Checkers Design Document

Group 2

Contents

1 Introduction

This document contains the decomposition, uses relationship, traceability, and internal evaluation. Note: Red links and the Uses diagram are clickable hyperlinks (depending on your PDF reader).

2 Requirements of Checkers

1. Assignment 1 Requirements
   1.1. Must set up an 8x8 checkers board
      1.1.1. Squares will be either light or dark
      1.1.2. The Bottom right square must be light
   1.2. User must be able to choose the standard set up
   1.3. Board Rules
      1.3.1. User must be able to specify starting position of each piece
      1.3.2. Notation for specifying piece location must use standard form (A7 = B)
      1.3.3. User should be able to specify type (normal or king)
      1.3.4. If they specified every pieces starting position, user must be able to indicate if the set up is complete
      1.3.5. User should be able to clear the board
   1.4. Maximum of 12 white and 12 black pieces can be placed on the board
   1.5. Illegal placement notification:
      1.5.1. User should be notified if a piece choice is illegal
      1.5.2. A piece on a light square
      1.5.3. Exceeding the maximum number
      1.5.4. Spelling/ typing error
      1.5.5. There is already a piece there
   1.6. User should be notified if there in an inappropriate number of pieces on the board Inappropriate includes:
      1.6.1. Blank board
2. Assignment 2 Requirements
   2.1. Load Saves
      2.1.1. Start Game from original starting positions
      2.1.2. Start a game from a previously stored state from a within a file
      2.1.3. Save a game to be resumed later
   2.2. Legal Moves and Crowning
      2.2.1. Make moves from one position to another, while making sure the move made is legal.

2.2.2. Simply move a piece to another square; jump the opponents piece (so that piece is removed from the board).

2.2.3. Crowning a piece to king

2.2.4. move kings in both directions (forwards and backwards).

2.2.5. Graphically or through code indicate possible movements.

3. Assignment 3 Requirements

3.1. Two Player Game mode

3.1.1. Two humans must be allowed to play against each other

3.1.2. GUI menu must allow for this type of interaction

3.1.3. Must allow for turn based action depending on the player

3.2. Single Player Against AI

3.2.1. GUI menu must allow for selection of Human vs AI mode

3.2.2. Both AI and Human opponent must only be allowed to make legal moves such as:

- Move a piece to another square
- jump the opponent's piece (so that piece is removed from the board)
- convert a piece to a king
- move kings in both directions (forwards and backwards)
- If a capture move is possible it must be taken. If more than one such move is possible, the player chooses which one to make.

3.2.3. Allowing the player to resign the game

3.2.4. Indicating moves graphically or by code (E3-D4 etc), or both.

3.3. Game play rules

3.3.1. Indicate if the game has been won

3.3.2. Include any on screen help you think is necessary or helpful.

3 Module Guide

3.1 Hardware Hiding Module

3.1.1 Input Module

Type	Hardware Module
Secret	This module translates mouse clicks and keyboard presses to be used by the rest of the software.
Requirements	None
Responsibilities	This module will take mouse and keyboard input and convert it to software usable states.
Uses	None
Design	7.5
Code File	Inside Game1.cs, and built into C#.
Explanation	The input module is a hardware hiding module since it translates hardware inputs to software.

3.2 Behaviour Hiding Module

3.2.1 Piece Module

Type	Software Module
Secret	This module hides and separates specific piece information.
Requirements	1.3.3.
Responsibilities	This will hold the necessary components to describe what a game piece will contain, which will be separate from the game board.
Uses	None
Design	7.1
Code File	Piece.cs
Explanation	The piece is a part of behaviour hiding since the piece module holds specific piece information and outputs values needed by other modules.

3.3 Software Decision Hiding Module

3.3.1 Game1 Module

Type	Software Module
Secret	This module hides how the graphics are displayed and how we switch between states of the game. This module has also the responsibility of giving the valid moves of any piece on the board.
Requirements	1.2. 1.3.1. 1.3.2. 1.3.4. 1.3.4. 2.2.
Responsibilities	This module will be the responsible for the initial execution of the game, this class connects and launches critical components together.
Uses	3.3.2, 3.2.1, 3.1.1
Design	7.3
Code File	Game1.cs
Explanation	The module is a part of software decision hiding since it determines how we draw the graphics and what to do when we switch between states of the game.

