Comp Sci 2ME3

Software Design

Assignment 3 : American Checkers Complete Game

Olabosipo Balogun	1069215
Ashrafal Chowdhury	1230995
Hyonwoo Kee	0746923
JingChao Ni	0852195
Peng Zheng	1230209

1. Introduction

Third part of checkers assignment will produce a fully working game.

Rule of American Checkers: http://boardgames.about.com/cs/checkersdraughts/ht/play_checkers.htm

1.1 Problem description

1.1.1 Changes - design

- From one class program -> object oriented programming
- Improved hierarchy among classes
- Now the orientation is changed (red pieces at the bottom and yellow pieces at the top)

1.1.2 Changes - code

- Single java file -> coordinated development by modules

1.1.3 Changes - requirements

- This changed requirements from next section is the major problem

1.2 Requirements

1. Fully working checkers game, against a player or computer

    a) starting of a game from standard starting position

    b) starting of a game from a previously stored state (file I/O)

    c) two players game, or against computer

    d) indicate whether a game has been won

    e) save a game for resume (b)

2. revision history of the design document

    a) extension to the existing design document (MIS / MID)

    b) extension to the existing code (will be new set)

    c) test record document

1.3 Decomposition of Classes

There will be main code game.java, board.java, gamebody.java, pieces.java, gamecontrol.java.

Game.java - executes the game

Gamebody.java - control of the overall game

Board.java - sets up the board

Pieces.java - pieces attributes

Gamecontrol.java - determines rule & legit moves

1.4 Overall Design

1. game.java integrates all the modules, and will include string[] args
2. gameboard sets up the layout, including menu, board, and structures
3. gamebody class will be the main body of the game; will impose rules
4. pieces will be instantiated by game body and will use gamecontrol method

   it was used extensively through the game, extension was done for later update
5. gamecontrol methods specify rules, called by gamebody.

1.4.1 Private implementation

1. JPanel Checkers
2. JMenuItem menuOption

1.4.2 Public interface

1. Board(), gameBoard

1.4.3 Classes

1. Pieces
2. Gamebody
3. Gamecontrol:

(i)isKingMove():
Checks if the piece is a king or a regular based on the value of the square of the board and implements the movement rules to the piece. This is done by checking the lblBoard[countThis].getText()

If the value returned is "K", a set of rules is implemented which allows the piece to move diagonally in the forward or backward direction irrespective of the colour of the piece.

If the square has a piece of the same colour, the move is declined. This is done by checking the lblBoard[countThis].getBackground() of the square to be moved to.

If the square has a piece of the opposite colour, that piece is removed from the board and the present piece is moved one square further given that there is an available square. This is done by checking the lblBoard[countThis].isOccupied of that square. Whenever a piece is removed, the maximum number of possible pieces on the board is decreased by 1. This is done by the"redMax --" or "yelMax --" syntax depending upon whichever colour is removed.

If the square is empty, the piece is moved to that square. This is done by checking the lblBoard[countThis].isOccupied of that square.

(ii)isRegMove():

Checks if the piece is a king or a regular based on the value of the square of the board and implements the movement rules to the piece. This is done by checking the lblBoard[countThis].getText()

If the value returned is “”, a set of rules is implemented which allows the piece to move diagonally in the forward direction in the case of the red pieces and the backward direction in case of the yellow pieces. (perspective of a red piece)

If the square has a piece of the same colour, the move is declined. This is done by checking the lblBoard[countThis].getBackground() of the square to be moved to.

If the square has a piece of the opposite colour, that piece is removed from the board if the piece is not a king and the present piece is moved one square further given that there is an available square. This is done by checking the lblBoard[countThis].isOccupied of that square. Whenever a piece is removed, the maximum number of possible pieces on the board is decreased by 1. This is done by the“redMax --” or “yelMax --” syntax depending upon whichever ever colour is removed.

If the square is empty, the piece is moved to that square. This is done by checking the lblBoard[countThis].isOccupied of that square.

iii) canMoves():

A method to check if the move is legal or not from one place to another. Assuming that position(x0,y0) contains one player’s piece and position (x1,y1) contains one piece from other player.

iv) canJump():

A method to check if the move is legal to jump from one position to another. Assuming that (x0,y0) contain one player’s piece and that (x2,y2) is a position that is 2 rows and 2 columns distant from (x0, y0) and that (x1,y1) is the square between (x0,y0) and (x1,y1).

