Friday, April 12th 2014

ASSIGNMENT III

Software Engineering

Software Design I 2AA4

Jadeesuan Anton	*1213386*
Connor Hallett	*1158083*
Spencer Lee	*1224941*
Nicolas Lelievre	*1203446*
Zhiting Qian	*1217485*

TABLE OF CONTENTS

PREFACE

The programing language we have chosen to use is Visual Basics (available within Visual Studios 2013 VB.net) since it allows an extensive customization of user interfaces. However, it does not follow the same implementation as other programming languages previously seen. Hence, modules, classes and methods are virtually defined and are composed of functions and subs. In this case, the application code was decomposed into five distinct modules, where each module performs a well-defined function.

Nearing the end of the project, it should be noted that there were some pre-existing bugs in the code that are yet to be resolved. However, their original solutions and pseudo implementations are well explained as part of the design documentation, as well as precise instances in which case the code failed in terms of strict testing methods in regards to each module. Both white and black boxes tests will demonstrate the original idea of the code as well as the actual implementations short comings.

Due to the lack of an included testing interface in Visual Basics (such as Junit in Java), testing instances for the public interface (Black Box) will be conducted visually. Internal implementations (White Box) will be completed through the use of labels, updated during runtime. These values will be examined, compared and documented on a pass or fail binary system. Each pass instance will signify a correct and expected calculation/output whereas a fail will count as an incorrect or an unexpected result. Other cases will be documented in which the output was not expected but performs a correct function.

Change Log

Interface Change Log

(April 3, 2014)	Added	**PlayVs.Ai Black Form**
(April 3, 2014)	Added	**PlayVs.Ai Red Form**
(April 9, 2014)	Reorganized	**Quit**, **Timer**, **Save**, **Load methods** (all of which changed into method instances of other modules instead of being distinct modules)
(April 11, 2014)	Removed	**High scores** (no previous function, simply removed interface)

Variables Change Log

Variable Name	Declaration Location	Date Changed	Change Type
C_compnext1	Local PictureBox	(April 8, 2014)	Added
C_compnext2	Local PictureBox	(April 8, 2014)	Added
C_compJump1	Local PictureBox	(April 8, 2014)	Added
C_compJump2	Local PictureBox	(April 8, 2014)	Added

Internal Implementation Change Log

(April 3, 2014)	Reorganized	**Start_Menu** (added method instances of load)
(April 4, 2014)	Reorganized	**Play** (added save as an instance of 2-player mode)

CLASS AND MODULE DESCRIPTION, INTERFACE AND IMPLEMENTATION

Form: Start_Menu *Reorganized*

Description

This succeeded the original form_load (now named custom_load). This is what the user initially sees when the game is loaded. It presents several options for gameplay including "Start Game", "Load Game", "Custom Game" and "Standard Game", all of which are available to the user.

Decomposition

The form's origin is largely due to the principles of modularity as this is a place that initializes one of the possible games modes and having them done in a single location rather than making states inside the actual play module would drastically increase maintainability and reduce the possibility of information leaking through internal states.

Interface Specification

- StartGame()
  - Makes Custom and Standard visible
- Custom()
  - Enters Custom Setup mode
- Standard()
  - Presents options for choice of AI colour to play against
- StandardAIAsRed() Added
  - Enters standard game setup as black player versus red computer
- StandardAIAsBlack() Added
  - Enters standard game setup as red player versus black computer
- Load()
  - Loads saved games

Uses Relationship

No other module call instances are present.

Variables

Variable Use	Variable Name	Variable Type	Declaration Location
Input	GamesetupC	Integer Array(32)	Global
Input	GamesetupS	Integer Array(32)	Global
Input	GamesetupL	Integer Array(32)	Global

Internal Implementation – Selection Logic

Output: Gamesetup(32) Type: Integer Array(32)

Logic: Input	Output Action
Custom	Initialize GameSetupC
Standard	Initialize GameSetupS
StandardAIAsRed	Initialize GameSetupS
StandardAIAsBlack	Initialize GameSetupS
Load	Initialize GameSetupL

Internal Implementation – Parameter setup

Output: Gamesetup(32) Type: Integer Array(32)

Logic: Input	Output Action
1	Black
2	Red
3	Black King
4	Red King

Internal Implementation – LoadGame() Added

The implementation of this module is a simple read file operation as modelled by the following pseudo code:

```
Dim reader as print writer
read(../SavedGame.txt)
if reader.empty = true
     msgbox (“No Saved Game”)
end if
gamesetupC(0 to 32) = 0
gamesetupS(0 to 32) = 0
gamesetupL(0 to 32) = reader
```

