**INTRODUCTION**

This software design document is designed to detail the implementation and deliverables of the checker board game software application, interaction between the user and the system and internal components that interact with each other.

Our design is a software application that allows a user to initially automatically set up checkers on the checkerboard and also be able to choose positions to place initial checker layout by using a graphical interface. The program traces the user’s input and determine the position’s legality.

This document includes a description of each application’s architecture design and their associated interface, design pattern, decomposition of each modules and uses MIS and MID to describe each of the classes in the program. A description of each of the modules, syntax and semantics of each public method in the class will be included. This design document includes the internal review and evaluation that focus on how well our application designed and the design motivation why we made it.

The intended audience of this software application includes computer and software students at university level or higher and any person who is interested in software design.

**Definitions, acronyms, and abbreviations**

MVC: Model View Controller

Model - The software logic: Business rules and Application data

View - The user interface

Controller - Communication between user and model

MID : Module Internal Design

MIS : Module Interface Specification

**References**

Fundamentals of Software Engineering

Introduction

1.1 Purpose of this document

This software design document is designed to detail the implementation and deliverables of Checker board game software application, interaction between the user and the system and internal components interact with each other. This is the project design stage one.

1.2 Scope of the development project

Our design is a software application that allows a user to automatically set up the initial standard positions on the checkerboard and also be able to choose customized positions to place initial checker layout by using a graphical interface. The program trace the user’s input and determine if the position’s legality.

1.3 Overview of Document

This document includes description of each application’s architecture design and their associated interfaces, design pattern, decomposition of each modules and use MIS and MID to describe each of the classes in the program. Also, a description of each of the modules, syntax, semantic of each public method in the class will comprehensively included. This design document only represent the stage one of the “Checker” game application. This design document includes the internal view and evaluation that focus on how well our application designed and the design motivation why we made it.

1.4 Definitions, acronyms, and abbreviations

u MVC: Model View Controller

Model - The software logic: Business rules and Application data

View - The user interface

Controller - Communication between user and model

u MID : Module Internal Design

u MIS : Module Interface Specification

1.5 Intended audience

The intended audience of this software application includes computer and software students at university level or higher and any person who is interesting in software design.

2. SYSTEM OVERVIEW

The checkers game program is being tackled in stages. First we identify the main functions of the program. This helps to identify how to design the solution and which design pattern to follow.

The two main functions of the checkers program is to have a standard opening position with 12 pieces for each side or custom board with missing pieces from both sides with initial positions described by the user and the number of pieces as well. The entire program must be visually represented with graphics (input and output).

The design pattern used to make this application is Model-View-Controller (MVC) design pattern. The model category of this structure will hold the business logic, controller will handle user inputs and view will display the graphical representation of the checkers game interaction.

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

The checkers game is designed with the MVC structure in mind to differentiate between three varying aspects. The MVC is a pattern, set up so there is a clear distinction between the user inputs, the graphical output and the logic. The interaction between the three defines how the our program will behave on an abstract level. The model category will hold the set of class with all the business logic to properly operate this program. The model then transfers this logic to the set of classes under the view category which takes this information and output this information in a graphical format. The controller category has a set of classes that interacts with the user. The set inputs get transferred from controller to model and then out to view for the final result.

The stimuli for this project will be the (standard button, manual button, position, piece colour). These will be all the possible inputs that the user can input. These variables also affect how the program will look and what the program outputs. The set of outputs will the piece and position on the board as well as an error message regarding exceeding 12 pieces. These will be the controlled variables.

3 .2 Decomposition

The checkers game is composed of six classes. It has been setup so that each class can be clearly categorized under each of the MVC sections. The *Mainframe* and *AIRobot* class fall under the Model section while *Checkpiece,GameStatus* and *CheckBoardBlock* fall under the View section. The *MenuItems* class handles inputs from users, meaning it falls under Controller.

