ChessAce Module Interface Specification

Team 18, Team MIF
Jerry Ke, kex1
Harry fu, fuh6
Morgan Cui, cuim2

December 4, 2018

Date	Version	Notes
2018-11-9	0.0	MIS for implemented Modules

Table 1: **Revision History**

Date	Version	Notes
ϕ	Null value	
$\vert\vert$	Concatenate	
$\bigcup$	Union	

Table 2: **Table of Symbol Definitions**

Cell Module

Module

Cell

Uses

Piece, JPanel

Syntax

Exported Types

Cell = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Cell	int, int, Piece	Cell	
setPiece	Piece		
removePiece			
getPiece		Piece	
select			
isSelected		boolean	
deselect			
check			
isCheck		boolean	
removeCheck			
setPos			
isPos		boolean	
removePos			
getX		int	
getY		int	

Semantics

State Variables

$x : \mathbb{N}$
$y : \mathbb{N}$
$isChecked : \mathbb{B}$
$isSelected : \mathbb{B}$
$isPos : \mathbb{B}$
piece: Piece
content: JLabel

State Invariant

$0 \leq x \leq 7$
$0 \leq y \leq 7$

Assumptions

The necessary constructor of Cell is called for each abstract Cell object before any access routine is called for Piece. The constructor cannot be called on an existing object.

Access Routine Semantics

Cell(s0, s1, p):

- transition: $x, y, piece := s0, s1, p$
- output: $out := self$
- exception: $\neg(0 \leq s0 \leq 7) \wedge \neg(0 \leq s0 \leq 7) \Rightarrow InvalidCellException$

setPiece(p):

- transition: $piece = p$
- exception: none

removePiece():

- transition: $piece := \phi$
- exception: none

getPiece():

- output: $out := piece$
- exception: none

select():

- transition: set the JPanel *board* to red, and *isSelected* to true.
- exception: none

isSelect():

- output: return the isSelected state value.
- exception: none

deselect():

- transition: set the JPanel *board* to none, and *isSelected* to false.
- exception: none

check():

- transition: set the JPanel *background* to red, and $isCheck$ to true.
- exception: none

isCheck():

- output: return the isCheck state value.
- exception: none

removeCheck():

- transition: set the JPanel *background* to white/brown depends its (x,y) coordination, and $isCheck$ to false.
- exception: none

setPos():

- transition: set the JPanel *board* to blue, and $isPos$ to true.

- exception: none

isPos():

- output: return the *isPos* state value
- exception: none

removePos():

- transition: set the JPanel *board* to null, and *isPos* to false.
- exception: none

getX():

- output: $out := x$
- exception: none

getY():

- output: $out := y$
- exception: none

Piece Module

Module

Piece

Uses

Cell

Syntax

Exported Types

Piece = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
setMove			
isMoved		boolean	
setX	int		
setY	int		
getX		int	
getY		int	
setPath	String		
getPath		String	
setId	String		
getId		String	
setColor	int		
getColor		int	
isKingInDanger	sequence of sequence of cell	boolean	

Semantics

State Variables

$neverMoved : \mathbb{B}$
$color : \mathbb{N}$
Id: String
$path$: String

$possiblemoves$: Sequence of Cell
$posMove$: abstract function
$x : \mathbb{N}$
$y : \mathbb{N}$

State Invariant

$0 \leq x \leq 7$
$0 \leq y \leq 7$

Assumptions

The necessary constructor of Cell is called for each abstract Cell object before any posMove() is called for any subclass of Piece.

Access Routine Semantics

setMove():

- transition: $neverMoved := false$
- exception: none

isMoved():

- output: $out := neverMoved$
- exception: none

setX(s):

- transition: $x := s$
- exception: none

setY(s):

- transition: $y := s$
- exception: none

getX():

- output: $out := x$

- exception: none

getY():

- output: $out := y$
- exception: none

setPath(p):

- transition: $path := p$
- exception: none

getPath():

- output: $out := path$
- exception: none

setId(d):

- transition: $Id := d$
- exception: none

getId():

- output: $out := Id$
- exception: none

setColor(c):

- transition: $color := c$
- exception: none

getColor():

- output: $out := color$
- exception: none

isKingInDanger(pos): This method has been moved from sub-class to Piece module

- output: check if cell is in the attack range of other hostile piece's attack range based on different piece types.