1.3.3 Uses Relationship (-> denotes calls)

game -> gameboard, gamebody, pieces

gamebody -> pieces, gamecontrol

gamecontrol -> pieces

board ->pieces

1.3.4 Traceability

vs. original design : i) A new selection named “save” and “load” will be implemented to the game menu for user’s convenience as required.

ii) The menu selection interface will connect with piece placement, which is an unfinished part from assignment 1.

iii) Piece movements will be working after the piece setup is finished and a “start” will be displayed as one of the options for user to start the game.

vs. requirements : 1) starting of a game from standard starting position

Module used: board.java, piece.java

Status: Implemented

Test: this will be tested by clicking on Standard Game button on the interface.

2)starting of a game from a previously stored state (file I/O)

Module used: board.java, piece.java

Status: Implemented

Test: this will be tested by first placing some peices of different types on the board, then click on the Save button to save the piece locations. We will then try to load saved state and resume the game.

3)make legal diagonal moves or jumping over the opponent’s piece makes it disappear.

Module used: Gamecontrol.java

Status: Implemented

Test: this will be tested by moving pieces over oppenent pieces.

4)a piece converting to a king piece, which can move forwards and backwards.

Module used: Gamecontrol.java

Status: Implemented

Test: This will be tested by moving a regular piece to the bottom line of opponent side to see whether it converts to king piece. We will also place king pieces anywhere on the board. In both cases pieces will be tested whether they are able to move both forward and backward.

1.4.1 Evaluation of the design

Our design is adequate but not perfect at the moment. Interface will be simple and user-friendly. Pieces will be able to move according to the rules. However, we could decompose certain module(such as Gamecontrol) further than we currently have.Function like Save/Load is not able to save/load multiple different states. Our design currently does not consider possible future development such as player vs player, player vs computer. Future method may have problem accessing current methods. During code implementation, some method might not be fully utilized or used at all.

1.5 Testing Cases

testJButton btnStandard: ActionListener

Test the standard option using the following test data:

JButton btnStandard (ActionListener())

Expected Output: Movement();

testJButton btnCustom: ActionListener

Test the custom option using the following test data:

JButton btnCustom (ActionListener())

Expected Output: Setup();

testJButton btnQuit: ActionListener

Test the Quit option using the following test data:

JButton btnQuit (ActionListener())

Expected Output: exit the game

testJButton btnStart: ActionListener

Test the Start button using the following test data:

JButton btnStart (ActionListener())

Expected Output: Movement();

testJButton btnSave: ActionListener

Test the Save option using the following test data:

JButton btnSave (ActionListener())

Expected Output: saveBoard(Pieces boardMe[]) (save.txt is created)

testJButton btnLoad: ActionListener

Test the Load option using the following test data:

JButton btnLoad (ActionListener())

Expected Output: btnLoad.addActionListener()

1.5.1 Logic : determination of legit moves
testIsKingMove(): boolean

Test the isKingMove() method using the following test data:

isKingMove()

Input: “K”

Expected Output: true (allow to move)

Input: “J”

Expected Output: false (not allow to move) testIsRegMove(): boolean

Test the isRegMove(): method using the following test data:

isRegMove()

Input: “”

Expected Output: true (allow to move)

Input: “A”

Expected Output: false (not allow to move)

Module Interface Specification

MODULE: Board

USES: Game.java, GameBody.java, Pieces.java

TYPE: None

ACCESS PROGRAMS:

Board (): board
- Constructs a chess board with proper labels. The method involves a combination of various classes that are imported from Java Library, such as JFrame, JPanel, JButton, JLabel, ActionListener, etc.
It returns a chess board as a whole with all referred classes.

setredMax (aNum: int): void
- Constructs a new method that creates and updates number of red chess being placed.
Input is set to 0 piece for red if standard game is selected, and is set to 12 pieces if clear option is selected, and it returns void type.

setyelMax (aNum: int): void
- Constructs a new method that creates and updates number of yellow chess being placed.
Input is set to 0 piece for yellow if standard game is selected, and is set to 12 pieces if clear option is selected, and it returns void type.

getredMax (): int
- Retrieve number of red chess that are being placed currently.
It retuns integer as output.

getyelMax (): int
- Retrieve number of yellow chess that are being placed currently.
It retuns integer as output.