*Mainframe:*

The mainframe class models the chess game using arrays. Arrays are used to represent physical objects of the game such as the checkers board and the pieces of the board. The checkers board was simulated with the use of a 2D array. The checkers pieces were represented by two separate, single dimension arrays. The mainframe is the class that handles the logic for checkers. It is responsible for simulating the the checkers game in a console version

AIRobot:

This class contains the rules for the AI, if the player chooses to play against the computer. The robotMove method, using other methods within the module will decided the move the ai will make. The changePiecePosition method will change the positions of the pieces when a move has been decided.

Checkpiece:

The Checkpiece class handles the graphics for creating the individual pieces. The state of the individual pieces is graphically represented by this class. The logic from the mainframe class is fed to the checkpiece class, which in turn displays the state of the piece on the monitor

CheckBoardBlock:

Similar to the Checkpiece class, it too handles graphics. This class draws individual squares of the checkers board. Each individual block represent a single location in the 2D array simulating the checkers board.

GameStatus:

The GameStatus class handles the status (state) of a checkers game.

Menuitems:

This class is responsible for providing menuitems that GUI based (i.e buttons, drop-down menus, etc)

**Users relationship**

The uses relationship is viewed as hierarchical structure that explore the relationships between modules. It represents the dependencies modules have on each other and define how the services are implemented through interfaces. As mentioned, “uses” should be a hierarchy that makes software easier to understand, build and test. And each level defines an abstract machine for the next level, known as “level of abstraction”. For instance, let S be a set of modules, S = { M1, M2,.....Mn}, if Mi and Mj are in S, k is the maximum level of all nodes of Mj such that Mi is relational to Mj, then there must exists level k+1 in Mi.

In the second version of checkers contains the modules, “MainFrame”, “GameStatus”, “ChessPiece”, “ChessBoardBlock”, “DragButton”, and “MenuItems”. All of these modules are implemented in “MainFrame”.“Jframe” creates a top level container for the rest of the interfaces’ graphical component. A window for a java application is created for the game. “JLayeredPane”, which is a “Jframe” layer object has split into two layers. Top layer “Jpanel boardBg”, which used to represent the checker game’s board and the bottom layer “Jpanel body” is used to describe the checker pieces. Some examples of the description of the methods under “MainFrame” module are as follows:

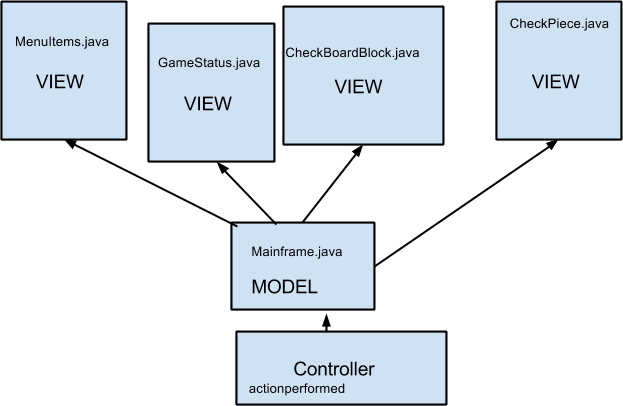
* MainFrame() generates menu bar, and set activates actionlisteners.
* InitPieces() initializes the collection of checker pieces (black and white ).
* Initialize() initializes the chess board; all pieces are on the correct starting position. It sets up the layer object onto “Jpanel boardBg” by using “1p =this.getLayeredPane;”. The board is defined by Jpanel with a 8 by 8 GridLayout.
* clearBoard() Removes all pieces from board and clear template data.
* setBoardEnabled() This enables mouse click actions.
* initchessBoardManually() ,initChessBoardAutomaticly() These two method allow user to set the checker pieces manually on the board or automatically at the default positions.
* putPieceOnBoard() put the pieces on the board when user select manually generated pieces on board.

The “MainFrame” module works as the skeleton of the game. It provides the game structure, initializes the board, checkers, menu bar, puts pieces on the board, allows pieces to move, determines the validity of the moves, and displays the result when a game is over. The “ChessPiece” module works on the specifications of the checker pieces (eg. Color). The “GameStatus” record the state of the game and updates the game. The “ChessBoard” mainly works on the graphical color of the board. “DragButton” provides the behavior of the mouse, when dragging a valid piece. Lastly, “MenuItems” initializes menu.