  $out :=$
  $\exists(r : Rook \mid r \in Rook(x, y).posMove(pos) \wedge r.color \neq color)$
  $\vee$
  $\exists(n : Knight \mid n \in Knight(x, y).posMove(pos) \wedge n.color \neq color)$
  $\vee$
  $\exists(k : King \mid k \in King(x, y).posMove(pos) \wedge k.color \neq color)$
  $\vee$
  $\exists(b : Bishop \mid b \in Bishop(x, y).posMove(pos) \wedge b.color \neq color)$
  $\vee$
  $\exists(q : Queen \mid q \in Queen(x, y).posMove(pos) \wedge q.color \neq color)$
  $\vee$
  $\exists(p : Pawn \mid p \in Pawn(x, y).posMove(pos) \wedge p.color \neq color)$

- exception: none

Bishop Module

Template Module

Bishop

Uses

Piece, Cell

Syntax

Exported Types

Bishop = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Bishop	String, String, int, int, int	Bishop	InvalidCellException
posMove	sequence of sequence of Cell	sequence of Cell	

Semantics

State Variables

none

State Invariant

none

Assumptions

none

Access Routine Semantics

Bishop($d, p, x0, y0, c$):

- transition: $Id, path, x, y, color := d, p, x0, y0, c$
- output: $out := self$

- exception: $\neg(0 \leq x0 \leq 7) \wedge \neg(0 \leq y0 \leq 7) \Rightarrow InvalidCellException$

posMove(pos):

- output: Check for all cells of diagonal direction, if there has no other piece, add this cell to the posmove, if there has the other piece, stop adding new cell for this direction and return posmove.

  $out :=$
  $(i : \mathbb{N}, j : \mathbb{N} \mid x+1 \leq i \leq 7 \;\wedge\; 0 \leq j \leq y-1 \wedge$
  $(|x-i| = |y-j|) \wedge (pos[i][j].getPiece() = \phi \vee pos[i][j].getColor() \neq this.getColor) \wedge$
  $(pos[i][j].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[i][j])$
  $\bigcup$
  $(i : \mathbb{N}, j : \mathbb{N} \mid 0 \leq i \leq x-1 \wedge y+1 \leq j \leq 7 \wedge$
  $(|x-i| = |y-j|) \wedge (pos[i][j].getPiece() = \phi \vee pos[i][j].getColor() \neq this.getColor) \wedge$
  $(pos[i][j].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[i][j])$
  $\bigcup$
  $i : \mathbb{N}, j : \mathbb{N} \mid 0 \leq i \leq x-1 \wedge 0 \leq j \leq y-1 \wedge$
  $(|x-i| = |y-j|) \wedge (pos[i][j].getPiece() = \phi \vee pos[i][j].getColor() \neq this.getColor) \wedge$
  $(pos[i][j].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[i][j])$
  $\bigcup$
  $i : \mathbb{N}, j : \mathbb{N} \mid x+1 \leq i \leq 7 \;\wedge\; y+1 \leq j \leq 7 \;\wedge$
  $(|x-i| = |y-j|) \wedge (pos[i][j].getPiece() = \phi \vee pos[i][j].getColor() \neq this.getColor) \wedge$
  $(pos[i][j].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[i][j])$

- exception: none

Knight Module

Template Module

Knight

Uses

Piece, Cell

Syntax

Exported Types

Knight = ?

