Introduktion til Programmering og Problemløsning (PoP)

Jon Sporring
Department of Computer Science
2021/01/03

UNIVERSITY OF COPENHAGEN

Skak

Opgave: chess.fs, pieces.fs, chessApp.fsx

Trin til at forstå koden:

1. Oversæt og kør. Virker den? Ser output ok ud?
2. Skab overblik over koden (bagfra).
3. Forstå detaljer i koden, stil kritiske spørgsmål, lav forsøg for at bekræfte din forståelse.

Oversæt og kør

15Chess — -zsh — 71×35

...sitories/PoP.git/lectures/15Chess — -zsh ~/repositories/fsharp-book — -zsh

```
sporring@Jons-mac 15Chess % fsharpc chess.fs pieces.fs chessApp.fsx
Microsoft (R) F# Compiler version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.
sporring@Jons-mac 15Chess % mono chessApp.exe
7
6
5
4   k
3
2
1   R
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 0)], [R])
R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (1, 2); (1, 3); (1, 4); (1, 5)
; (1, 6); (1, 7); (1, 0)],
 [k])
k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0)
; (3, 0)], [])
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)], [])
R: Some (3, 1) ([(2, 1); (1, 1); (0, 1); (3, 2); (3, 3); (3, 4); (3, 5)
; (3, 6); (3, 7); (3, 0)],
 [k])
k: Some (4, 1) ([(3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0); (3, 0)
], [R])
sporring@Jons-mac 15Chess %
```

Skab overblik over koden (bagfra).

chessApp.fsx

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

```
open Chess
open Pieces

/// Print various information about a piece
let printPiece (board : board) (p : chessPiece) : unit =
  printfn "%A: %A %A" p p.position (board.availableMoves p)

// Create a game
let board = board () // Create a board
// Pieces are kept in an array for easy testing
let pieces = [|
  king (White) :> chessPiece;
  rook (White) :> chessPiece;
  king (Black) :> chessPiece |]
// Place pieces on the board
board.[0,0] <- Some pieces.[0]
board.[1,1] <- Some pieces.[1]
board.[4,1] <- Some pieces.[2]
printfn "%A" board
Array.iter (printPiece board) pieces
// Make moves
board.move (1,1) (3,1) // Moves a piece from (1,1) to (3,1)
printfn "%A" board
Array.iter (printPiece board) pieces
```

-:--- **chessApp.fsx** All (17,30) Git:master (fsharp AC)

What we know from looking
- Chess and Pieces files (se
- board and chessPiece typ
- chessPieces has a positio
- chessPiece, position, and
- king and rook are a type
- board offers .[] – indexing

15Chess — -zsh — 71×35

...sitories/PoP.git/lectures/15Chess — -zsh | ~/repositories/fsharp-book — -zsh

```
sporring@Jons-mac 15Chess % fsharpc chess.fs pieces.fs chessApp.fsx
Microsoft (R) F# Compiler version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.
sporring@Jons-mac 15Chess % mono chessApp.exe
7
6
5
4   k
3
2
1   R
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 0)], [R])
R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (1, 2); (1, 3); (1, 4); (1, 5)
; (1, 6); (1, 7); (1, 0)],
 [k])
k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0)
; (3, 0)], [])
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)], [])
R: Some (3, 1) ([(2, 1); (1, 1); (0, 1); (3, 2); (3, 3); (3, 4); (3, 5)
; (3, 6); (3, 7); (3, 0)],
 [k])
k: Some (4, 1) ([(3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0); (3, 0)
], [R])
sporring@Jons-mac 15Chess %
```

Brug bibli

New Open Recent Revert Save

chessApp.fsx

```
open Chess
open Pieces

/// Print various information ab
let printPiece (board : board) (
  printfn "%A: %A %A" p p.positi

// Create a game
let board = board () // Create a
// Pieces are kept in an array f
let pieces = [|
  king (White) :> chessPiece;
  rook (White) :> chessPiece;
  king (Black) :> chessPiece |]
// Place pieces on the board
board.[0,0] <- Some pieces.[0]
board.[1,1] <- Some pieces.[1]
board.[4,1] <- Some pieces.[2]
printfn "%A" board
Array.iter (printPiece board) pi
// Make moves
board.move (1,1) (3,1) // Moves
printfn "%A" board
board.move (3,1) (4,1)
printfn "%A" board
Array.iter (printPiece board) pi
```

-:--- **chessApp.fsx** All (27,0) Git:master (fsharp AC)

15Chess — -zsh — 71×35

...sitories/PoP.git/lectures/15Chess — -zsh ~/repositories/fsharp-book — -zsh

```
sporring@Jons-mac 15Chess % fsharpc chess.
Microsoft (R) F# Compiler version 10.2.3 f
Copyright (c) Microsoft Corporation. All R
sporring@Jons-mac 15Chess % mono chessApp.
7
6
5
4   k
3
2
1   R
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 0)], [R])
R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (
; (1, 6); (1, 7); (1, 0)],
 [k])
k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (
; (3, 0)], [])
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)],
R: Some (3, 1) ([(2, 1); (1, 1); (0, 1);
; (3, 6); (3, 7); (3, 0)],
 [k])
k: Some (4, 1) ([(3, 2); (4, 2); (5, 2);
], [R])
sporring@Jons-mac 15Chess %
```

15Chess — -zsh — 102×36

~/repositories/PoP.git/lectures/15Chess — -zsh ~/repositories/fsharp-book — -zsh

```
sporring@Jons-mac 15Chess % mono chessApp.exe
7
6
5
4   k
3
2
1   R
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 0)], [R])
R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (1, 2); (1, 3); (1, 4); (1, 5); (1, 6); (1, 7); (1, 0)],
 [k])
k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0); (3, 0)], [])
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
7
6
5
4   R
3
2
1
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)], [])
R: Some (4, 1) ([(5, 1); (6, 1); (7, 1); (3, 1); (2, 1); (1, 1); (0, 1); (4, 2); (4, 3); (4, 4);
  (4, 5); (4, 6); (4, 7); (4, 0)], [])
k: <null> ([], [])
```

Skab overblik over koden (bagfra).