In the third iteration of our checkers game, artificial intelligence has been implemented. The MainFrame refers to the class AIRobot when it is required for the computer to make a move. Whenever it is required for the computer to make a move, MainFrame will ask AIRobot to make a move, through the robotMove method.

The complex system is divided into modules such that each module is highly cohesive because each component in the module is closely related to one another. Modules have very few interactions with one another and that satisfies low coupling.

Figure. 1

Below are Module Interface Specifications the individual module of this application

**MIS 1**

**Module: MenuItems.java**

Constants: public static final int itemsCount [], public static final string items []

Constructor: Static

* Initializes menu-items

**MIS 2**

**Module: Mainframe.java**

**Public class MainFrame implements ActionListener and JPanel**

Constructor Mainframe:

* Initializes menu-items

Public void initPieces()

* Initialize both white and block pieces

Public void Initialize()

* Board initialized and pieces on initial positions

Public void initCheckBoardAutomaticly()

* Initialize all pieces to default opening positions

Public void setBoardEnabled(boolean flag)

* make board clickable

public void initCheckBoardManually()

* let user define positions for pieces manually

public void putPieceOnBoard(CheckBoardBlock b, String player)

* triggered during manual setup, this puts one piece in the board

public void reFormatPieceLayer()

* Refresh the pieces according to the piece record

public int getBlackCountInVector()

* get black piece on the board

public int getWhiteCountInVector()

* get white piece on the board

public void print()

* prints checker piece for testing

public void actionPerformed(ActionEvent e)

* detects a clickable event and performs the appropriate operation

publc void mouseDragged(MouseEvent e)

* changes the position of the piece according to the mouse drag direction and destination

public CheckBoardBlock findMovedBlock(int x, int y)

* Returns the block on the board of the piece that was just moved

public void changePosition(ChessPiece p1, String newP)

* Move the a piece to a new location on the board

public void clearCapturedPiece(ChessPiece p1, String newP)

* Clears the piece that was captured

public void disablePlayer(String player)

* disables a player from moving

public void enablePlayer(String player)

* enables a player to move

public boolean isPeace()

* Checks to see if the game is at Peace

public String win()

* Checks to see if the game is over and returns the winner of the game

public void startGame()

* Starts the game

**MIS 3**

**Module: CheckBoardBlock**

**Types: Strings, Graphics**

public CheckBoardBlock(String color, String position)

* position and colour values are initialized by this constructor

public void paint(Graphics g)

* paints each block either red or white

**MIS 4**

**Module: CheckPiece**

public CheckPiece(String player, String position)

* Position and colour values are initialized

public void paintComponent(Graphics g)

* Paints the individual pieces

public void paintBorder(Graphics g)

* Paints the border

**MIS 5**

**Module: GameStatus**

**Types: Strings**

public String getStatus()

* returns the status of the game, whatever the may be

public void setStatusInit()

* sets the status to initialize

public void setStatusExec()

* sets the status to execute

public void setStatusOver()

* sets the status to over

public GameStatus()

* sets the game status to initialize

MIS 6

Module: AIRobot

**Interface:**

public static HashMap <String, CheckPiece> getPlayerPieces(MainFrame m, HashMap<String, CheckPiece> ps, String flag)

* gets all the pieces of the flag player

public static void robotMove(MainFrame m)

* The robot will control the movement of a piece, once this method is called
* contains a sequence on decisions that determine the appropriate for the AI to make
* done by analyzing first which pieces can be captured and which pieces are in danger and picks the appropriate move depending on the prioritization of these decisions.

public static HashMap<String, CheckPiece> getMovablePieces(MainFrame m, HashMap<String, CheckPiece> ps)