Exported Constants

X, Y := {2, 1 , -1, -2, -2, -1, 1, 2}, {1, 2, 2, 1, -1, -2, -2, -1}

Exported Access Programs

Routine name	In	Out	Exceptions
Knight	String, String, int, int, int	Knight	InvalidCellException
posMove	sequence of sequence of Cell	sequence of Cell	

Semantics

State Variables

none

State Invariant

none

Assumptions

none

Access Routine Semantics

Knight$(d, p, x0, y0, c)$:

- transition: $Id, path, x, y, color := d, p, x0, y0, c$
- output: $out := self$
- exception: $\neg(0 \leq x0 \leq 7) \wedge \neg(0 \leq y0 \leq 7) \Rightarrow InvalidCellException$

posMove(pos):

- output: Check for any of the closest squares that are not on the same rank, file or diagonal, if there has no other piece, add this cell to the posmove, if there has another piece, do not add this cell to the posmove. After done the checking, return the posmove.

  $out := (i : \mathbb{N} \mid 0 \leq i \leq 7 \ \wedge \ (0 \leq X[i] \ + \ x \leq 7) \ \wedge \ (0 \leq Y[i] \ + \ y \leq 7) \ \wedge \ ((pos[X[i] \ + \ x][Y[i] \ + y].getPiece() = \phi) \vee (pos[X[i] \ + \ x][Y[i] \ + y].getColor() \neq this.getColor())) \ \wedge \ (pos[X[i] \ + \ x][Y[i] \ + y].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[X[i] \ + \ x][Y[i] \ + y])$

- exception: none

Pawn Module

Template Module

Pawn

Uses

Piece, Cell

Syntax

Exported Types

Pawn = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Pawn	String, String, int, int, int	Pawn	InvalidCellException
setPromo			
getPromo		boolean	
posMove	sequence of sequence of Cell	sequence of Cell	

Semantics

State Variables

$promoPossible : \mathbb{B}$

State Invariant

none

Assumptions

none

Access Routine Semantics

Pawn($d, p, x0, y0, c$):

- transition: $Id, path, x, y, color, promoPossible := d, p, x0, y0, c, false$
- output: $out := self$
- exception: $\neg(0 \leq x0 \leq 7) \wedge \neg(0 \leq y0 \leq 7) \Rightarrow InvalidCellException$

setPromo():

- transition: promoPossible = true
- exception: none

getPromo():

- output: return the promoPossible state value.
- exception: none

posMove(pos):

- transition:Set promoPossible to true if the pawn reaches the other end of the board. $(this.getColor() = 1 \Rightarrow (this.x = 0) \Rightarrow (promoPossible := true)) \wedge$ $(this.getColor() = 0 \Rightarrow (this.x = 7) \Rightarrow (promoPossible := true))$

- output: Pawn only moves one step except the first step, for the first step it may move one or two steps, check if it is the first step, if true add two cells to pos move, if false add one cell to the posmove.
  Pawn can only move in a diagnoal when it is attacking a piece of opposite color. Check if there exists enermy piece on the diagonal cell, if true call elemination, if false, do not add this cell to the posmove.

  out := $(this.getColor() = 1 \Rightarrow$
  $\{pos[x-1][y].getPiece() = \phi) : pos[x+1][y],$

  $(pos[x-1][y].getPiece() = \phi \ \wedge \ x = 6 \ \wedge \ pos[4][y].getPiece() = \phi) : pos[4][y],$

  $(y > 0) \wedge (pos[x-1][y-1].getPiece() \neq \phi) \wedge (pos[x-1][y-1].getPiece().getColor() \neq this.getColor()) : pos[x-1][y-1],$

$(y < 7) \wedge (pos[x{-}1][y{+}1].getPiece() \neq \phi) \wedge (pos[x{-}1][y{+}1].getPiece().getColor() \neq this.getColor()) : pos[x-1][y+1] : pos[x-1][y+1]\}$

$\wedge$

$(this.getColor() = 0 \Rightarrow$
$\{pos[x+1][y].getPiece() = \phi) : pos[x-1][y],$

$(pos[x+1][y].getPiece() = \phi \ \wedge \ x = 1 \ \wedge \ pos[3][y].getPiece() = \phi) : pos[3][y],$