Module Interface Design

Module: Board.java

This module is to set up a chess board.

USES: Game.java, GameBody.java, Pieces.java

VARIABLES:

colourSelect: int
kingSelect: int
redMax: int
yelMax: int
gameTurn: int
here: int
there: int
Checkers: JPanel
menuOption: JMenuItem
lblBoard[]: array of Pieces
infoAddr[]: array of JLabel

ACCESS PROGRAMS:

Board (): board

- Constucts a chess board game interface with multiple classes involved. The chess board is consist of 100 JButton array and are built up with two for-loops for rows and columns.

```
  boardPanel.setBounds(10, 10, 500, 500); //board size 500x500
 for(int x=0; x<=9; x++){ //ROW
   for(int y=0; y<=9; y++){ //COLUMNS
lblBoard[(x*10)+y] = new Pieces();
```

- Menubar is implemented through a combination multi-class uses.

```
JPanel panel = new JPanel();
   Checkers.add(panel);
  panel.setLayout(null);

  final JButton btnStandard = new JButton("STANDARD");
   panel.add(btnStandard);

  final JButton btnCustom = new JButton("CUSTOM");
   panel.add(btnCustom);
  JButton btnQuit = new JButton("QUIT");
    panel.add(btnQuit);

  final JButton btnStart = new JButton("START");
    panel.add(btnStart);

  final JLabel info = new JLabel("Checkers!");
    info.setHorizontalAlignment(SwingConstants.CENTER);
   panel.add(info);
  infoAddr[0] = new JLabel();
  infoAddr[0] = info;

  JButton btnSave = new JButton("SAVE");
   panel.add(btnSave);

  final JButton btnLoad = new JButton("LOAD");
   panel.add(btnLoad);

  final JButton btnClear = new JButton("CLEAR");
   panel.add(btnClear);
```

setredMax (aNum: int): void
- The method takes an integer and returns void type as an response to user’s selection.

```
 public static void setredMax(int aNum){
  redMax = aNum;
 }
```

setyelMax (aNum: int): void
- The method takes an integer and returns void type as an respond to user’s selection.

```
 public static void setredMax(int aNum){
redMax = aNum;
}
```

getredMax (): int
- It retuns integer as output.

```
 public static int getredMax(){
 return redMax;
}
```

getyelMax (): int

- It retuns integer as output.

```
public static int getyelMax(){
return yelMax;
}
```

MODULE: GameBody
- This class setup the game by initiating a Board, GameControl and 24 pieces (in an array). It also contains the functionality of the buttons.

INTERFACE

USES: GameControl.java, Board.java, Pieces.java

TYPE: None

ACCESS PROGRAMS:

Gamebody(): Gamebody
Constuct a new Gamebody object which contains a name, a Board object and set the board visibility.

runIt (lblBoard[]: Pieces, countThis: Int, here: Int, there: Int, infoAddr[]: JLable): void
This method controls the game flow. If player click on a legal spot to move a piece after selecting a piece, the selected piece will be moved from here to there. Possibility of double jump is considered. It also checks if any piece can be turned to King piece on the board (lblBoard[]). JLable will show which player's turn next.

saveBoard(boardMe: Pieces): void
This method take pieces locations (boardMe) as input. Then check their colours and store them into a new array with 1 represent red piece, 2 represent red king, 3 represent yellow piece and 4 represents yellow king. It will then generate a “save.txt” file that stores this array for

future load.

loadBoard(): int[]
This method read “save.txt” file (previously saved game) then produce an array of integers with 1 representing red piece, 2 representing red king, 3 representing yellow piece and 4 representing yellow king.

clearBoard(thisBoard[]: Pieces): void
This method checks colours of the pieces on the board. If background colour is yellow or red it will turn them to black thus clearing the board. It will also reset the maximum yellow pieces and red pieces allowed to be placed on the board to 12.

standardBoard(thisBoard[]: Pieces): void
This method will first clear the board if there is any pieces on the board. Then it will set up the board with standard position: 12 pieces of yellow on top, 12 pieces red on bottom by using a nested for loop. Pieces can only be placed on black squares. Lastly it will set the maximum red/yellow pieces allowed to be placed to 0.