Testing – Public Interface

Input	Expected output	Pass Instance
Click StartGame	Option buttons show up	Pass (1/1)
Click LoadGame	Play form opens to last game	Pass (1/1)
Click Standard	Play form shows up with standard setup (2 players)	Pass (1/1)
Click StandardAsRedAI	Play form with AI as red shows up	Pass (1/1)
Click StandardAsBlackAI	Play form with AI as black shows up	Pass (1/1)
Click Custom	Play form with custom setup shows up	Pass (1/1)

Testing – Private Implementation

Variable	Variable Changed	Variable Expected	Pass Instance
Click StartGame	GameSetup	N/A	Pass (31/31)
Click LoadGame	GameSetup	Gamesetup in text file	Empty: Pass (31/31) 1 Piece: Pass(31/31) Standard: Pass (31/31)
Click Standard	GameSetup	Gamesetup (0-11,12-19,20-31) = 0,1,2	Pass (31/31)

Form: Custom *No Changes*

Description

The Custom form is used to arrange a customized piece layout. The user will be able to select the positioning of each piece and place a total of 12 of each color (including King pieces). The setup will be complete once the user selects the "complete setup" button and will then load into either AI or two-player mode.

Decomposition

This form is decomposed in a manner such that an internal array tracks the location of each piece. These pieces may be updated by the user through clicks. Overall, this module is a well-defined area of code that performs the exact function of allowing custom setup and generally follows a MVC (Model View Controller) format with its implementation.

Interface Specifications

- New PictureBox()
    - Draw the checkers board

Uses Relationship

No other module call instances are present.

Variables

Variable Use	Variables	Variable Type	Declaration Location
Output	C_track	Integer	Local
Output	InitialX	Integer	Local
Output	C_trackarray	PictureBox Array(31)	Local
Output	M_custom	Boolean	Local
Output	M_standard	Boolean	Local
Input	ErrorClick	PictureBox	Local

Internal Implementation

This code is meant to create picture boxes at given locations in a square matrix in order to display the black tiles. The logic is to have a variable (i) increase by 50 pixels each time and draw a new picture box at that location. At every 4 i variables, a j variable will be present to move the next time. The implementation of the logic can be found below

```
For I in range 0 to 7
     For j in range 0 to 3
           C_trackarray(c_track) = new PictureBox
           C_trackarray(c_track).position = (I*50 + InitialX,j*50)
           C_track +=1
           ErrorClick.sendtoback
```

Testing – Public Interface

Action	Output	Pass Instance
Click Finish	Enters 2 player Play	Pass (1/1)
Click Black AI	Enters play with Black AI	Pass (1/1)
Click Red AI	Enters play with Red AI	Pass (1/1)
Clicks on Black Space, State:Red	Red Piece appears on clicked box	Pass (1/1)
Clicks on Black Space, State:Black	Black Piece appears on clicked box	Pass (1/1)
Clicks on Black Space, State:Red King	Red King Piece appears on clicked box	Pass (1/1)
Clicks on Black Space, State:Black King	Black King Piece appears on clicked box	Pass (1/1)

Testing – Private Implementation

Action	Output	Pass Instance
Clicks On PictureBox Instance	Label displays Picturebox Index	Pass (32/32)
Eg: PictureBox0	0	Pass (1/1)
PictureBox1	1	Pass (1/1)
….	2-30	Pass (29/29)
PictureBox31	31	Pass (1/1)

Form: Play *Reorganized*

Description

The play form is a 2-player generic checkers form. It allows for turns, legal moves, crowning and recognizing a winner.

Decomposition

The Play form is another well-defined area of code, specifically used for the 2-player mode. Similarly to previous modules, it has an internal controller array which dictates the current location of checker pieces, the model is dictated by legal move functions and the view is given to the user and updates based on user inputs. This generally follows the MVC architecture of design.

Interface Specification

- New PictureBox()
  - 32 picture box objects are created for legal moves
- FirstClick()
  - Clicking on a picturebox selects the piece on it
- SecondClick()
  - The second click moves the first click piece, if the move is illegal, it moves the piece back and stays focused on the original piece
- Current Piece()
  - Displays the current selected piece and its relevant information
- Resign() Added
  - Based, on the turn, forfeit the game

Uses Relationship

- Startmenu, Startmenu.Getsetup()
  - Returns initial gamesetup array (the internal controller for the form)

Variables

Variable Use	Variable Name	Variable Type	Declaration Location
Input	M_standard	Integer Array(32)	Global
Internal/Output	C_nextR1, C_nextR2	Strings	Local
Internal/Output	C_nextR3, C_nextR4	PictureBoxes	Local
Internal/Output	C_nextB1, C_nextB2	Strings	Local
Internal/Output	C_nextB3, C_nextB4	PictureBoxes	Local
Internal	E_MoveCount	Integer	Local
Internal	C_LegalMove	Boolean	Local
Internal	M_Black	Integer Array(12)	Local
Internal	M_red	Integer Array(12)	Local
Output	C_b1,C_b2	Integers	Local
Output	C_R1, C_R2	Integers	Local