$(y > 0) \wedge (pos[x{+}1][y{-}1].getPiece() \neq \phi) \wedge (pos[x{+}1][y{-}1].getPiece().getColor() \neq this.getColor()) : pos[x+1][y-1],$

$(y < 7) \wedge (pos[x{+}1][y{+}1].getPiece() \neq \phi) \wedge (pos[x{+}1][y{+}1].getPiece().getColor() \neq this.getColor()) : pos[x+1][y+1] : pos[x+1][y+1]\}$

- exception: none

Rook Module

Template Module

Rook

Uses

Piece, Cell

Syntax

Exported Types

Rook = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Rook	String, String, int, int, int	Rook	InvalidCellException
posMove	sequence of sequence of Cell	sequence of Cell	

Semantics

State Variables

none

State Invariant

none

Assumptions

none

Access Routine Semantics

Rook$(d, p, x0, y0, c)$:

- transition: $Id, path, x, y, color, neverMoved := d, p, x0, y0, c, true$
- output: $out := self$

- exception: $\neg(0 \leq x0 \leq 7) \wedge \neg(0 \leq y0 \leq 7) \Rightarrow InvalidCellException$

posMove(pos):

- output: Basic Movement, rook can be move or attack cells one unit beside him in any directions. Check cells of all directions, if there has no other piece, add this cell to the posmove, if there exists other piece, stop adding new posmove from this direction. After checking all cells, return the posmove.

  $out := (i : \mathbb{N} \mid x + 1 \leq i \leq 7 \wedge$
  $(pos[i][y].getPiece() = \phi \vee pos[i][y].getColor() \neq this.getColor) \wedge (pos[i][y].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[i][y])$
  $\bigcup$
  $(i : \mathbb{N} \mid 0 \leq i \leq x - 1 \wedge$
  $(pos[i][y].getPiece() = \phi \vee pos[i][y].getColor() \neq this.getColor) \wedge (pos[i][y].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[i][y])$
  $\bigcup$
  $(i : \mathbb{N} \mid 0 \leq j \leq y - 1 \wedge$
  $(pos[x][j].getPiece() = \phi \vee pos[x][j].getColor() \neq this.getColor) \wedge (pos[x][j].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[x][j])$
  $\bigcup$
  $(i : \mathbb{N} \mid y + 1 \leq j \leq 7 \wedge$
  $(pos[x][j].getPiece() = \phi \vee pos[x][j].getColor() \neq this.getColor) \wedge (pos[x][j].getColor() = this.getColor() \Rightarrow \textbf{break}) : pos[x][j])$

- exception: none

Queen Module

Template Module

Queen

Uses

Piece, Cell

Syntax

Exported Types

Queen = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Queen	String, String, int, int, int	Queen	InvalidCellException
posMove	sequence of sequence of Cell	sequence of Cell	

Semantics

State Variables

none

State Invariant

none

Assumptions

none

Access Routine Semantics

Queen($d, p, x0, y0, c$):

- transition: $Id, path, x, y, color := d, p, x0, y0, c$
- output: $out := self$

- exception: $\neg(0 \leq x0 \leq 7) \wedge \neg(0 \leq y0 \leq 7) \Rightarrow InvalidCellException$

posMove(pos):

- output: Queen can move horizontally, vertically and diagnoally for any cell only other piece block its way. Check for these direction, if there has no other piece exist, add this cell to the posmove, if there has other piece exist, stop adding new cell from this direction. After checking all cells, return the posmove.

  $out :=$
  Rook.posMove(pos) $\bigcup$ Bishop.posMove(pos)

King Module

Template Module

King

Uses

Piece, Cell

Syntax

Exported Types

King = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
King	String, String, int, int, int	King	InvalidCellException
posMove	sequence of sequence of Cell	sequence of Cell	
isKingInDanger	sequence of sequence of Cell	bool	

Semantics

State Variables

$notCheckmated : \mathbb{B}$

State Invariant

none

Assumptions

none

Access Routine Semantics

King($d, p, x0, y0, c$):

- transition: $Id, path, x, y, color, neverMoved, notCheckmated := d, p, x0, y0, c, true, true$

- output: $out := self$
- exception: $\neg(0 \leq x0 \leq 7) \wedge \neg(0 \leq y0 \leq 7) \Rightarrow InvalidCellException$

posMove(pos):

- output: Check the 8 Cells around King is valid for moving, and also check the two cells for castling are valid or not.