* Analyzes which pieces are able to move depending on the given number and position of pieces

public static HashMap<String, CheckPiece> getPiecesCanCapture(MainFrame m, HashMap<String, CheckPiece> ps)

* Analyzes which pieces are able to move and can also capture opponent’s pieces

public static HashMap<String, CheckPiece> getPiecesWillBeCaptured(MainFrame m, HashMap<String, CheckPiece> ps)

* Captures pieces of the opposite player

public static CheckPiece getPieceCanBeHelp(MainFrame m, CheckPiece tp)

* Checks to see if a piece in danger can be helped
* if none of the pieces can help then it returns null or else returns position of the piece

public static CheckPiece getHelpPiece(MainFrame m, CheckPiece p, CheckPiece helpP)

* gets the piece that can help the piece in danger

public static HashMap<String, CheckPiece> getPiecesCanBeKing(MainFrame m, HashMap<String, CheckPiece> ps)

* gets the piece that will be a king

public static void changePiecePosition(MainFrame m, CheckPiece p1, String newP)

* moves piece to new position

**Implementation**

public static void robotMove(MainFrame m)

* The position and the pieces are stored in a hash map
* It later runs through a function called getMovablePieces which returns positions of the movable pieces
* It goes through possible scenarios for AI to make decisions based on the current positions of the board
  + If robot can capture a piece, take them first
  + If there are pieces will be captured by player, take them first
  + If there are pieces will be a King, take them
* predicts the next best move
  + If there a position, the piece moved there and able to capture a opposite player's piece,Record this position
  + Help a piece which will be captured
  + If a piece can be a King, move it

public static HashMap<String, CheckPiece> getMovablePieces(MainFrame m, HashMap<String, CheckPiece> ps)

* get the moveable pieces by running a loop that with a boolean indicator that determines which piece is movable

public static HashMap<String, CheckPiece> getPiecesCanCapture(MainFrame m, HashMap<String, CheckPiece> ps)

* it has two nested for loops to check for pieces of both sides (AI side and human side) and runs a boolean indicator that determines which piece is in a position of capture with respect to the AI pieces

public static HashMap<String, CheckPiece> getPiecesWillBeCaptured(MainFrame m, HashMap<String, CheckPiece> ps)

* this one has a single loop to which one iterates for one side and performs the capture

**Traceability Table**

|  |  |
| --- | --- |
| Users are able to start a game  from original starting positions | Class: MenuItem  Activate when user click Initial  ChessBoard automaticaly  Class: MainFrame  Calls 2 method  initPieces()  Which sets Black, White and  Empty pieces.  initchessBoardAutomaticly()  Which cleared the board and put  pieces on board. |
| Users are able to set the starting positions of the pieces manually | Class: MenuItem  Activates when user clicks Initial chessBoard manually from the menu    Class: MainFrame  initchessBoardManually()  Changes the status of the game to GAME\_STATUS\_EDIT, which enables mouse interaction to place the pieces. (left click to place piece, right click to set/cancel king)  Activate and set visible the button “switcher” to switch the player  Dragging an existing piece with the mouse will change its position to a valid location upon the mouse release  Activate and set visible the button “confirm” which confirms the placement of the pieces |
| Users are able to save one game | Class: MainFrame  saveGame()  creates the file (save.txt) then store  the current player, position of all the  pieces and the game status |
| Users are able to load the saved  game | Class: MainFrame  loadGame()  load the current player, position of all  the pieces and the game status |
| Make moves Class: | Class: MainFrame  changePosition(ChessPiece p1,  String newP) – move piece(pi) to new  position(newP)  clearCapturedPiece(ChessPiece p1,  String newP) – Clear the piece if was  captured |
| Players can choose to play against an AI | Class: MenuItems  This can be activated by clicking “play with black AIi” or “play with white AI” |
| Players can choose to play against a second player | Class: MenuItems  This can be activated by clicking “play with human” |
| AI can play against the palyer | Class: AIRobot  The robotMove method decided where the AI will move, and will call the changePiecePosition method to move it. |

3.3. Design Rationale (Evaluation)

The MVC pattern was the ideal decision for this application as it yielded itself perfectly for break down of the checkers game. The rules and logic of the game can be placed in the model category while the graphics could fall under *view* and user inputs under the *controller* category. The benefit of this setup allow for easier future manipulation of the source code. By separating into three categories, one can effectively manage the internal components of each class while worrying less about the interaction between them. This design pattern is best practiced for graphical applications. MVC was the natural choice for this application.

