Checkers Design Document

2me3

Contents

1 Introduction **1**

2 Module Guide **2**
- 2.1 Hardware Hiding Module . . . 2
  - 2.1.1 Input Module . . . 2
- 2.2 Behaviour Hiding Module . . . 2
  - 2.2.1 Piece Module . . . 2
- 2.3 Software Decision Hiding Module . . . 2
  - 2.3.1 Board Module . . . 2
  - 2.3.2 Game1 Module . . . 3

3 Uses Relationship **3**

4 Module Design (MIS and MID) **4**
- 4.1 Piece Module . . . 4
  - 4.1.1 Interface . . . 4
  - 4.1.2 Implementation . . . 4
- 4.2 Board Module . . . 5
  - 4.2.1 Interface . . . 5
  - 4.2.2 Implementation . . . 6
- 4.3 Game1 Module . . . 7
  - 4.3.1 Interface . . . 7
  - 4.3.2 Implementation . . . 7
- 4.4 Input Module . . . 8
  - 4.4.1 Interface . . . 8
  - 4.4.2 Implementation . . . 8

5 Internal Evaluation **8**

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 Module Guide

2.1 Hardware Hiding Module

2.1.1 Input Module

Type	Hardware Module
Secret	This module translates mouse clicks and keyboard presses to be used by the rest of the software.
Responsibilities	This module will take mouse and keyboard input and convert it to software usable states.
Uses	None
Design	4.4
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.

2.2 Behaviour Hiding Module

2.2.1 Piece Module

Type	Software Module
Secret	This module hides and separates specific piece information.
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	4.1
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.

2.3 Software Decision Hiding Module

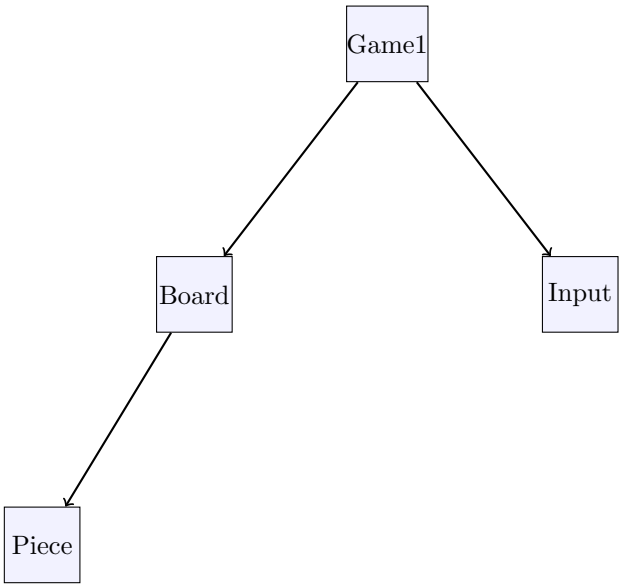
2.3.1 Board Module

Type	Software Module
Secret	This module serves to hide the secret of how the board is defined internally.
Responsibilities	This module is responsible for holding the necessary components and attributes to setup the board and describe piece locations.
Uses	2.2.1
Design	4.2
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.

2.3.2 Game1 Module

Type	Software Module
Secret	This module hides how the graphics are displayed and how we switch between states of the game.
Responsibilities	This module will be the responsible for the initial execution of the game, this class connects and launches critical components together.
Uses	2.3.1, 2.2.1, 2.1.1
Design	4.3
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 Uses Relationship



4 Module Design (MIS and MID)

4.1 Piece Module

4.1.1 Interface

Types

typeState	enumerate if the piece is normal or king
player	enumerate if piece owned by Black or White

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.

4.1.2 Implementation

Variables

pieceType : typeState	holds current piece type
owner : player	holds information of the piece's owner

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.

4.2 Board Module

4.2.1 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 determine if a piece exists on a square of the board. If the piece does exist, we pass it along to the caller.
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).
clear()	Removes all pieces from the board.

4.2.2 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.

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	None
Outputs	None
Updates	None
Description	Moves piece at said location to the location specified.

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.

4.3 Game1 Module

4.3.1 Interface

Types

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

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.

4.3.2 Implementation

Variables

currentState : state	holds information of the current state
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().

4.4 Input Module

4.4.1 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.

4.4.2 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

5 Internal Evaluation

This document will be an evaluation of our design decisions. 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 abstract the board, the board abstract the pieces, etc. so we could start assigning different parts to the group right away. For this reason, 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.