  $out :=$
  $(i : \mathbb{N} \mid (-1 \leq i \leq 1) \wedge (-1 \leq j \leq 1) \ \wedge \ (0 \leq i + x \leq 7) \ \wedge \ (0 \leq j + y \leq 7) \ \wedge \ (pos[x + i][y + j].getPiece() = \phi \vee pos[x + i][y + j].getPiece().getColor() \neq this.getColor()) : pos[x + i][y + j])$

  $\bigcup$

  $(neverMoved \wedge notCheckmated) \Rightarrow ($
  $(i : \mathbb{N} \mid i \in \{1, 2\} \ \wedge \ pos[this.x][this.y + i].getPiece() \neq \phi : pos[this.x][this.y + 2])$

  $\bigcup$

  $(j : \mathbb{N} \mid i \in \{1, 2, 3\} \wedge pos[this.x][this.y - j].getPiece() \neq \phi : pos[this.x][this.y - 2]))$

- exception: none

Main Module

Main Module

Main

Uses

Piece, Cell, Rook, Pawn, Knight, King, Queen, Bishop

Syntax

Exported Types

Main = ?

Exported Constants

Exported Access Programs

Routine name	In	Out	Exceptions
Main		Main	
move	Cell, Cell		
isCastling	Cell, Cell	int	
retrieveKing	int	King	
transform	Cell		
filterDestination	Piece, sequence of Cell		
existPos*	int*	boolean	
mouseClicked	int, int		WrongPlayerException, DuplicateCellException
gameOver*			
playerChange*			

* means modified

Semantics

State Variables

pos: sequence of sequence of Cell
isCheckmate: bool
Black: Sequence of Piece

$White$: Sequence of Piece
$player$: int

State Invariant

Assumptions

Access Routine Semantics

Main():

- transition: set up the chess board based on the basic rule.

  $Black, White := \{Rook, Knight, Bishop, Queen, King, Bishop, Knight, Rook\}_{black}$, $\{Rook, Knight, Bishop, Queen, King, Bishop, Knight, Rook\}_{white}$
  $\forall(i : \mathbb{N} \mid 0 \leq i \leq 7\ : board[1][i] = Pawn_{black} \wedge board[6][i] = Pawn_{white} \wedge board[0][i] = Black[i]\ \wedge\ board[7][i] = White[i]$

- output: $out := self$

- exception: none

move(a, b):

- transition: $b.piece := a.piece \Rightarrow b.piece.x, b.piece.y := b.x, b.y$

- exception: none

Castling(k, kt, r, rt)

- transition: $move(k, kt) \wedge move(r, rt)$

- exception: none

isCastling(p, c):

- out: Check if p is an instance of King and never moved before. If so, depends on the direction it is moving to, it shall check whether the correspondsing rook has been moved before or not. Depnds on the direction it is moving to, it shall return different numerical value after all verification passes.

  $(p \in King \wedge \neg(p.getPiece().isMoved())) \Rightarrow$
  $((p.y - c.y = 2\ \wedge\ pos[c.x][c.y-2].getPiece() \in Rook\ \wedge\ \neg((pos[c.getX()][c.getY()-2].getPiece().isMoved()))) \Rightarrow 0$

$\wedge$
$(p.y - c.y = -2 \wedge pos[c.x][c.y+1].getPiece() \in Rook \wedge \neg((pos[c.getX()][c.getY()+1].getPiece().isMoved()))) \Rightarrow 1$
$\wedge$
$((p.y - c.y) \notin \{2, -2\}) \Rightarrow -1$
$\wedge$
$\neg(p \in King \wedge \neg(p.getPiece().isMoved())) \Rightarrow -1)$

- exception: none

retrieveKing(c):

- out: $((c = 1) \Rightarrow King_{white}) \wedge ((c = -1) \Rightarrow King_{Black}) \wedge c \notin \{1, -1\} \Rightarrow \phi$
- exception: none

transform(c):

- transition: Check if the pawn's next move will reach the other end of the board. If so, transform the pawn to queen with the same color after the movement.