The design could have been improved by merging *GameStatus.java* with the view with *Mainframe.java* due to the simplicity of the program but this has its own disadvantages as well. The distinction between the model and view is necessary for accommodating future changes. By separating the two, the purpose of each class becomes clear and allows for changes as well.

**Public Interface**

Menu

The object menuBar in the MainFrame module acts as the menu bar of the game. It uses the java package Menubar to implement it. The menu bar is populated with the items in the class MenuItems. The method actionPerformed detects which menu item has been clicked, and will perform the corresponding action.

Set Pieces

Automatically:

The pieces will be set automatically when the menu item "Initial Chessboard automatically" is pressed. This process is done by the initChessBoardAutomaticly method of the MainFrame. This method calls on the methods initPieces and clearBoard. initPieces will reset the locations of the pieces are stored. The clearBoard method removes all the current pieces from the board. This is in case there is already a game in progress when the player decides to create a new game.

Manually:

The pieces will be set manually when the menu item "Initial Chessboard manually" is pressed. The initChessBoardManually method is called, and once again, this method will call clearBoard to remove any pieces that were on the board before the menu button press. The state is changed to edit. setBoardEnabled will run and allow the board to be clickable, so the positions can be set. A button labeled “Switch Player” can be pressed, and will change the colour of the pieces being set. The mousePressed, mouseDragged, and mouseReleased method will change the location of a piece when it is dragged using the mouse to a new valid location on the board. A left click on a checker piece already placed on the board will toggle it to a king or back to a regular piece. When the “confirm button is pressed, the confirm variable will tell the game to start, and the state is set to play game. Hitting the confirm button will disable switcher, setBoardEnabled, and confirm.

Vs. Human or AI:

Players can choose whether they would like to play against an AI or a human in the menu, by clicking “play with human”, “play with black AI”, or “play with white AI”.

Movement

A piece can be moved when the game is the the proper state, and only a valid piece can be moved. The mousePressed method in the module MainFrame is in charge of moving pieces. Once a valid piece is pressed and held on to, the getMovablePosition method will run, and will find valid spots for target piece to move according to checkers rules. This method will also check if a piece has reached the end of their opponent’s side. If this is true, the piece will be assigned the king status, and will be subjected to different rules when moved, according to checkers rules. The letter “k” will be drawn on pieces as an indicator. When a value piece is clicked, the target will be drawn with a blue outline, and the valid move blocks will be outlined green. Methods mouseDragged and mouseReleased will then record the location of the mouse after a piece is dragged and released, for this location should be the location the player wants the piece to move. If the move is valid, the exchangeActivePlayer method will switch to the other player’s turn. The positions of the pieces will update.

Wins

During gameplay, every time the mouse is released and a piece is moved, the isPeace method will check if there is a winner in the game. The game will also check if there is a winner after the board has been manually set up, for the players could fail to put any piece of a specific colour on the board, which will result in an automatic win for one player. isPeace will modify the win and winner variables if there are no pieces of a single colour left. The game will then print “Winner is” followed by the winning colour.

Saving

Once the “save” button in the menu is pressed, the game will write a text file in the directory named save.txt. It will contain the current state of the game, and which player’s turn is active. The positions and status of the pieces will also be recorded.

Loading

Once the “load” button in the menu is pressed, the game will try and read a file in the directory named save.txt. It will modify the array containing the position of the pieces accordingly. It will then set up the board for playing, such as enabling all the actionlisteners, setting the states.