Internal Implementation – Initial Setup

Output: Gamesetup(32) Type: Integer Array(32)

Logic: Input	Output Action
1	Black
2	Red
3	Black King
4	Red King

Internal Implementation – Decide Move

Selected Piece	Output Action
Is Black and Occurs in M_Black	C_B1 = 3, C_B2 = 4
Is Black and Occurs not in M_Black	C_B1 = 4, C_B2 = 5
Is Red and Occurs in M_red	C_R1 = 3,C_R2 = 4
Is Red and Occurs Not in M_red	C_R1 = 4,C_R2 = 5

Internal Implementation – NextPiece

```
Inputs = {NextR1, NextR2 = Currentpiece.index +C_R1,
Currentpiece.index +C_R2; NextB1, NextB2 = Currentpiece.index +C_B1,
Currentpiece.index +C_B2}
Outputs = {C_B1,C_B2,C_R1,C_R2}
```

Input	Piece Present	Output Action
NextB3.image =/= CurrentPiece.image	Black	C_B1 = 7, C_B2 = No change
NextB4.image =/= CurrentPiece.image	Black	C_B1 = No Change, C_B2 =9
NextR3.image =/= CurrentPiece.image	Red	C_R1 = 7, C_R2 = No change
NextR4.image =/= CurrentPiece.image	Red	C_R1 = No Change, C_R =9
Next all Images are already filled	Red/Black	Disallow move (If Currentpiece is nothing)

Internal Implementation – Move Logic

```
If ClickedPiece. Index = CurrentPiece.index + {C_B1, +C_B2} or {C_R1,
+C_R2} then
     NextB3/NextB4.image = nothing
     ClickedPiece.image = CurrentPiece.image
```

Internal Implementation – Save Game

The implementation of this module is a simple write to file operation as modelled by the following pseudo code:

```
Dim writer as print writer
Write(../SavedGame.txt)
Writer(gamesetup)
```

Testing – Public Interface

Action	Turn	Output	Pass Instance
Clicks On A1 (Black)	Black	Can Switch Pieces, No allowed Moves	Pass (3/3)
Clicks On E8 (Black)	Black	Can Move	Pass (1/1)
Clicks On G8 (Black)	Red	Can Switch Pieces, No allowed Moves	Pass (3/3)
Clicks On G8 (Black)	Red	Can Move	Pass (1/1)
Clicks on F1 (Blank)	Red	Nothing	Pass (1/1)
Clicks on F1 (Blank)	Black	Nothing	Pass (1/1)
Load Custom Setup	Black (Initial Condition)	Checker Piece Same as Custom	Pass (32/32)

Testing – Private Interface

Action	Turn	Output	Pass Instance
Clicks On A1 (Black)	Black	Can Switch Pieces, No allowed Moves	Pass (3/3)
Clicks On E8 (Black)	Black	Can Move	Pass (1/1)
Clicks On G8 (Black)	Red	Can Switch Pieces, No allowed Moves	Pass (3/3)
Clicks On G8 (Black)	Red	Can Move	Pass (1/1)
Clicks on F1 (Blank)	Red	Nothing	Pass (1/1)
Clicks on F1 (Blank)	Black	Nothing	Pass (1/1)
Load Custom Setup	Black (Initial Condition)	Checker Piece Same as Custom	Pass (32/32)

Form: PlayAsBlk *New*

Description

This module allows the user to play as the black side against the Red AI. The form follows a turn based operation and allows the player to resign (forfeit) on his turn. This form loads from other forms and thus its operation is either based on a standard or custom setup, both of which are dictated by the user upon start-up.

Decomposition

This area of code is the first half of a "play against AI" functionality. It allows the user to play as the black pieces while having the AI opponent play as red. Like the play form, this operates an individual game and has self-sustaining abilities for its specified functions (with the exception of moving user placed red pieces). This generally follows an MVC model as it also has a setup array as the controller. The view is based off of the control and the move model. As this form performs a specific, well-defined function it has been given a unique module.

Uses Relationship

- Startmenu.getsetup
    - Returns the setup of a standard game based on the start menu or returns the setup of the loaded game from the start menu
- Custom.getsetup
    - Returns the setup of a user defined custom game from the custom form

Variables

Variable Use	Variable Name	Variable Type	Declaration Location
Input	C_GamesetupC	Integer Array(32)	Global
Input	C_GamesetupS	Integer Array(32)	Global
Input	C_GamesetupL	Integer Array(32)	Global
Input	M_Badnum	Picturebox	Local
Internal	C_B1	Integer	Local
Internal	C_B2	Integer	Local
Internal	C_R1	Integer	Local
Internal	C_R2	Integer	Local
Internal	C_B3	Picturebox	Local
Internal	C_B4	Picturebox	Local
Internal	C_R3	Picturebox	Local
Internal	C_R4	Picturebox	Local
Internal	M_BlackT	Boolean	Local
Internal	M_RedT	Boolean	Local
Internal	E_MoveCount	Integer	Local
Output	M_Infoarray	String Array (31)	Local
Output	CompNext, CompJump	PictureBox	Local