  $c.getPiece().getColor() = 1 \Rightarrow c.setPiece(Queen_{White}) \wedge c.getPiece().getColor() = -1 \Rightarrow c.setPiece(Queen_{Black})$
- exception: $exc := (c \notin \{1, -1\} \Rightarrow$ NullTransnformException)

filterDestination(p, a):

- output: If p is an instance of King, keep the move in posMove list that will not cause a stalemate. If p is not an instance of King, keep the move in posMove list that will not cause friendly king to be in stalemate.

  $p \in King \Rightarrow (i : Cell | i \in p.posMove() \;\wedge\; \neg(move(p.Cell, i)$
  $\Rightarrow p.isKingInDanger()) : i)$
  $\wedge$
  $p \notin King \Rightarrow (i : Cell | i \in p.posMove() \;\wedge\; \neg(move(p.Cell, i)$
  $\Rightarrow retrieveKing(p.color).isKingInDanger()) : i)$
- exception: none

existPos(c):

- output: Depends on the input color c, then iterate through each piece on the side of c, and immediately return true if any piece has a feasible move path.

  $out :=$
  $c = 1 \Rightarrow \exists(p : Piece \mid p \in$white: $\neg(filterDestination(p.p.posMove(pos)) = \phi))$
  $\wedge$
  $c = -1 \Rightarrow \exists(p : Piece \mid p \in$black: $\neg(filterDestination(p.p.posMove(pos)) = \phi))$

- exception: none

mouseClicked(a, b):

- transition: check if the cur is empty or contains hostile piece respect to the pre cell. If so,check if the current cell is a valid move respect to the pre cell piece. Depends on the position and type of the piece, identify movement and transform possibility.

  King: castling and normal movement, and set to move after movement.

  Pawn: auto transform when reach the other end and normal movement.

  Rook: set to moved after movement.

  Other: normal movement

  all normal movement just call move() review the check status on the hostile king and remove self check status. If no movement possible ⇒ GameOver switch side.

- exception: $(pre = \phi \wedge pos[a][b] \neq \phi \wedge pos[a][b].color \neq player) \Rightarrow WrongPlayerException$

- exception: $(pre.x = a \wedge pre.y = b) \Rightarrow DuplicateCellException$

gameOver():

- transition: prompt message that the current player has won the game, and re-initiate the game.

- exception: none

playerChange():

- transition: $player = player * -1$

- exception: none

countDownTimer Module

countDownTimer Module

countDownTimer

Uses

Main

Syntax

Exported Types

countDownTimer = ?

Exported Constants

N/A

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
countDownTimer	JLabel, Main	countDownTimer	
start			
stop			
reset			
actionPerformed	ActionEvent		

Semantics

State Variables

$countdownTimer : Timer$
$label : JLabel$
$timeRemain : \mathbb{N}$
$m : Main$

State Invariant

Assumptions

countDownTime will only be constructed during the construction of Main instance.

Access Routine Semantics

countDownTimer(p, m0):

- transition: set countdownTimer, and integrate it with a listener that listens to specific event.

  $label$ = p, m = m0, $timeRemain$ = m0.timeRemain

- output: $self$

- exception: none

start():

- transition: start $countdownTimer$

- exception: none

stop():

- transition: stop $countdownTimer$

- exception: none

reset():

- transition: reset $countdownTimer$ to the initial time setting stored in m(Main)

- exception: none

actionPerformed():

- transition: After every second, check if there is still time left. If so, formulate the time in min:sec, and display it on the GUI. Discrement $timeRemain$ by 1.
  If not, send notification of GameOver state, and initial GameOver routine in Main module.

- exception: none