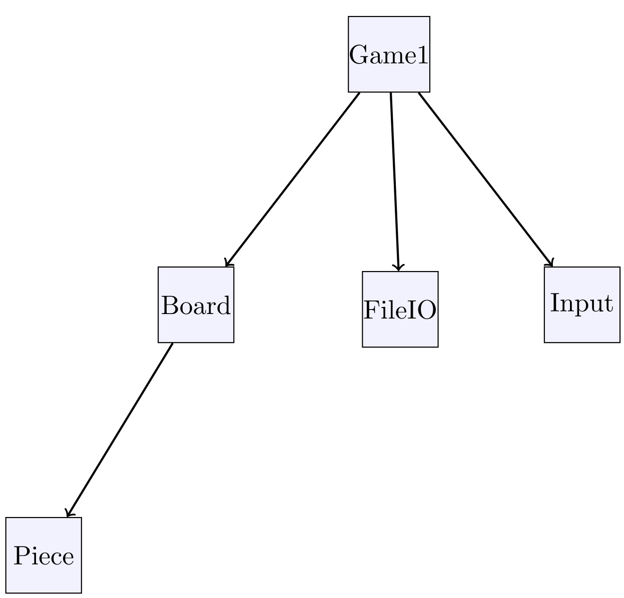
3.3.2 Board Module

Type	Software Module
Secret	This module serves to hide the secret of how the board is defined internally.
Requirements	1.1. 1.3.1. 1.3.2. 1.3.3. 1.3.5. 1.4. 1.5. 1.6.
Responsibilities	This module is responsible for holding the necessary components and attributes to setup the board and describe piece locations.
Uses	3.2.1
Design	7.2
Code File	Board.cs
Explanation	The board is a part of software decision hiding since the board implements a data structure that holds the placement of the pieces, this data structure might be changed for increased performance. Another software decision is deciding how to take user input to parse the placement of pieces.

3.3.3 FileIO Module

Type	Software Module
Secret	This module allows the user to save the current game session as a plain text file and also load previous games by parsing the plain text file into a representation of the board.
Requirements	1.3.2. 2.1.
Responsibilities	This module is responsible for the load and saving of the current game
Uses	none
Design	7.4
Code File	FileIO.cs
Explanation	The module is a part of software decision hiding since it determines how the game will be saved and represented in a text file, and also how the file is parsed to reload a saved game file.

4 Uses Relationship



5 Changelog from Assignment 1 to Current

5.1 Board

- nothing

5.2 Piece

- nothing

5.3 Game1

- nothing

Added Module

- nothing

6 American Checkers Rule that is Implemented in the Game

6.1 Game play Rules

[1]

- Checkers is played by two players. Each player begins the game with 12 colored discs. (Typically, one set of pieces is black and the other red.)
- The board consists of 64 squares, alternating between 32 dark and 32 light squares. It is positioned so that each player has a light square on the right side corner closest to him or her.
- Each player places his or her pieces on the 12 dark squares closest to him or her.
- Black moves first. Players then alternate moves.
- Moves are allowed only on the dark squares, so pieces always move diagonally. Single pieces are always limited to forward moves (toward the opponent).
- A piece making a non-capturing move (not involving a jump) may move only one square.
- A piece making a capturing move (a jump) leaps over one of the opponent's pieces, landing in a straight diagonal line on the other side. Only one piece may be captured in a single jump; however, multiple jumps are allowed on a single turn.
- When a piece is captured, it is removed from the board.
- If a player is able to make a capture, there is no option – the jump must be made. If more than one capture is available, the player is free to choose whichever he or she prefers.
- When a piece reaches the furthest row from the player who controls that piece, it is crowned and becomes a king. One of the pieces which had been captured is placed on top of the king so that it is twice as high as a single piece.
- Kings are limited to moving diagonally, but may move both forward and backward. (Remember that single pieces, i.e. non-kings, are always limited to forward moves.)
- Kings may combine jumps in several directions – forward and backward – on the same turn. Single pieces may shift direction diagonally during a multiple capture turn, but must always jump forward (toward the opponent).
- A player wins the game when the opponent cannot make a move. In most cases, this is because all of the opponent's pieces have been captured, but it could also be because all of his pieces are blocked in.