pieces.fs

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

```
module Pieces
open Chess

/// <summary> A king is a chessPiece which moves 1 square in any direction. </su…
mmary>
/// <param name = "col"> The color black or white </param>
/// <returns> A king object. </returns>
type king(col : Color) =
  inherit chessPiece(col)
  // king has runs of 1 in 8 directions: (N, NE, E, SE, S, SW, W, NW)
  override this.candiateRelativeMoves =
      [[(-1,0)];[(-1,1)];[(0,1)];[(1,1)];
      [(1,0)];[(1,-1)];[(0,-1)];[(-1,-1)]]
  override this.nameOfType = "king"

/// <summary> A rook is a chessPiece which moves horisontally and vertically. </…
summary>
/// <param name = "col"> The color black or white </param>
/// <returns> A rook object. </returns>
type rook(col : Color) =
  inherit chessPiece(col)
  // rook can move horisontally and vertically
  // Make a list of relative coordinate lists. We consider the
  // current position and try all combinations of relative moves
  // (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...; (0,-7).
  // Some will be out of board, but will be assumed removed as
  // illegal moves.
  // A list of functions for relative moves
  let indToRel = [
    fun elm -> (elm,0); // South by elm
    fun elm -> (-elm,0); // North by elm
    fun elm -> (0,elm); // West by elm
    fun elm -> (0,-elm) // East by elm
```

-:--- pieces.fs Top (14,0) Git:master (fsharp AC)
Beginning of buffer

chess.fs

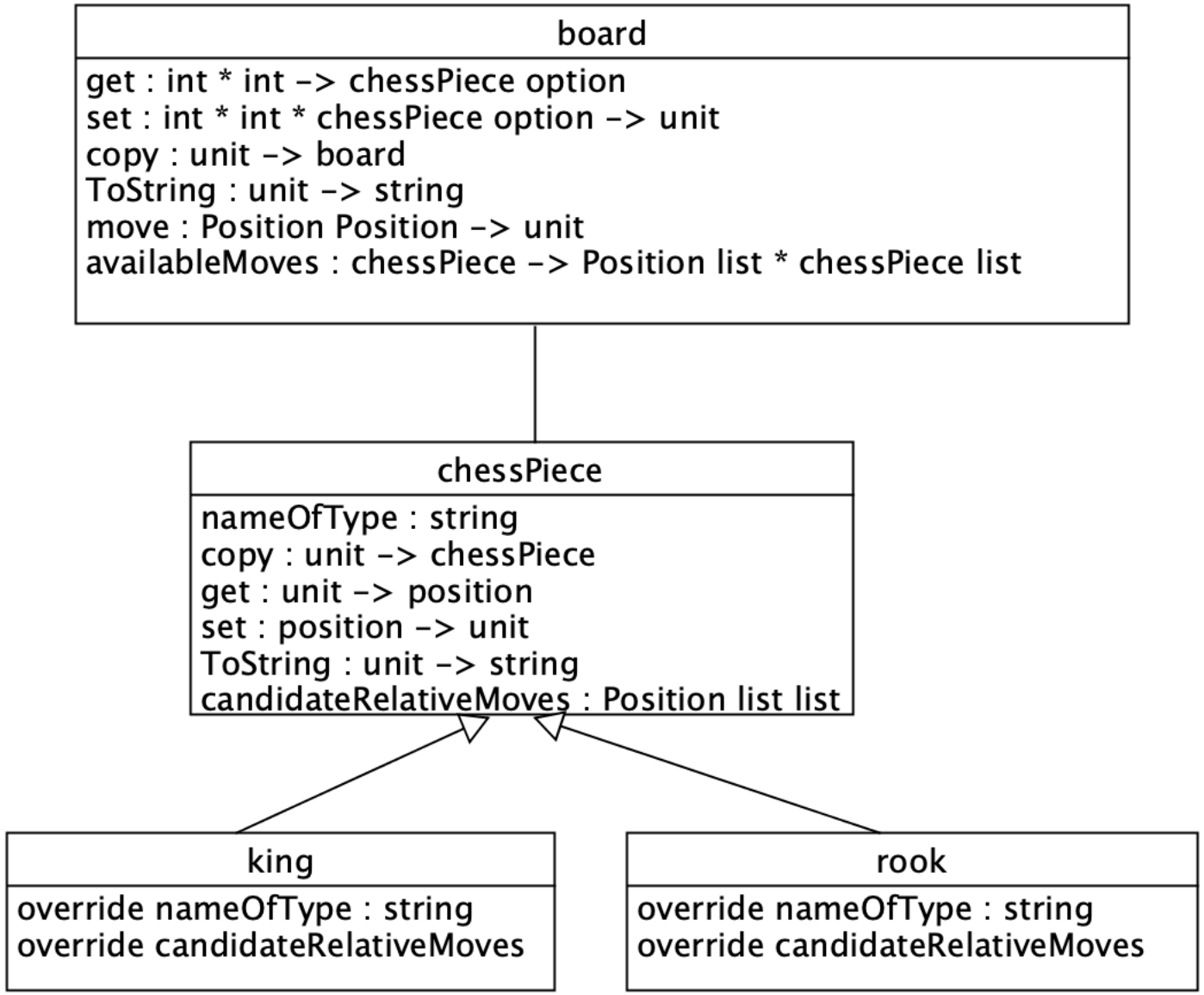
New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