GameBodyChangeName(): void
This method sets/updates the name of a GameBody object. It is created for testing purpose.

startGame(rM: int, yM: int): boolean
This method checks if there is any piece on the board by checking yellowMax or redMax. Game will be able to start if there is. Otherwise it will not be able to start.

getMyName(): String
This method returns the name of a GameBody object as a string.

IMPLEMENTATION

USES: GameControl.java, Board.java, Pieces.java

VARIABLES:

gameStarted: boolean
myName: String
canJump: boolean
canMove: boolean

ACCESS PROGRAMS:

Gamebody(): Gamebody
Constuct a new Gamebody object which contains a name myName, a Board object gameBoard and sets the board visible.

runIt (lblBoard[]: Pieces, countThis: Int, here: Int, there: Int, infoAddr[]: JLable): void

This method controls the game flow.

```
if (lblBoard[countThis].getBackground() = Color.GREEN), then:
Gamecontrol.hidePiece: there
Gamecontrol.movePiece:here->there
Gamecontrol.reachEnd
Gamecontrol.showTurn: infoAddr
else:
        Gamecontrol.hidePiece: countThis
        Gamecontrol.legalMove: countThis
```

saveBoard(boardMe: Pieces): void

This method saves the colour and location of the pieces currently on the board into a txt file named “save.txt”.

```
for i=0 to 77 do:
  if color=red, then:
    saveMe[i]=1
    if text=”K”, then:
      saveMe[i]=2
  else if color=yellow, then
    saveMe[i]=3
    if text=”K”, then:
      saveMe[i]=4
try:
  for i=0 to saveMe.length do:
    output.write(Interger.toString(saveMe[i])+” “)
Exception: IO exception
```

loadBoard(): int[]

This method read “save.txt” file (previously saved game) then produce an array of integers.

```
while bufferReader.readLine() != null do:
  try:
    for i=0 to 77 do:
      loadMe[i]=Integer.parseInt[i]
      loadMe2[i]=loadMe[i]
Return loadMe2[i]
Exceptions: Parsing Int exception, Reading file exception
```

clearBoard(thisBoard[]: Pieces): void

This method clears the board and reset the piece count to 12 when the clear button is pressed.

```
for i=11 to 88 do:
  if color=red or color=yellow then:
    thisBoard[i].setBackground(black)
    thisBoard[i].setText("")
setYelMax(12)
setRedMax(12)
```

standardBoard(thisBoard[]: Pieces): void

This method will first clear the board if there is any pieces on the board. Then it will set up the board with standard position.

```
for i=11 to 88 do:
  if color=red or color=yellow then:
    thisBoard[i].setBackground(black)
    thisBoard[i].setText("")
for x=0 to 8 do:
  for y=0 to 8 do:
     countThis=x*10+y
     if (x%2 != y%2) then:
       if(color=black)
        if countThis<=39 then:
          color=yellow
        else if 88>countThis>60 then:
          color=red
setRedMax(0)
setYelMax(0)
```

GameBodyChangeName(): void

This method sets/updates the name of a GameBody object.

```
myName=something
```

startGame(rM: int, yM: int): boolean

Player can start a game when there is at least one piece of each color on the board.

```
if rM<12 and yM<12 then:
  gameStarted=true
  Return true
else:
  Return false
```

getMyName(): String

This method returns the name of a GameBody object as a string.

```
Return myName
```

Module: Gamecontrol.java

This module is to apply methods of the rules for the game.