Internal Implementation – Board Setup

Similarly to previous custom mode, this section will require a game board, as setup by the following pseudo code:

```
For I in range 0 to 7
     For j in range 0 to 3
            C_trackarray(c_track) = new PictureBox
            C_trackarray(c_track).position = (I*50 + InitialX,j*50)
            C_track +=1
            ErrorClick.sendtoback
     Next
Next
```

Internal Implementation – Move Logic

```
Output: {C_B1, C_B2, C_R1, C_R2} Types: Integers
```

Logic: Row Number	Output Action: C_R1,C_R2	Output Action: C_B1,C_B2
Odd	3,4	4,5
Even	4,5	3,4

Internal Implementation – Detect Piece Logic

```
Updates: MoveCount Types: Integers
```

Logic: Click	Update: Movecount (Current Value: 0)	Update: Movecount (Current Value: 1)
Object: Empty	0	1
Object: Not Empty	1	0

Internal Implementation – Jump Logic

This logic follows the move piece logic. Simply put, it checks if the moves will land on tiles of other pieces. If a jump can be made, it removes the piece from the originally occupied tile and moves it to another tile (the jumped location).

```
Output: {C_B1, C_B2, C_R1, C_R2} Types: Integers
```

Logic: Next Piece	Update: Same Color	Update: Different Color
Top Right Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Top Right Corner: Not Empty	C_B1, C_B2, C_R1, 0	C_B1, C_B2, C_R1, C_R2+4
Top Left Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Top Left Corner: Not Empty	C_B1, C_B2, 0 C_R2	C_B1, C_B2, C_R1+4, C_R2
Bottom Left Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Bottom Left Corner: Not Empty	0, C_B2, C_R1, C_R2	C_B1+4, C_B2, C_R1, C_R2
Bottom Right Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Bottom Right Corner: Not Empty	C_B1, 0, C_R1, C_R2	C_B1, C_B2+4, C_R1, C_R2

Internal Implementation – King Logic

```
Outputs: {King, No Change} Types: Picturebox
```

Piece Location/ Color	Output
28-31/ Red	Red King
28-31/ Black	No Change
0-3/ Red	No Change
0-3/ Black	Black King
Else	No Change

Internal Implementation – Ai Logic

Current Piece	Next Piece Left/Right	Move
Red	None/None	Move Left Corner
Red	Black/None	Jump Left Black
Red	None/Black	Jump Right Black Piece
Red	Black/Black	Jump Left Black Piece
Black	Any	None

Testing – Public Interface

Action/State	Turn	Output	Pass/Fail
Ai can Jump Piece(s)	Not Ai	No Move	Pass (1/1)
Ai can Jump Piece	Ai	Jump left most piece	Pass (1/1)
Ai can Jump 2 Pieces	Ai	Jump left most piece	Pass (1/1)
User cannot Jump Piece	User	All Available Moves	Pass (1/1)
User cannot Jump Piece	Ai	No Allowed Move	Pass (1/1)
User can Jump Piece	User	Force Jump, Disable all other pieces	Pass (1/1)
User can jump Piece	Ai	No Action	Pass (1/1)
Resign	User	Close Form	Pass (1/1)
Piece at the end	Both	Crowned Piece	Pass (1/1)

Testing – Private Implementation

Action	Calculation Piece	Variables	Pass or Fail
Click On Piece during turn	Current.image = Clicked Piece	C_next3, C_next4 = currentpiece.index + Constants	Pass (1/1)
Clicked On Piece Not one Turn	Current.image = Clicked Piece, Disabled Move	No Calculation	Pass (1/1)
Picturebox20 -> 16	(Black) PictureBox20	D4 = E3.image, E3 is empty	Pass (1/1)
PictureBox9 -> 13	(Red) PictureBox9	B6 = A7.image, A7 = empty	Pass (1/1)
PictureBox10 -> 14	(Black)PictureBox10	Picturebox14.image= Pictuerbox10, PictureBox is empty	Pass(1/1)
PictureBox19->16	(Red) PictureBox19	Picturebox19.image= Pictuerbox16, PictureBox is empty	Pass(1/1)
PictureBox 13->16, Jumped By Red	(Black) PictureBox13	PictureBox13 = red.image	Pass(2/2)
Picturebox9->16, Jumps Red Piece on 13, In turn Jumped By red 22->15	(Black) PictureBox13 (Red)PictureBox 22	PictureBox13 emptied twice, First Black then red	Pass(3/3)
PictureBox0 -> 9, PictureBox 5 is Red	(Black) PictureBox0	Nothing	Failed(1/1)
PictureBox0 Calculations	(Black) CompJump	CompJump = PictureBox8	Failed(1/1)
PictureBox5 Calculations	(Red) CompJump	Nothing	Pass(2/2)
PictureBox5 Focus	(red) CurrentPiece	CurrentPiece.image =/= red.image	Passed(1/1)
Resign_Click	M_Resign	M_Resign = 6	Passed(1/1)
Start_Menu.open	M_Resign	------	Failed(1/1)
Call RedCount()	If GamesetUp(i) = 1	GamesetUp	Passed(12/12)
Call BlackCount()	If GamesetUp(i) = 2	GamesetUp	Passed(12/12)
Resign_Click	M_resign	M_Resign = 7	Passed(1/1)
PictureBox next Piece	PictureBox(0 to 31)	CompNext, CompJump	Passed (48/64) Failed (18/64)
C_controlarray(0 to 31)	C_InfoArray	C_infoArray(31)	Passed(32/32)
C_CurrentPiece(0 to 31)	C_trackarray(31)	Currentpiece.image	Passed(24/32) Not Focused(8/32) Overall Pass(32/32)