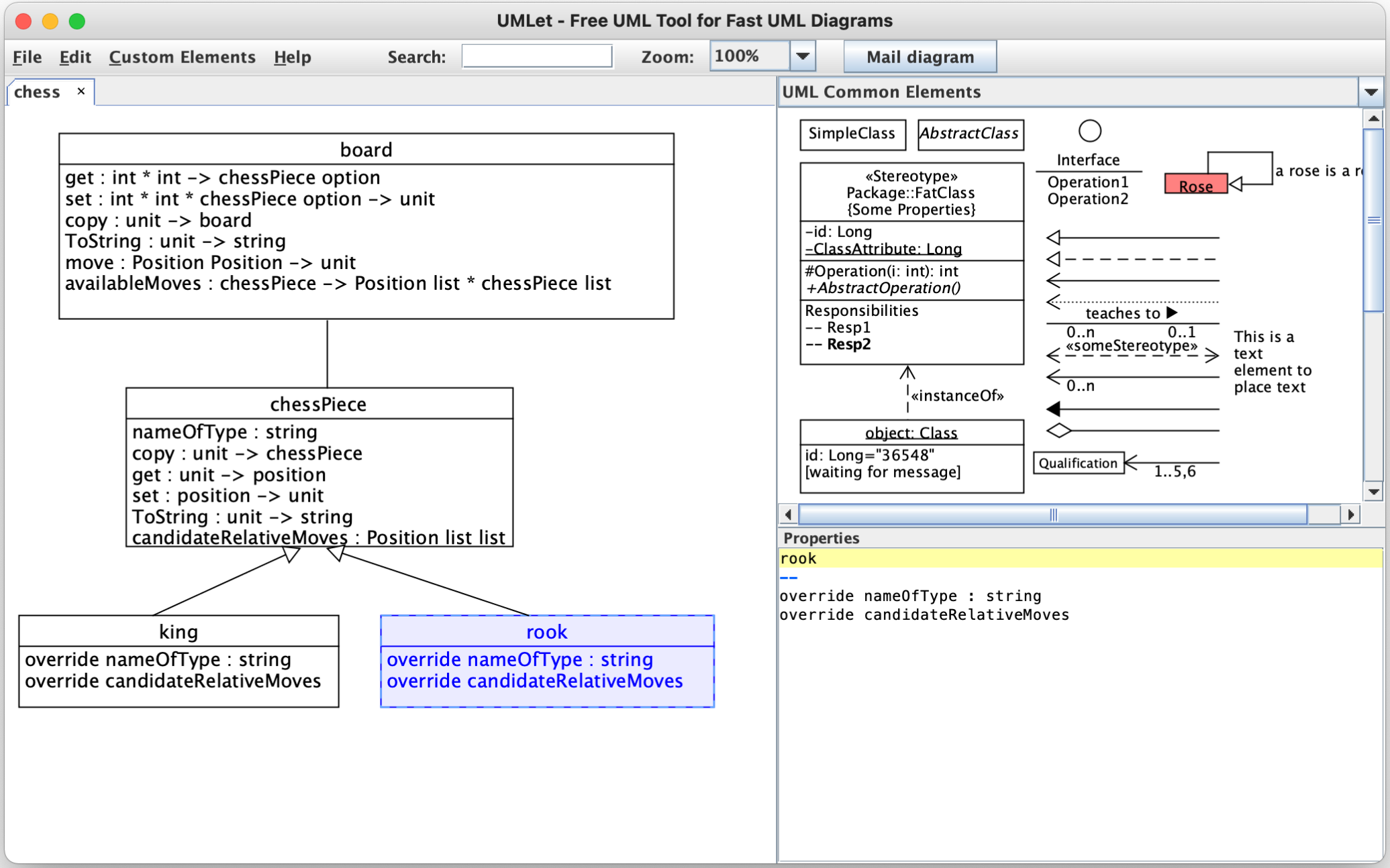
```
module Chess
/// The possible colors of chess pieces
type Color = White | Black

/// A superset of positions on a board
type Position = int * int

/// <summary> An abstract chess piece. </summary>
/// <param name = "col"> The color black or white </param>
[<AbstractClass>]
type chessPiece(color : Color) =
  let mutable _position : Position option = None

  /// The type of the chess piece as a string, e.g., "king" or "rook".
  abstract member nameOfType : string

  /// The color either White or Black
  member this.color = color

  /// The position as a Position option, e.g., None, Some (0,0), Some
  /// (3,4).
  member this.position
    with get() = _position
    and set(pos) = _position <- pos

  /// Return the first letter of the piece's type usint capital case
  /// for white pieces and lower case for black pieces. E.g., "K" and
  /// "k" for white and a black king respectively.
  override this.ToString () =
    match color with
      White -> (string this.nameOfType.[0]).ToUpper ()
      | Black -> (string this.nameOfType.[0]).ToLower ()
```

-:--- chess.fs Top (32,0) Git:master (fsharp AC)
Beginning of buffer

Klassehierarki



UMLet


UMLet - Free UML Tool for Fast UML Diagrams
File Edit Custom Elements Help Search: Zoom: 100% Mail diagram
chess
board
get : int * int -> chessPiece option
set : int * int * chessPiece option -> unit
copy : unit -> board
ToString : unit -> string
move : Position Position -> unit
availableMoves : chessPiece -> Position list * chessPiece list
chessPiece
nameOfType : string
copy : unit -> chessPiece
get : unit -> position
set : position -> unit
ToString : unit -> string
candidateRelativeMoves : Position list list
king
override nameOfType : string
override candidateRelativeMoves
rook
override nameOfType : string
override candidateRelativeMoves
UML Common Elements
SimpleClass
AbstractClass
Interface
Operation1
Operation2
Rose
a rose is a r
«Stereotype»
Package::FatClass
{Some Properties}
-id: Long
-ClassAttribute: Long
#Operation(i: int): int
+AbstractOperation()
Responsibilities
-- Resp1
-- Resp2
teaches to
0..n
0..1
«someStereotype»
This is a text element to place text
0..n
«instanceOf»
object: Class
id: Long="36548"
[waiting for message]
Qualification
1..5,6
Properties
rook
override nameOfType : string
override candidateRelativeMoves

Upcasting og downcasting med [<AbstractClass>]

Upcasting i chessApp.fsx

```
let pieces = [|
  king (White) :> chessPiece;
  rook (White) :> chessPiece;
  king (Black) :> chessPiece |]
```