**Private Interface**

Module: GameStatus

It contains the variable “status”. This records the state of the game. When status is set as:

GAME\_STATUS\_INIT: the game has just been opened, and the game is in the initial state.

GAME\_STATUS\_EDIT: the game is allowing the players to manually set up the location of the checker pieces

GAME\_STATUS\_EXEC: the players can now play the game of checkers.

GAME\_STATUS\_OVER: the game has ended. This state is similar to the initial state.

Module: MainFrame

Here is the list of hidden objects in the main module.

Jpanel boardBg: Background layer of the checkerboard

JLayeredPane lp: frame layering of objects

JPanel body: layer of the checker pieces

GridLayout gridLayout: Layout of the checkerboard blocks

ChessPiece blackPieces: object containing the locations of black pieces, and will draw them as well

ChessPiece whitePieces: object containing the locations of white pieces, and will draw them as well

ChessPiece spacePieces: object containing the locations of the empty valid spaces on the board

Module: AIRobot

Hashmap tmpPiece: This is the object containing the current movable pieces on the AI side.

ChessPiece tp:

Checkers Test Documentation

Introduction

This document is the Test Plan for Phase II of the Checkers Project. It describes the testing strategy and approach used to assure the quality of the software, and reports the results.

Old Features from previous version:

1. Set up a Checkers game either by choosing the standard setup position, or by graphically placing pieces on the game board, as long as their positions are valid
2. Load unfinished game from a save file
3. Save unfinished game to a save file
4. Players can move the pieces as long as the moves are legal (legal moves are highlighted)
5. Printing out the state space of the current game to the console (this is for testing)
6. Status bar displaying players' turns

New features in current version:

1. Added AI component

Test Approach

As the majority of the code runs within a GUI, the testing approach will be to manually test each component of the GUI as the program is running to ensure that the software works as specified. The only class to be unit-tested is the GameStatus Class, as this class maintains the game state and is therefore appropriate for unit testing. Once the GUI has been tested, the game logic itself will also be tested to ensure that no rules of Checkers can be violated.

Unit Test Results

Class:  GameStatus

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method under Test | Expected Result | Actual Result | Failure(?) |  |
| getStatus() | Object is not null | Object is not null | No |  |
| setStatusInit() | getStatus() == "INIT" | getStatus() == "INIT" | No |  |
| setStatusEdit() | getStatus() == "EDIT" | getStatus() == "EDIT" | No |  |
| setStatusExec() | getStatus() == "EXEC" | getStatus() == "EXEC" | No |  |
| setStatusOver() | getStatus() == "OVER" | getStatus() == "OVER" | No |  |

Class: MainFrame

|  |  |  |  |
| --- | --- | --- | --- |
| Method under Test | Expected Result | Actual Result | Failure(?) |
| initPieces() | Populate blackPieces[], whitePieces[], spacePieces[] | Populated successfully | No |
| initialize() | Place all the pieces in the board | All pieces placed in the correct position | No |
| clearBoard() | All pieces on the board will be deleted | All pieces cleared | No |
| initChessBoardAutomaticly(\_) | Call initPieces and initialize() to set up the board | Board successfully set up | No |
| putPiecesOnBoard() | Place on piece on the board | One piece placed | No |
| reFormatPieceLayer() | Refresh the board, updating all the piece locations | All pieces refreshed | No |
| getBlackCountinVector() | 12 at setup | 12 | No |
| getWhiteCountinVector() | 12 at setup | 12 | No |

Bug/Problems Report:

All unit tests passed. There does not appear to be a bug in the functionality of the GameStatus class.

GUI Test Results

1. Run Program

Expected Result: New empty checkers board is generated

Actual Result: New empty checkers board is generated

Comments:  Main program runs as expected. Board is missing column and row labels.

1. Button: File

Expected Result: Drop-down menu containing Save and Load buttons

Actual Result: Drop-down menu containing Save and Load buttons

Comments: File button works as specified.