[1] Arneson, Erik. Ḧow To Play Checkers (Using Standard U.S. Rules).Äbout.com Board / Card Games. About.com, n.d. Web. 7 Apr. 2014. ¡http://boardgames.about.com/cs/checkersdraughts/ht/play$_{c}heckers.htm >$

7 Module Design (MIS and MID)

7.1 Piece Module

7.1.1 Interface

Types

typeState	enumerate if the piece is normal or king
player	enumerate if piece owned by Black or White
validMovementStruct	structure that holds the valid movements

Constants

None

Access Programs

getType() : typeState	Retrieves the piece's current type.
setType(newType : typeState)	Changes the piece's type.
getOwner() : player	Says who owns the piece.
getValidMovements()	Retrieves the movements that a piece can make.
setValidMovements (direction : validMoveDirection, col : int, row : int)	Assigns the valid movements for a piece

7.1.2 Implementation

Variables

pieceType : typeState	holds current piece type
owner : player	holds information of the piece's owner
validMovementArray : validMovementsStruct	holds all valid movements for a piece

Access Programs

getType() : typeState

Inputs	None
Updates	None
Outputs	pieceType
Description	Returns the current type of the piece.

setType(newType : typeState)

Inputs	newType
Updates	None
Outputs	pieceType
Description	Changes the type of piece to the type given.

getOwner() : player

Inputs	None
Updates	None
Outputs	owner
Description	Returns which player the piece is owned by.

getValidMovements() : validMOvementsStruct[]

Inputs	None
Updates	None
Outputs	validMovementArray
Description	Returns an array of the valid movements for a particular piece.

setValidMovements(direction : validMoveDirection, col : int, row : int)

Inputs	direction, col, row
Updates	None
Outputs	validMovementArray
Description	Sets the places that are valid for a specific piece to move.

7.2 Board Module

7.2.1 Interface

Types

None

Constants

None

Access Programs

setUpBoard()	Sets up board based on user input.
getPiece(col : int, row : int) : Piece	This method is used to get a a piece from the given (x, y) location. If the piece does exist, we pass it along to the caller.
getPiece(location : Vector2) : Piece	This method is used to get a a piece from the given Vector. If the piece does exist, we pass it along to the caller.
getPieceArray() : Piece[]	Returns an array of all pieces that are currently on the board.
placePiece(col : int, row : int, piece : Piece)	Places the piece on the board while checking if the placement is legal (in terms of checkers).
movePiece(fromCol : int, fromRow : int, toCol : int, toRow : int)	Moves the piece from starting to end positions while checking if the movement is valid (in terms of checkers).
movePiece(originalLocation : Vector2, newLocation : Vector2)	Moves the piece from starting to end positions while checking if the movement is valid (in terms of checkers).
removePiece(column int, row int)	This method removes a piece off the board and will throw an exception if there is no piece at the given location.
clear()	Removes all pieces from the board.

7.2.2 Implementation

Types

None

Constants

None

Variables

pieceArray : array	Contains all the Piece objects currently on the board in an array.
numWhitePieces : int	Holds the number of white pieces on the board as an integer.
numBlackPieces : int	Holds the number of black pieces on the board as an integer.

Access Programs

setUpBoard(input : string)

Inputs	input
Outputs	pieceArray, numWhitePieces, numBlackPieces
Updates	None
Description	Parses input to be interpreted as Piece locations. Place Piece on correct Piece location using the PlacePiece() access program. numWhitePieces' = numWhitePieces + c and numBlackPieces' = numBlackPieces + d where c and d are between 0 and 12. pieceArray' = pieceArray with c + d more PieceObjects.

getPiece(col : int, row : int) : Piece

Inputs	col, row
Outputs	piece
Updates	None
Description	Returns the piece currently at the location specified.

getPiece(Location : Vector2) : Piece

Inputs	Location
Outputs	piece
Updates	None
Description	Returns the piece currently at the location specified.

placePiece(col : int, row : int, piece : Piece)

Inputs	col, row, piece
Outputs	None
Updates	pieceArray, numWhitePieces, numBlackPieces
Description	If piece placement is valid, it will put it there in the data structure. Either numWhitePieces' = numWhitePieces + 1 or numBlackPieces' = numWhitePieces + 1. pieceArray' = pieceArray with one more Piece object.

movePiece(fromCol : int, fromRow : int, toCol : int, toRow : int)

Inputs	fromCol, fromRow, toCol, toRow
Outputs	None
Updates	pieceArray
Description	Moves piece at said location to the location specified.

movePiece(originalLocation : Vector2, newLocation : Vector2)

Inputs	originalLocation, newLocation
Outputs	None
Updates	pieceArray
Description	Moves piece at said location to the location specified.

removePiece(column : int, row : int)

Inputs	column, row
Outputs	None
Updates	pieceArray
Description	Removes the piece at the specified board location given by row and column.

clear()

Inputs	None
Outputs	pieceArray, numWhitePieces, numBlackPieces
Updates	None
Description	Clears the board of all pieces. pieceArray' = Array of null objects, numWhitePieces' = 0 or numBlackPieces' = 0.