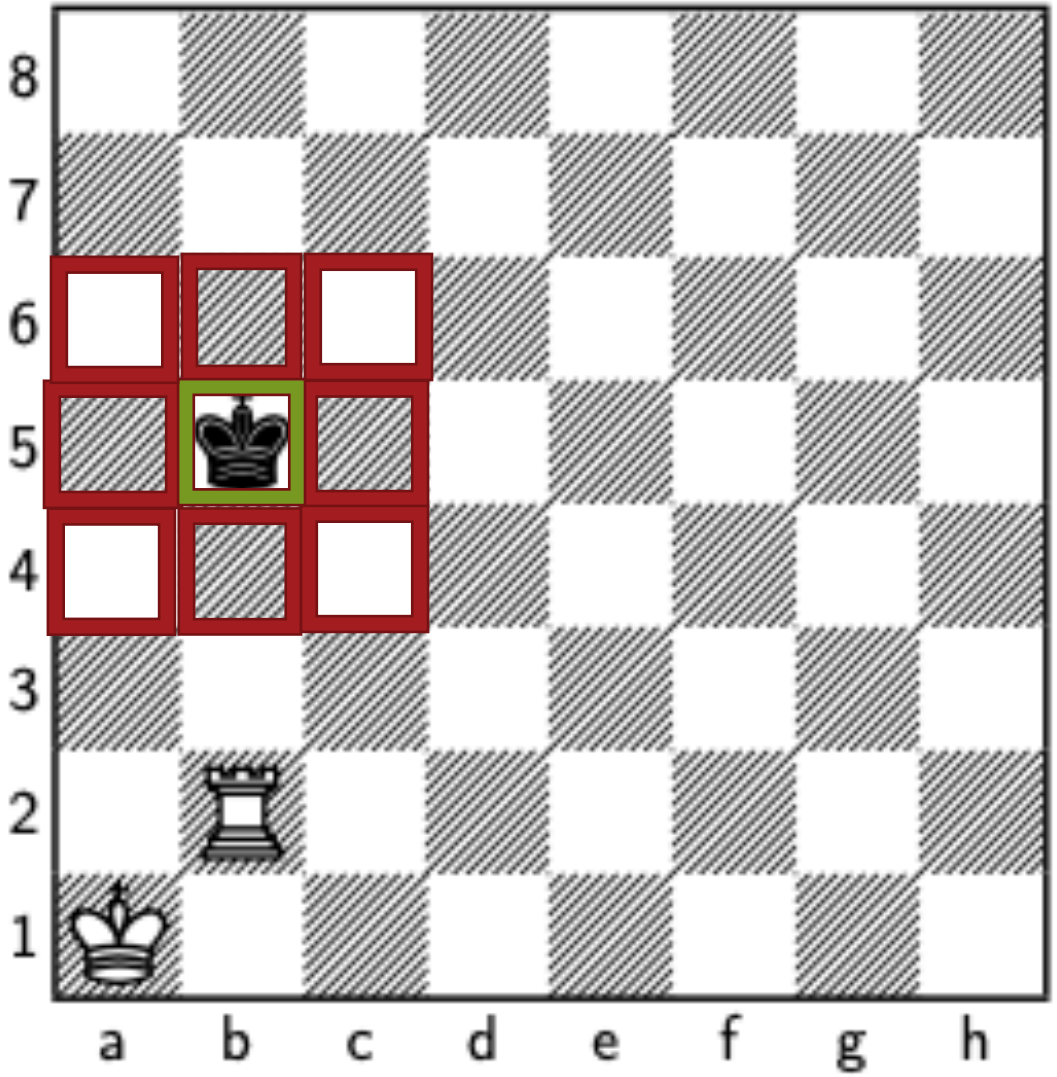
Upcasting:

+ vi kan maskere forskellige brikker som same type i listen
- Når vi skal bruge brikkernes specielle træk skal vi downcaste, men til hvilken type?

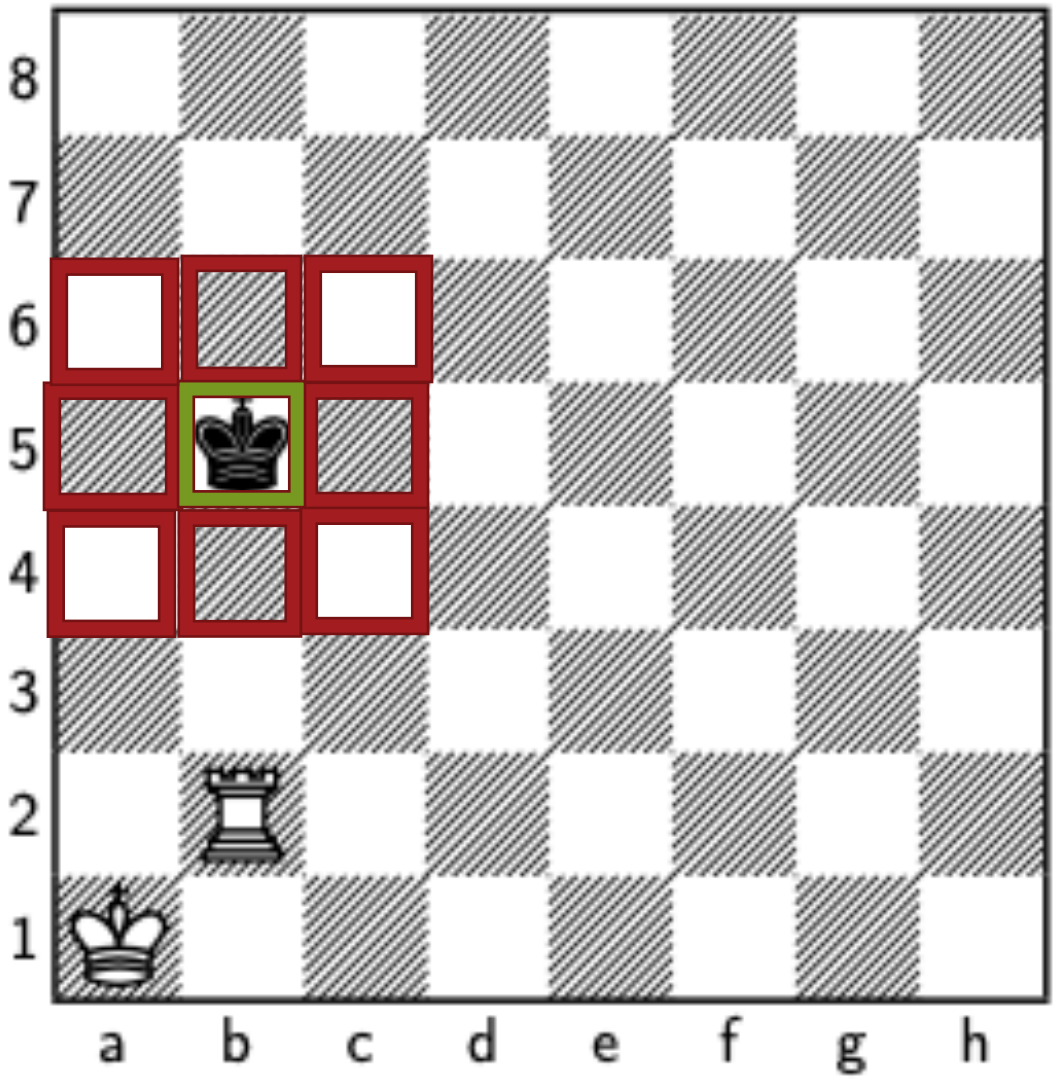
Skal brættet bruge downcasting for at få adgang til candiateRelativeMoves?