Form: PlayAsRed *New*

Description

This module allows the user to play as the red side against the black AI. The form follows a turn based operation and allows the player to resign (forfeit) on his turn.

Decomposition

This area of code is half of a play against AI and allows the user to play as black while having the AI play as the red opponent. Like previous forms, this operates as an individual game and has self-sustaining abilities for its specified functions; it can then be classified as a well-defined area of code. Furthermore, the operation method of this form is a similar MVC model to the previous form, with the controller being the interpretation of the move logic, the view being the game board and the model being the move logic.

Uses Relationship

- Startmenu.getsetup
  - Returns the setup of a standard game based on the start menu or returns the setup of the loaded game from the start menu
- Custom.getsetup
  - Returns the setup of a user defined custom game from the custom form

Variables

Variable Use	Variable Name	Variable Type	Declaration Location
Input	C_GamesetupC	Integer Array(32)	Global
Input	C_GamesetupS	Integer Array(32)	Global
Input	C_GamesetupL	Integer Array(32)	Global
Input	M_Badnum	Picturebox	Local
Internal	C_B1	Integer	Local
Internal	C_B2	Integer	Local
Internal	C_R1	Integer	Local
Internal	C_R2	Integer	Local
Internal	C_B3	Picturebox	Local
Internal	C_B4	Picturebox	Local
Internal	C_R3	Picturebox	Local
Internal	C_R4	Picturebox	Local
Internal	M_BlackT	Boolean	Local
Internal	M_RedT	Boolean	Local
Internal	E_MoveCount	Integer	Local
Output	M_Infoarray	String Array (31)	Local
Output	CompNext, CompJump	PictureBox	Local

Internal Implementation – Board Setup

Similar to the previous custom mode, this section will require a game board, as setup by the following pseudo code:

```
For I in range 0 to 7
     For j in range 0 to 3
           C_trackarray(c_track) = new PictureBox
           C_trackarray(c_track).position = (I*50 + InitialX,j*50)
           C_track +=1
           ErrorClick.sendtoback
     Next
Next
```

Internal Implementation – Move Logic

`Output: {C_B1, C_B2, C_R1, C_R2} Types: Integers`

Logic: Row Number	Output Action: C_R1,C_R2	Output Action: C_B1,C_B2
Odd	3,4	4,5
Even	4,5	3,4

Internal Implementation – Detect Piece Logic

`Updates: MoveCount Types: Integers`

Logic: Click	Update: Movecount (Current Value: 0)	Update: Movecount (Current Value: 1)
Object: Empty	0	1
Object: Not Empty	1	0

Internal Implementation – Jump Logic

This logic follows the move piece logic. Simply put, it checks if the moves will land on tiles of other pieces. If a jump can be made, it removes the piece from the originally occupied tile and moves it to another tile (the jumped location).

`Output: {C_B1, C_B2, C_R1, C_R2} Types: Integers`

Logic: Next Piece	Update: Same Color	Update: Different Color
Top Right Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Top Right Corner: Not Empty	C_B1, C_B2, C_R1, 0	C_B1, C_B2, C_R1, C_R2+4
Top Left Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Top Left Corner: Not Empty	C_B1, C_B2, 0 C_R2	C_B1, C_B2, C_R1+4, C_R2
Bottom Left Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Bottom Left Corner: Not Empty	0, C_B2, C_R1, C_R2	C_B1+4, C_B2, C_R1, C_R2
Bottom Right Corner: Empty	C_B1, C_B2, C_R1, C_R2	N/A
Bottom Right Corner: Not Empty	C_B1, 0, C_R1, C_R2	C_B1, C_B2+4, C_R1, C_R2

Internal Implementation – King Logic

```
Outputs: {King, No Change} Types: Picturebox
```

Piece Location/ Color	Output
28-31/ Red	Red King
28-31/ Black	No Change
0-3/ Red	No Change
0-3/ Black	Black King
Else	No Change