7.3 Game1 Module

7.3.1 Interface

Types

state	enumerate if the game is in Menu, Setup, Playing, or Load

Constants

None

Access Programs

Update()	Allows the game to run logic such as switching state, updating the game, and gathering input.
Draw()	Draws the correct graphics on screen depending on the state.
takeInput()	Takes user input for setting up a board.
setValidMovements(Board : board)	Sets the valid movement for every piece.
setValidMovements(Board : board, X : int, Y : int)	Sets the valid movement for an individual piece at a given (X,Y) location on the board.

7.3.2 Implementation

Variables

currentState : state	holds information of the current state.
keyState : KeyboardState	holds information about the state of the keyboard.
input : string	holds board setup from user
board : Board	
pieceList : List	Holds information of where to graphically place each piece.

Access Programs

Update()

Inputs	None
Updates	currentState
Outputs	None
Description	Changes the state based on keyboard press or mouse presses on graphical buttons.

Draw()

Inputs	board
Updates	pieceList
Outputs	None
Description	Draws the buttons, board tiles and pieces in proper place on the screen. The piece locations are stored in pieceList. And we just loop through the graphics objects to draw them each frame.

takeInput()

Inputs	input
Updates	None
Outputs	board
Description	Takes user input and sends it to the board using board.SetUpBoard().

setValidMovements(Board : board)

Inputs	Board
Updates	Board
Outputs	None
Description	Sets the valid movements for every Piece. Default constructor to set them for the entire board. The safe locations to move to are stored within the Pieces. The valid locations are assigned in the order: Top Left -¿ Top Right -¿ Bottom Right -¿ Bottom Left. The default constructor allows for calling the function with no paramaters to set up the entire board. This function loops through each piece element and calls upon setValidMovements for each individual piece to calculate the valid movements of the entire board.

setValidMovements(Board : board, X : int, Y : int)

Inputs	Board, X , Y
Updates	Board
Outputs	None
Description	Sets the valid movements for an individual Piece. The valid movements are a combination of an x direction and a y direction. initialized to a flag of an unreachable location. Game logic and checkers rule and game logic are contained within this function to judge where a piece can be placed.

7.4 FileIO Module

7.4.1 Interface

Types

None

Constants

None

Access Programs

Save(board : Board, turn : PLAYER) : void	Saves the current board state
Load(board : Board) : String	Loads the board with a previous board state

7.4.2 Implementation

Variables

path : String	holds the path to the location of where the save file is to be placed

Access Programs

Save(Board board, PLAYER turn)

Inputs	board, turn, path
Updates	None
Outputs	None
Description	Saves the current game session, it parses the current board array state into a text file along with the information of which player's turn to move.

Load(Board : Board)

Inputs	board
Updates	None
Outputs	None
Description	Loads a new game with a previous save file. This will return an exception if there is no load file present.

7.5 Input Module

7.5.1 Interface

Types

Mouse	enumeration of mouse button states
Keys	enumerates keyboard buttons

Constants

None

Access Programs

GetState() : Mouse	Gets if mouse button is pressed.
IsKeyDown(key : Keys) : bool	Checks if the key is pressed.

7.5.2 Implementation

Variables

mouseState : Mouse	Holds if mouse is pressed.
mouseClickedPiece	Holds the graphical object the mouse is clicking on.
mousePos	Stores current mouse position.

Access Programs

GetState()

Inputs	None
Updates	None
Outputs	mouseState

IsKeyDown(key : Keys)

Inputs	None
Updates	None
Outputs	None

8 Internal Evaluation

Evidently, our design makes use of several essential design principles for simplicity and efficiency. Our design makes use of a hierarchical structure to make the system easier to build and test. We made use of abstraction by having the program abstract the whole game, the game abstracts the board, the board abstracts the pieces, etc. so we could start assigning different parts to the group right away. We also used the idea of information hiding to make things that are likely to change private. This maximized efficiency and allowed us to get our design done very quickly. Our design makes use of the high cohesion and low coupling principles as much as possible to make sure our modules are meaningful when standing alone. We made a variety of decisions that improve the design, the following are examples.The setting of valid moves is done incredibaly efficiently. We used integer comparisons in the logic section and assign the valid moves to every piece which is much faster than if every piece's moves were stored on the piece array. Special types in the design are enumerations so they can be converted quickly. Finally, the Struct used combines all the information needed in one place and is very intuitive. Overall we made conscious decisionsin the design to ensure that the principles of software design were followed closely.