Nej! Abstrakte metoder beholder deres implementation ved upcasting

Runs: Black king - candidates



Runs: Black king - candidates



pieces.fs

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search

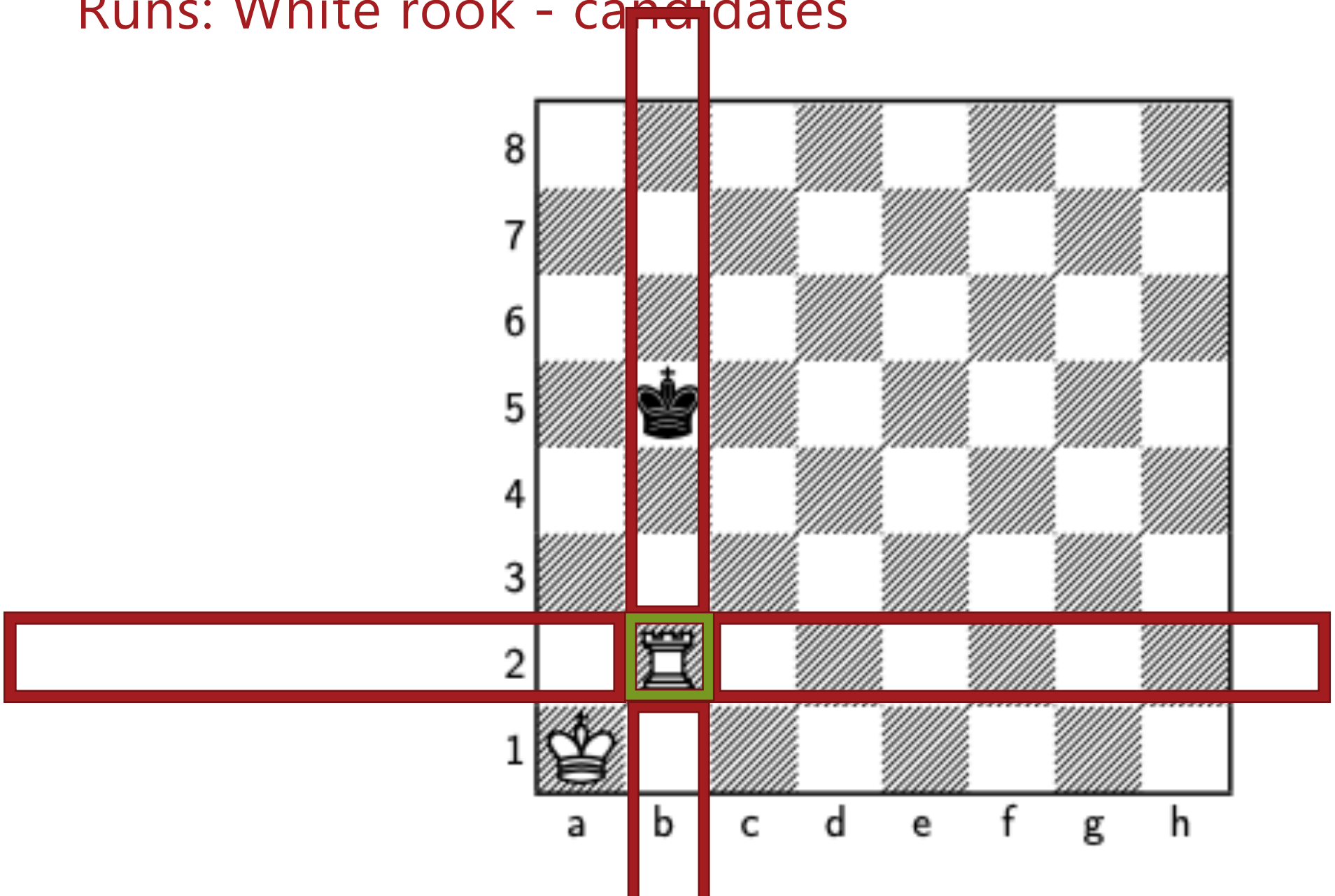
chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

```
module Pieces
open Chess

/// <summary> A king is a chessPiece which moves 1 square in any direction. </su…
mmary>
/// <param name = "col"> The color black or white </param>
/// <returns> A king object. </returns>
type king(col : Color) =

 // king has runs of 1 in 8 directions: (N, NE, E, SE, S, SW, W, NW)
 override this.candiateRelativeMoves =
     [[(-1,0)];[(-1,1)];[(0,1)];[(1,1)];
     [(1,0)];[(1,-1)];[(0,-1)];[(-1,-1)]]
 override this.nameOfType = "king"

/// <summary> A rook is a chessPiece which moves horisontally and vertically. </…
summary>
/// <param name = "col"> The color black or white </param>
/// <returns> A rook object. </returns>
type rook(col : Color) =
  inherit chessPiece(col)
  // rook can move horisontally and vertically
  // Make a list of relative coordinate lists. We consider the
  // current position and try all combinations of relative moves
  // (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...; (0,-7).
  // Some will be out of board, but will be assumed removed as
  // illegal moves.
  // A list of functions for relative moves
  let indToRel = [
    fun elm -> (elm,0); // South by elm
    fun elm -> (-elm,0); // North by elm
    fun elm -> (0,elm); // West by elm
    fun elm -> (0,-elm) // East by elm
```