Internal Implementation – AI Logic

Current Piece	Next Piece Left/Right	Move
Red	None/None	Move Left Corner
Red	Black/None	Jump Left Black
Red	None/Black	Jump Right Black Piece
Red	Black/Black	Jump Left Black Piece
Black	Any	None

Testing – Public Interface

Action/State	Turn	Output	Pass/Fail
Ai can Jump Piece(s)	Not Ai	No Move	Pass (1/1)
Ai can Jump Piece	Ai	Jump left most piece	Pass (1/1)
Ai can Jump 2 Pieces	Ai	Jump left most piece	Pass (1/1)
User cannot Jump Piece	User	All Available Moves	Pass (1/1)
User cannot Jump Piece	Ai	No Allowed Move	Pass (1/1)
User can Jump Piece	User	Force Jump, Disable all other pieces	Pass (1/1)
User can jump Piece	Ai	No Action	Pass (1/1)
Resign	User	Close Form	Pass (1/1)
Piece at the end	Both	Crowned Piece	Pass (1/1)

Testing – Private Implementation

Action	Calculation Piece	Variables	Pass/ Fail
Click On Piece during turn	Current.image = Clicked Piece	C_next3, C_next4 = currentpiece.index + Constants	Pass (1/1)
Clicked On Piece Not one Turn	Current.image = Clicked Piece, Disabled Move	No Calculation	Pass (1/1)
Clicked On resign	M_Resign	M_resign = 6	Pass (1/1)
Jumped Piece	Current.image = Clicked Piece	C_next3 or C_next4 image removed	Pass (1/1)
Ai Jumped Piece	Controlarray Image focused	Compnext1, compnext2 Removed	Pass (1/1)

REQUIREMENT TRACE BACKS

Generally speaking, "this assignment consists of generating the graphical representation of a checkers board, and being able to specify initial piece positions" (from Assignment 1 specifications). The following will describe how the implementation of our classes reflects and accomplishes the prescribed requirements.

Note that changes to the design have been indicated with *New*, meaning we have implemented a new class previously inexistent, *New name*, meaning we have simply renamed the class for increased coherency, and *Revised*, meaning we have revised the class to manage its efficiency and tasks handled. Note that classes missing from this version have been removed since the last and are therefore irrelevant to the systems current traceability.

`Start_Menu` *Revised*

The `Start_Menu` window is now the first one seen when launching the application as, originally, a player would simply be thrown into a standard game. However, with the option to save and load games, a more adequate solution was necessary to allow the user to choose which game method he/she prefers upon start-up. Usability is thus greatly enhanced. The page itself contains buttons including "Start Game" and "Load Game". "Load Game" will launch a game previously saved (launches `Load_Game`) whereas "Start Game" will give two game option: "Standard" or "Custom". As is obvious by the names chosen, "Standard" will load a checkers board arranged with the pieces in a standard manner (ie. three rows of red and black pieces on opposite sides, launches `Standard_Mode`) and "Custom" will give the player the opportunity to design a custom game, placing up to 12 pieces of each colour on any black tile (launches `Custommode_Load`). Selecting "Standard" game mode also reveal two additional options: "Standard (Red AI)" and "Standard (Blk AI)". As is obvious by the chosen names, the first will load a game in which the user will play as black and the AI will play as red, whereas the second will load a game in which the user will play as red and the AI will play as black.

Even though `Start_Menu` does not accomplish any of the requirements described in Assignment 2, it is imperative to the applications ease of use as it offers a simple user interface (UI) from which a player may select from many game modes.

`Play` *Revised*

This form displays the general user interface. This includes, in broad terms, the game board, a title, a score board for each player and alphanumeric place holders as well as a menu strip located at the top.

The game board consists of light and dark squares (8 on the height and 8 on the width) from which a light square may be found in the bottom right corner. The board itself (modeled to imitate a wooden surface) lies on a coloured background image. The different colored tiles on the board are easy to differentiate, making piece recognition and placing rather simple and intuitive.

The board is also traced by a sequence of letters and numbers used to identify individual piece locations. The letters (placed on the top and bottom) start with A on the left and end with H on the right-most tile. Similarly, the numbers (found on the right and left sides of the board) start with 1 at the bottom tile and incrementally increase up to 8 at the top tile.

Therefore, `Play` manages to accomplish the "initial set up" of "an 8-by-8 checkers board with dark and light squares" in which "a light square" is found "in the bottom right corner" of the playing surface. Place naming conventions have also been met.

Originally, `Play` managed two independent modules which allowed their own playing experience. This, however, necessitated an enormous amount of code duplication. It was therefore decided that the `Play` form would be the one on which all 2-player gameplay would be controlled, eliminating otherwise necessary repetition and, hence, avoiding possible pitfalls such as duplicated bugs.