1. Button: Start

Expected Result: Drop-down menu containing the Automatic Checkers Board Initialization button, the Manual Checkers Board Initialization button, the Print Checkers Board button, and the exit button

Actual Result: Drop-down menu containing all the required buttons

Comments: Label for start button should be capitalized i.e. "Start" not "start". Initialization and print button should say "Checkers" not "Chess", and "Initialize" not "Initial". Otherwise this works as specified.

4. Button: AI

Expected Result: Drop-down menu containing 3 radio button, play with Human (for 2 players), play with black AI, and play with white AI

Actual Result: Drop-down meny containing all required buttons

Comments: User is able to switch between the modes at anytime

5. Feature: Automatically setup a game of Checkers with the standard opening positions

Button: Initial ChessBoard automatically

Expected Result: The empty board should be populated with checkers pieces in the standard opening formation. The status bar should display the first player's turn.

Actual Result: The board is set up in the standard opening position. The status bar displays "white's turn". A new game is started.

Comments:  Works as specified. The button sets up a new game with the correct standard opening positions whenever it is pressed with white as the first player, regardless of the current state of the board. Initializing a new checkers board automatically does not affect the state of the Switch Player and Confirm buttons i.e. if they are active at the time a board is created automatically, they will remain active until deactivated by clicking the Confirm button. While active, the Switch Player:\_ button will change the colour it displays if clicked, but this has no further effect on the game being played.

6. Feature: Manually setup a game of Checkers in any legal configuration of the players' choosing

Buttons: Initial ChessBoard automatically , Switch Player:\_ , Confirm

7. Feature: prints out the status of the entire board along with all the checker pieces to the console.

Buttons: Print out ChessBoard

Mouse Actions: Left click, Right click

* 1. Initial ChessBoard manually

Expected Result: When pressed, the board should be set to an empty board and the Switch Player:\_ and Confirm buttons should become active, status bar should display "Please choose position to put the pieces. Right click to set/cancel King flag".

Actual Result: Works as specified. Board is set to an empty board and the Switch Player:\_ and Confirm buttons are activated regardless of the current state of the game. Status bar displays correct message.

Comments: Works as specified. The state of the Switch Player button is not modified by initializing the board manually i.e. if Switch Player:\_ is set to white, initializing a new board manually will not change it back to its original value (black) automatically, even if it just became active.

* 1. Switch Player: \_

Expected Result: Button displays the colour of the piece to be placed on the board. When pressed, the colour should change and the test to the right of the colon on the button should display the new colour accordingly.

Actual Result: The button switches the colour of the piece to be placed. The text on the button accurately reflects the colour of the piece being placed. By default, this is black.

Comments: The state of Switch Player (active/inactive, black/white) remains the same even if a new board is created. It is only deactivated by clicking the Confirm button. Otherwise works as specified.

* 1. Mouse: Left click

Expected Result: Create a checkers piece of the colour displayed on the Switch Player Button:\_ on any empty red tile. Left clicks on white tiles should do nothing. Left clicks on non-empty tiles should do not create a new piece, but if held should allow the piece being clicked to be moved to any empty red tile. The status bar should display the number of pieces of each colour on the board and update automatically with each successful click accordingly.

Actual Result: Left clicking red tiles creates a checkers piece of the colour displayed on the Switch Player:\_  button. Left clicking white tiles highlights the tile being clicked, but does not create a piece on that tile. Creating a piece on the "end" of the board for a particular colour automatically creates a King piece, designated by a "K" on the piece. Status bar accurately reflects the number of pieces of each colour on the board. If the number of black pieces is 12 and the Switch Player:\_ is set to black, left clicking a red tile does not create another black piece but updates the status bar to show: "Black pieces are maximum" underneath the piece count. If the number of white pieces is 12 and Switch Player:\_ is set to white, left clicking a red tile does not create another white piece but updates the status bar to show: "White pieces are maximum" underneath the piece count. If the number of both black and white pieces is 12, left clicking an empty red tile does not create another pieces, but updates the status bar to display "Cannot put anymore pieces on". If held, left clicking an existing piece lets it be moved to any empty red tile. Attempting to move a piece to a non-empty red tile or to any tile results in the piece being moved back to its original position.