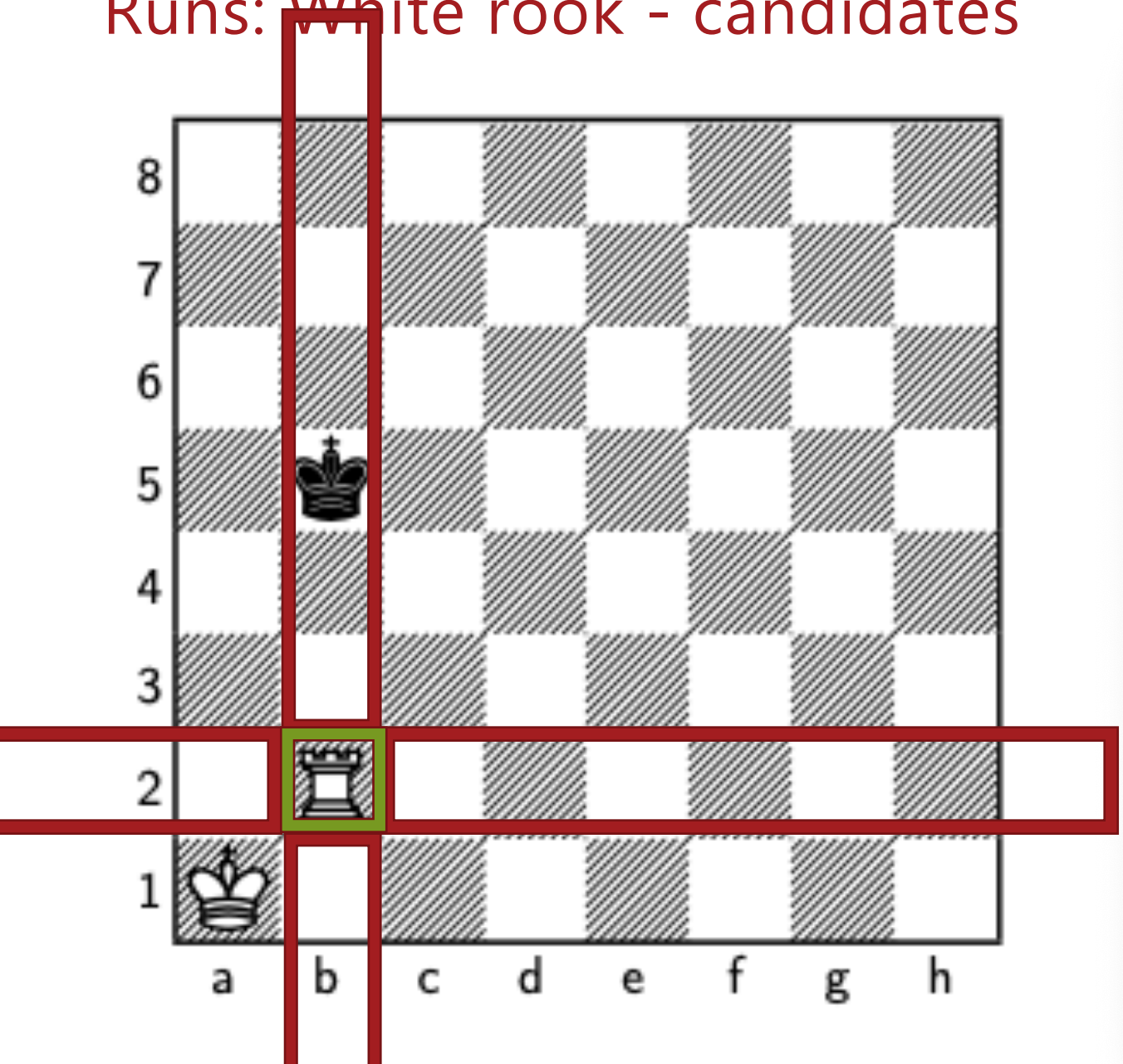
-:--- pieces.fs Top (14,0) Git:master (fsharp AC)

