagram. My best attempt is the following:



Figur 1: Legend has it that Figur 1 is the finest UML-diagram known to man.

12i3

Whoever wrote the code handout made a royal[1] screw up, as they forgot the special rules for the movements of the king-pieces. It was up to me to correct the wrongs, which I did by using one of the most powerful moves in coding[2]: a double for-loop:

```
let mutable threatened = []
if piece.nameOfType = "king" then
    for i in 0..7 do
        for j in 0..7 do
          match this.[i,j] with
```

[1]pun intended
[2]I'm gonna do what's called a pro-grammer move

```
                    | Some (opp) when opp.color <> piece.color && opp.nameOfType <> "king" ->
                        threatened <- threatened @ fst (this.availableMoves opp)
                    | Some (opp) when opp.color <> piece.color && opp.nameOfType = "king" ->
                        threatened <- threatened @ (/*The possible king moves*/)

let possible = List.filter (fun v -> not (List.exists (fun o -> o = v) threatened)) vaca
```

As can be seen, a special case had to made for the opponents king, as my first implementation ended in an infinite loop of two kings trying to figure out each others moves without ever actually reacting.[3]

Conclusion

I did what you asked of me. Sorry for making you read this. Thank you for PoP :)

[3]something-something Russia and USA in the cold war