Comments: Works as specified.

* 1. Mouse: Right click

Expected Result: Right clicking white tiles and empty red tiles should do nothing. Right clicking an existing piece should make it a King piece, designated by a "K" on the piece, or make it a normal piece if it is already a King piece.

Actual Result: Right clicking white tiles and empty red tiles do nothing. Right clicking pieces designates them as King pieces (the functionality of this will be tested later).  However, the status bar updates to display "Winner is:black" if a black piece is right clicked and "Winner is: white" is right clicked if there are only pieces of the that colour on the board.

Comments: The faulty status update is a major bug. If another piece of either colour is created, the status bar reverts back to the piece count, but the bug recurs again if a piece is right clicked and there are still no pieces of the opposing colour on the board. If the Confirm button is clicked while the status bar is incorrectly displaying the winner, the pieces can no longer be moved and the game ends without being played (this can only occur when there are only pieces of one colour on the board).

* 1. Confirm

Expected Result: Should start a game of Checkers with the custom setup from placing the pieces manually. Status bar should display the first player's turn.

Actual Result: Starts the game with the custom setup. Status bar displays "white turn". Both Switch Player:\_ and Confirm are deactivated.

Comments: When Confirm is clicked with pieces of just one colour on the board (without the aforementioned faulty status bug occurring), a game is started instead of choosing that colour as the winner. This is also a bug. When there are only white pieces on the board, the status bar displays white as the winner once any move has been made, but if there are only black pieces on the board, no further moves can be made since it is white's turn. This also occurs if confirm is clicked with an empty board. Another version of this bug occurs if a custom game is started with no possible moves for any of the pieces on the board.

1. Button: Print Board

Expected Result: Prints the current state of the game board to the console

Actual Result: Prints the current state of the game board to the console

Comments: This is to test that game state reflects what is on the board.

1. Feature: Save game to file

Button: Save

Expected Result: Saves the current state of the game to a text file

Actual Result: Saves the current player's turn, the state of board, and the current game status to a text file

Comments:  Works as specified. If more than one save is made, the older save is overwritten

1. Feature: Load game from file

Button: Load

Expected Result: Loads the most recently saved game from save file

Actual Result: Loads the most recently saved game from save file

Comments: Works as specified

Bug/Problems Report:

1. Spelling corrections on button labels (3)
2. Switch Player:\_  and Confirm buttons do not reset automatically if a new game is started while they are active (4), (5.I), (5.II)
3. Faulty status update bug (5.IV)
4. Confirm button bug (5.V)

Game Logic Testing

This section will document the results of the testing the rules of the game. This was tested by starting a number of instances of the Checkers application and playing through games (from both automatic and manual configurations, as well as save files).

|  |  |  |
| --- | --- | --- |
|  | Rule | Bug(?) |
| 1. | Black pieces can only be moved when status bar indicates black's turn | No |
| 2. | White pieces can only be moved when status bar indicates white's  turn | No |
| 3. | Black pieces cannot be moved when status bar indicates white's turn | No |
| 4. | White pieces can only be moved when status bar indicates black's turn | No |
| 5. | Only legal moves are highlighted | No |
| 6. | Pieces can only be moved to highlighted squares | No |
| 7. | Pieces that are jumped over by a piece of the opposingcolour are removed | No |
| 8. | A piece has to jump a piece of an opposing colour if it can on its turn; if two or more pieces can jump a piece of an opposing colour, one of them has to do so on its turn | Yes |
| 9. | A piece becomes a King piece once it has reached the other end of the board | No |
| 10. | King pieces can move both forward and backwards | No |
| 11. | Only one piece may be captured in a single jump although multiple jumps are allowed on a single turn if the board configuration allows it | Yes |

Bug/Problems Report

1. (8) This does not work as specified. Pieces are not forced to jump over opposing pieces as per the rules.

2. (9) This does not work as specified. Pieces are not able to make multiple jumps at all.