Beginning of buffer

Runs: White rook - candidates



Runs: White rook - candidates



pieces.fs

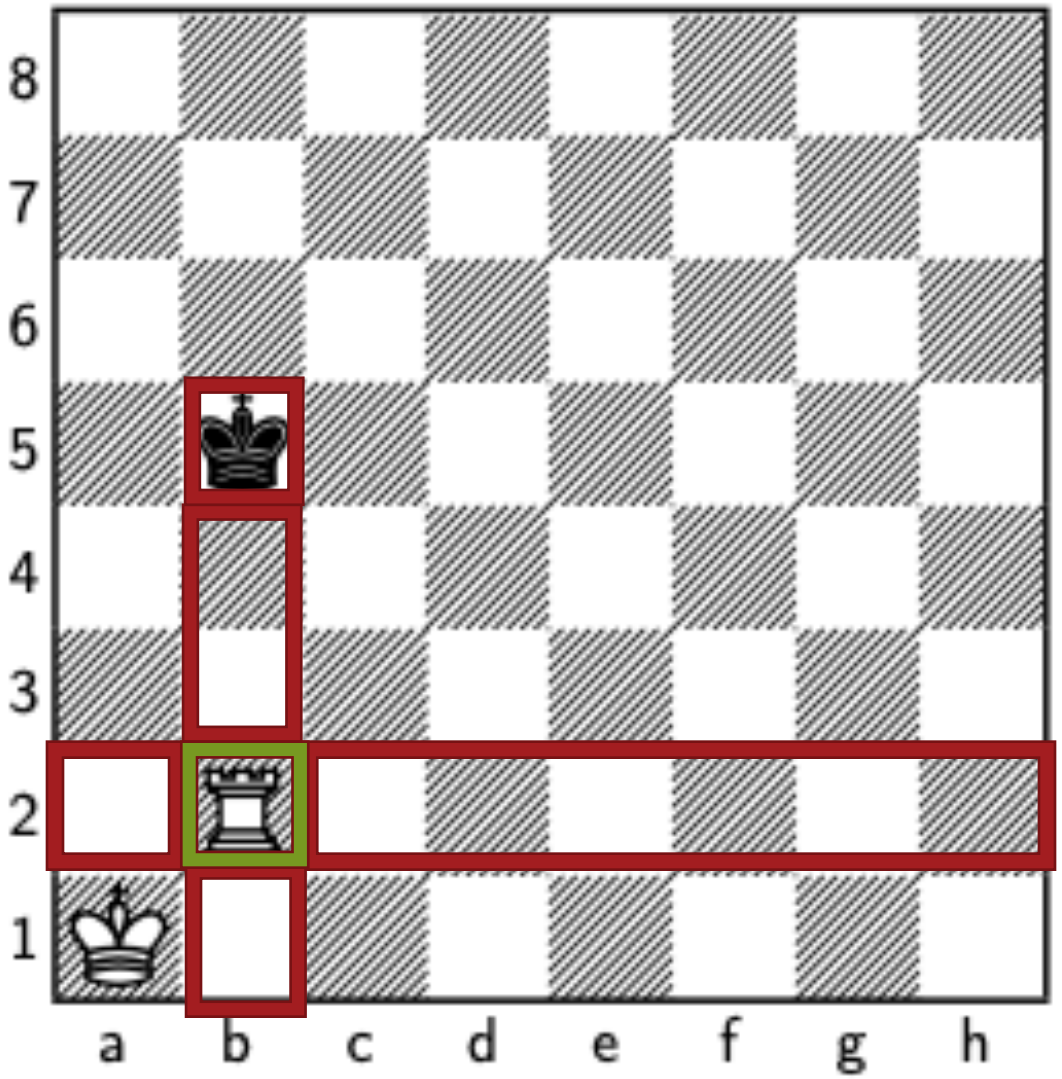
New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

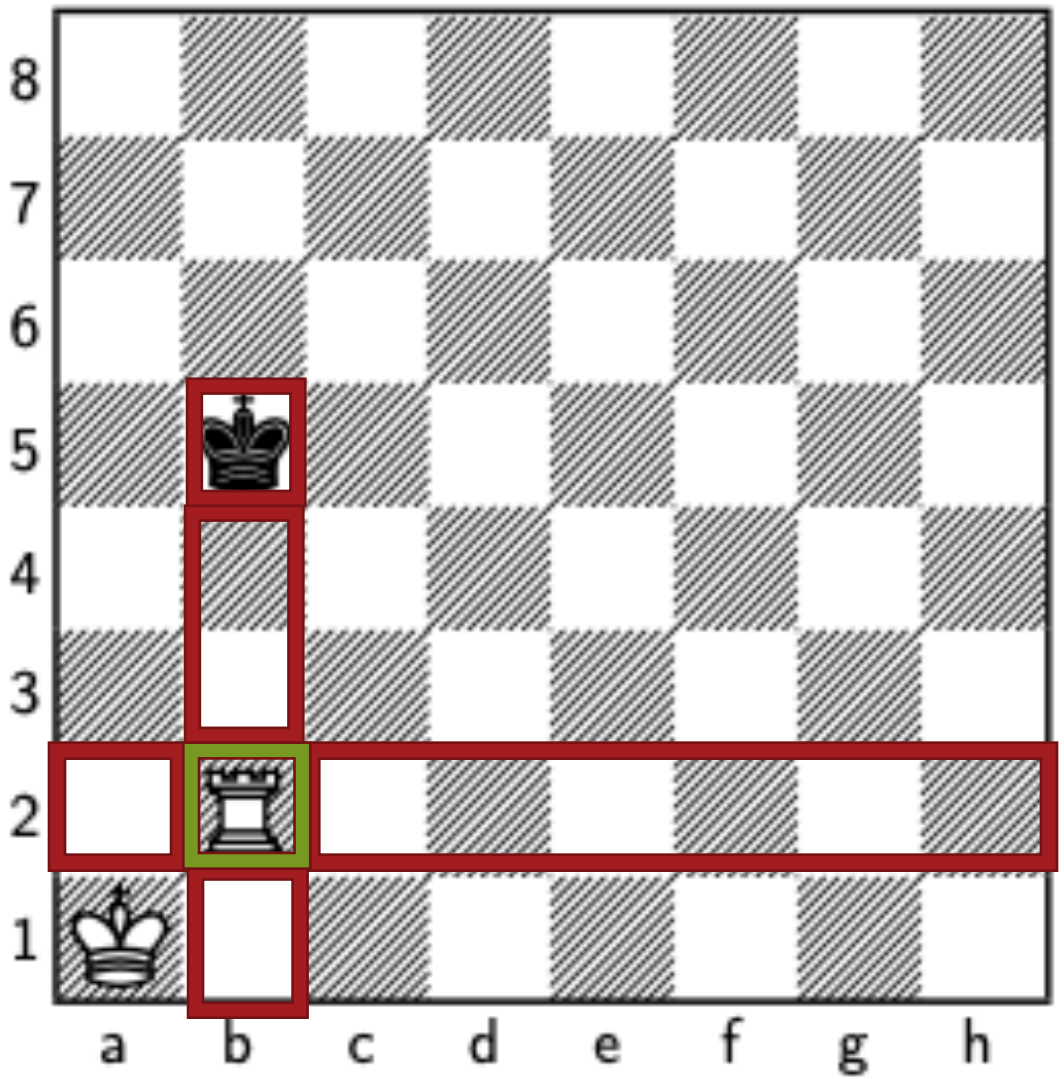
```
/// <summary> A rook is a chessPiece which moves horisontally and vertically. </summary>
/// <param name = "col"> The color black or white </param>
/// <returns> A rook object. </returns>
type rook(col : Color) =
  inherit chessPiece(col)
  // rook can move horisontally and vertically
  // Make a list of relative coordinate lists. We consider the
  // current position and try all combinations of relative moves
  // (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...; (0,-7).
  // Some will be out of board, but will be assumed removed as
  // illegal moves.
  // A list of functions for relative moves
  let indToRel = [
    fun elm -> (elm,0); // South by elm
    fun elm -> (-elm,0); // North by elm
    fun elm -> (0,elm); // West by elm
    fun elm -> (0,-elm) // East by elm
    ]
  // For each function in indToRel, we calculate List.map f [1..7].
  override this.candiateRelativeMoves =
    let rec makeRuns lst =
      match lst with
        | [] -> []
        | f :: rest ->
          let run = List.map f [1..7]
          run :: makeRuns rest
    makeRuns indToRel
```