This module therefore imports game setups from `Standard_Mode`, `Custommode_load` and `Load_Game` as well as containing all movement logic. The `Play` module allows a user to "start a game from a previously stored state", "make moves" that include "mov[ing] pieces from one position to another" that include "mov[ing] a piece to another square; jump[ing] the opponent's piece (so that piece is removed from the board); convert[ing] a piece to a "king" [and] mov[ing] kings in both directions (forwards and backwards)" (Assignment 2). Finally, this form allows "two people [to] play agains each other" (Assignment 3). This is all accomplished using a graphical user interface in which the user must simply click the piece he/she desires to move and then select the empty square they would like to move the piece to. The action will be completed providing the move is legal.

`Standard_Mode` *Revised*

As was aforementioned, `Start_Menu` included a button "Standard" under "Start Game". Selecting this game option pushes the locations of the game pieces to the `Play` window, displaying a standard game piece arrangement. Hence, "the user [is] able to set up an initial position of pieces on the board by specifying […] the standard opening position".

Upon pressing "Standard" in the start menu, the play window opens, replacing the initially viewed menu. It then places three rows of red pieces (white pieces on the Assignment specification diagram) on rows one to three, strictly positioned on dark tiles. Similarly, black pieces are placed on rows six to eight, once again, only on black tiles. A total of 12 pieces for each colour is placed on the board.

The initial game setup is therefore completely legal by default and obeys the initial setup requirements instilled by the assignment specifications.

`Custommode_load` *Revised*

This class, as the name implies, allows the user to initiate a custom game in which pieces are placed at the user's will. This mode is accessed through the start menu after selecting "Start Game" followed by "Custom".

After selecting this mode of gameplay, a single red game piece appears at the bottom of Form1 accompanied by a "Complete Setup" button as well as a label that reads "Click on the left picture to change pieces". At this moment, the red piece is selected and clicking on any black tile will place a red piece at that location. Clicking on the red piece outside of the board will circulate a red king piece. At this time, clicking on any black square will place a red king. A total of twelve combined king and standard red pieces may be placed. Clicking again on the selection piece will make a black piece appear followed by a black king piece, allowing the user to place black and black king pieces on the board respectively. Finally, pressing on the black king piece will show an empty space with the caption "Now removing. Click me to go to red". This allows the user to remove pieces originally placed on the board. Simply clicking on any dark tile housing any piece will remove it from game play.

Tiles may be overwritten in custom mode's set-up process. In other words, setting a piece of one color and then clicking the same space with a different piece will place the latter piece on the tile. Clicking multiple times on the same tile while in the same piece mode has no effect other than placing a piece on the first click. Clicking on any white space on the board during custom setup in any piece mode will display a pop-up window which reads "You cannot place a piece here" notifying the user that he/she has attempted an illegal procedure. Accepting this dialog box returns the player to the custom setup.

Once the player is content with the placement of all the game pieces, he/she may press the "Complete Setup" button at the bottom of the form. At this point, the Custommode_load window disappears, giving way to the Play window which has imported the piece locations designated by the user while in custom set-up. The game is then ready to start.

Therefore, the player uses a graphical interface method of placing pieces as opposed to the much less intuitive and timely alternative method offered alongside in the Assignment specifications. Requirements such as "users shall be warned if the position is illegal" and "pieces must not be placed on illegal squares (white/light square)" as well as "a maximum of 12 [red] pieces and 12 black pieces may be placed on the board" are all met within this class' interface. There is also "a way for the user to indicate that set up is complete" by using the "Complete Setup" button and commencing the game.

Save_Game *Revised*

The Save_Game module allows a user to save a game in progress. At this point, any legal game play mode may be saved which includes standard and empty boards.

Saving a game may be accomplished through the menu strip located on the Play form under "Menu". Selecting "Save Game" from these options will record piece locations in a file which may be retrieved whenever "Load Game" is executed. The state of the game in progress is therefore "saved within a file [which may] be resumed later" (Assignment 2).

Requirements are therefore met in order to save game progress at any moment during gameplay, allowing the user to record a games state to be retrieved and reloaded at a later date. Also note, since there is only one save slot at this time, that saving multiple times overwrites any older game state between the current game and past games.

Load_Game *Revised*

As opposed to an option within the Play menu strip, Load_Game can be found as a game mode in the Start_Menu form. Simply put, this allows Play to load piece locations saved previously using Save_Game. Since there is, at the moment, only one storing state, Load_Game loads the last-saved (most currently saved) game. This allows a user to play a game saved at an earlier date.

PlayAsBlk *New*

This form allows a user to play the computer (AI) as red with a black opponent. Its graphical user interface is, in all key aspects, identical to that of the Play form (refer to said form for details concerning the GUI).

This form does, however, allow the unique ability to play against a computer and therefore satisfies that "one person plays against the computer", "let[ting] the user choose which colour pieces they want to use" (Assignment 3).

PlayAsRed *New*

Not dissimilar from the above form, this one allows a user to play the computer (AI) as black with a red opponent. Its graphical user interface is, in all key aspects, identical to that of the Play form (refer to said form for details concerning the GUI).

Similarly to the above, this form offers the ability to play against a computer and therefore satisfies that "one person plays against the computer", "let[ting] the user choose which colour pieces they want to use" (Assignment 3). Both PlayAsBlk and PlayAsRed cover both binary instances in which a player would want to play against a black or red AI respectively.

INTERNAL REVIEW AND EVALUATION OF DESIGN

The newest update to Ultimate Checkers (UC) adds the ability for the user to play against the AI as either red or black, or against another user; there are also changes in the overall structure of the program in terms of the way the modules interact with one another and how they generally function.

Upon start-up of the application, UC looks similar to its previous version. This time, however, there are new options to select from the home menu. These new options, when selected, load a standard board and allow the user to play against the AI as whatever colour they want (either black or red). Users will not however, be able to save their games and load them again in any AI mode. If users do not want to play against the AI, they can play with one another in the revised versions of standard mode or custom mode. These revisions have proper turns, only allow legal moves and give the ability to play a full game of checkers until either player runs out of pieces or resigns (forfeits). Game progress may be saved in this case and loaded at another time.

The changes to the structure of UC are visible in the code; this application reflects the idea that programs should have high cohesion and low coupling within their modules. Changes were made to some modules, and new ones were implemented to create the new AI. Nonetheless, this did not affect the proper functioning of the overall system.

Furthermore, most of the modules specialize in one area of the program. An example of this is the PlayAsBlk module, which specializes in playing against the AI when the user elects to play as black. It doesn’t require any other module’s data or outputs (with the exception of various setup modules that are required to display the board) and it has been broken into its own unique module. Overall, UC is a great example of how programs should have high cohesion and low coupling in order to maximize efficiency, readability and maintainability.

ANNEX

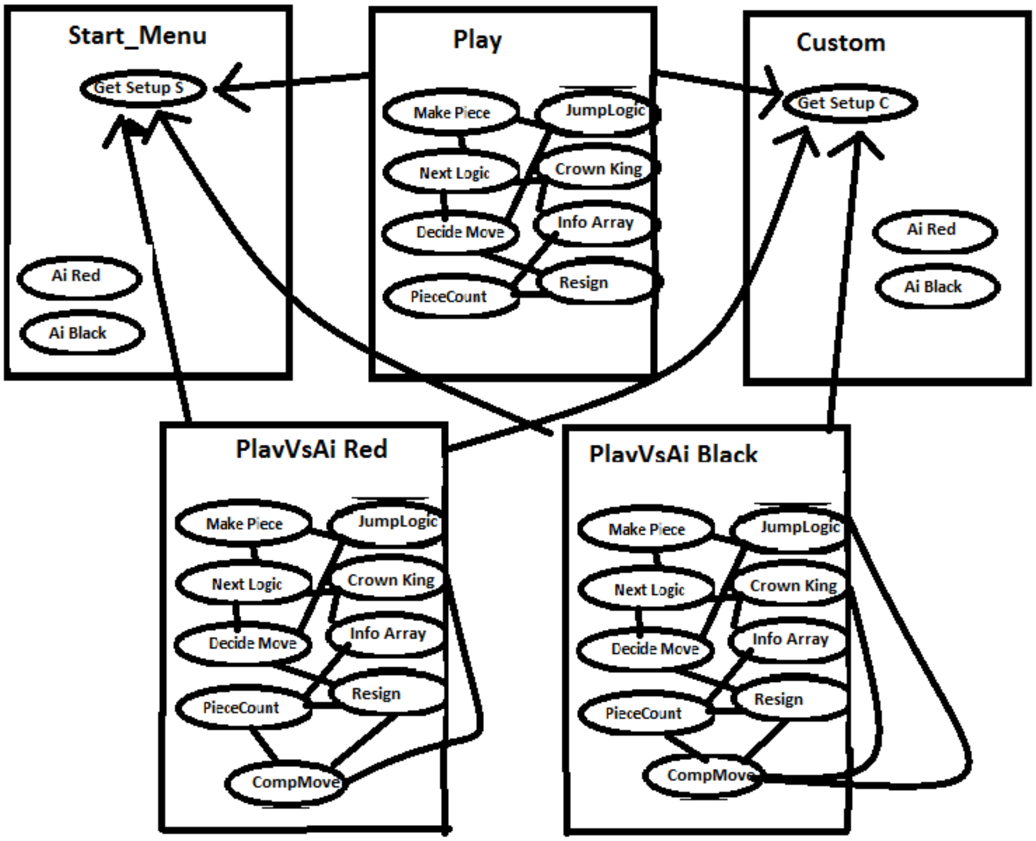

Figure 1

Current hierarchy. Notice the high level of cohesion and low coupling.