-:**- pieces.fs Bot (34,0) Git-master (fsharp AC)

Runs: White rook – filtered and classified



Runs: candiateRelativeMoves -> availableMoves



chess.fs

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

```
/// piece on the target position is removed. </summary>
/// <param name = "source"> The source position </param>
/// <param name = "target"> The target position </param>
member this.move (source : Position) (target : Position) : unit =
  // Update piece' knowledge about it's position
  Option.iter (fun (p : chessPiece) -> p.position <- None) this.[fst target, snd target]
  Option.iter (fun (p : chessPiece) -> p.position <- Some target) this.[fst source, snd so
urce]
  // Update board's pieces
  this.[fst target, snd target] <- this.[fst source, snd source]
  this.[fst source, snd source] <- None

/// <summary> Find the list of available empty positions for this
/// piece, and the list of possible own and opponent pieces, which
/// can be protected or taken. </summary>
/// <param name = "piece"> A chess piece </param>
/// <returns> A pair of lists of all available moves and neighbours,
/// e.g., ([(1,0); (2,0);...], [p1; p2]) </returns>
member this.availableMoves (piece : chessPiece) : (Position list * chessPiece list)  =
  match piece.position with
    None ->
      ([],[])
  | Some p ->
      let convertNWrap =
        (relativeToAbsolute p) >> getVacantNOccupied
      let vacantPieceLists = List.map convertNWrap piece.candiateRelativeMoves
      // Extract and merge lists of vacant squares
      let vacant = List.collect fst vacantPieceLists
      // Extract and merge lists of first obstruction pieces and filter out own pieces
      let opponent = List.choose snd vacantPieceLists
      (vacant, opponent)
```

-:--- chess.fs Bot (127,0) Git:master (fsharp AC)

Runs: candiateRelativeMoves ->

chess.fs

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

```
/// piece on the target position is removed. </summary>
/// <param name = "source"> The source position </param>
/// <param name = "target"> The target position </param>
member this.move (source : Position) (target : Position) : unit =
  // Update piece' knowledge about it's position
  Option.iter (fun (p : chessPiece) -> p.position <- None) this.[fst target, snd target]
  Option.iter (fun (p : chessPiece) -> p.position <- Some target) this.[fst source, snd so
urce]
  // Update board's pieces
  this.[fst target, snd target] <- this.[fst source, snd source]
  this.[fst source, snd source] <- None

/// <summary> Find the list of available empty positions for this
/// piece, and the list of possible own and opponent pieces, which
/// can be protected or taken. </summary>
/// <param name = "piece"> A chess piece </param>
/// <returns> A pair of lists of all available moves and neighbours,
/// e.g., ([(1,0); (2,0);...], [p1; p2]) </returns>
member this.availableMoves (piece : chessPiece) : (Position list * chessPiece list)  =
  match piece.position with
    None ->
      ([],[])
  | Some p ->
    let convertNWrap =
      (relativeToAbsolute p) >> getVacantNOccupied
    let vacantPieceLists = List.map convertNWrap piece.candiateRelativeMoves
    // Extract and merge lists of vacant squares
    let vacant = List.collect fst vacantPieceLists
    // Extract and merge lists of first obstruction pieces and filter out own pieces
    let opponent = List.choose snd vacantPieceLists
    (vacant, opponent)
```

-:--- chess.fs Bot (127,0) Git:master (fsharp AC)

chess.fs

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

chessApp.fsx 1 | chess.fs 2 | pieces.fs 3

```
/// <summary> Wrap a position as option type. </summary>
/// <param name = "pos"> a position </param>
/// <returns> Some pos or None if the position is on the board or
/// not </returns>
let validPositionWrap (pos : Position) : Position option =
  let (rank, file) = pos // square coordinate
  if rank < 0 || rank > 7 || file < 0 || file > 7
  then None
  else Some (rank, file)

/// <summary> Converts relative coordinates to absolute and removes
/// out of board coordinates. </summary>
/// <param name = "pos"> an absolute position </param>
/// <param name = "lst"> a list of relative positions </param>
/// <returns> A list of absolute and valid positions </returns>
let relativeToAbsolute (pos : Position) (lst : Position list) : Position list =
  let addPair (a : Position) (b : Position) : Position =
    (fst a + fst b, snd a + snd b)
  // Add origin and delta positions
  List.map (addPair pos) lst
  // Choose absolute positions that are on the board
  |> List.choose validPositionWrap

/// <summary> Find the tuple of empty squares and first neighbour if any. </summary>
/// <param name = "run"> A run of absolute positions </param>
/// <returns> A pair of a list of empty neighbouring positions and
/// a possible neighbouring piece, which blocks the run. </returns>
let getVacantNOccupied (run : Position list) : (Position list * (chessPiece option)) =
  try
    // Find index of first non-vacant square of a run
    let idx = List.findIndex (fun (i, j) -> _board.[i,j].IsSome) run
    let (i,j) = run.[idx]
    let piece = _board.[i, j] // The first non-vacant neighbour
    if idx = 0
    then ([], piece)
    else (run.[..(idx-1)], piece)
  with
    _ -> (run, None) // outside the board
```

-:--- chess.fs 28% (71,0) Git:master (fsharp AC)

Resumé

I denne video hørte du om:

- Trin til at forstå kode
- En simplificeret version af den udleverede kode:
  - Imperativ udskrivning af brættet
  - Ingen copy-constructor
  - Ingen brug af swap i rook.candiateRelativeMoves.

  Den må I gerne bruge i stedet.
- En gennemgang af væsentlige elementer i den simplificerede kode:
  - UML diagram
  - Runs