Types: None

USES: Board.java, Pieces.java, Game.java

VARIABLES:

None

ACCESS PROGRAMS:

legalMove (Piece[], int index): boolean

Construct a method that is used for testing the move is legal or not. (Including regular moves and jump moves)

secondJump (Piece[], int index): boolean

Construct a method for checking second jump move if it is capable.

trunChange(): void

Construct a method to trace the turns on both sides.

showTurn(JLabel infoadr[]): void

Construct a method to display a message for turns, so that the users will know whose turn is on at that moment.

movePiece(Piece[], int here, int there): void

Construct a method for moving piece after it it is checked as either a legal jump move or a regular move.

findPiece(Piece[], int index): boolean

Construct a method to search through all slots on the board for any green spots existence. This method is connected and used as a condition to trigger canJump() and canMove() methods if it returns true.

showPiece(Piece[], int index): void

Construct a method to display green spots which are used to show legal moves.

hidePiece(Piece[], int index): void

Construct a method to restore green pieces to background colour(Black). This method is used once a move is made, at the meanwhile, the rest green piece will be set back to background colour.

chkBoundary(Piece[], int index): boolean
Construct a method to check if the clicked button is KING or not.

canMove(Piece[], int index): boolean
Construct a method of regular movements for KING pieces and regular pieces. The method chkBounday() helps to detect if the piece is a KING or a regular piece.

canJump(Piece[], int index): boolean
Construct a method of jump-moves for KING pieces and regular piece.

mustJump(Piece[]): void
Construct a method that detects if any jump-moves can be made, and then require to perfrom the move.

reachEnd(Piece[]): void
Construct a method that checks location for all regular pieces on both sides, and set to KING piece once the end is reached.

IMPLEMENTATIONS

USES:
Board.java, Pieces.java, Game.java

VARIABLES:
None

ACCESS PROGRAMS:
legalMove (Piece[], int index): boolean
This method will check if a move legally can be made between canJump() and canMove() , returns true as it exists and sets false for the other.

```
        if (findPiece(thisBoard, index) == true) {
    Gamebody.canJump = true;
    return true;
    } else {

   Gamebody.canJump = false;
 Gamecontrol.canMove(checkArray, index);

  if (findPiece(thisBoard,index) == true) {
    Gamebody.canMove = true;
    return true;
  }

  Gamebody.canMove = false;
  return false;
  }
}
```

secondJump (Piece[], int index): boolean

This method takes the position of jumped piece as its input and verify if a further or a second jump can be made. It returns true if a consecutive jump-move is found, otherwise canJump is false.

```
if (findPiece(thisBoard, index) == true) {
    Gamebody.canJump = true;
 return true;
} else {
        Gamebody.canJump = false;
        return false;
  }
 }
```

trunChange(): void

This method is to count whose turn is on.

```
gameTurn = 1;
yellow’s turn
gameTurn == 0;
red’s turn
```

showTurn(JLabel infoadr[]): void

This method is to set Text as JLabel on the interface so that turn message will be displayed.

```
if (gameTurn ==0)
{setText("Red's Turn")
}
if (gameTurn ==1){
setText("Yellow’s Turn")
}
```

movePiece(Piece[], int here, int there): void

This method is to move one piece from one position to another by changing the index if either canMove() or canJump() is true.

```
 if (canMove == true) {
    canMove = false;
```

```
  }
  if (canJump == true) {
   canJump = false;
    Gamecontrol.canJump(Board, there);
  }
```

Check canMove or canJump

```
Ex: //63 -> 54 -> 45 diff = 18
    //65 -> 54 -> 43 diff = 22
    //take difference of (there - here)
    //take smaller #
    //add diff/2
```

```
if (Math.abs(there - here) >= 12) {
int target = Math.min(here,there) + Math.abs(there-here)/2;
thisBoard[target].setBackground(Color.BLACK);
 if (secondJump(thisBoard, there) != true) { Gamecontrol.turnChange(); }
  } else {
Gamecontrol.turnChange();
}
}
```

findPiece(Piece[], int index): boolean

This method is to search through the board and to find any valid moves that can be made ,once a position is selected.

To do this, we need have a count variable that records all valid moves when they are found.

```
int count = 0;
for (int i = 0; i < 100; i ++) {
    if(Board[i].getBackground() == Color.GREEN ) { count ++; }  // green means valid moves
  }
// once it is found , count ++, otherwise false is returned

  if (count > 0)
{ return true; }
  return false;
 }
```

showPiece(Piece[], int index): void

This method is to show user's valid moves that are highlighted in green once a position is selected.

```
Board[index].setBackground(Color.GREEN);
```

hidePiece(Piece[], int index): void

This method is to set back the green pieces to original background colour (Black).

```
for (int i = 0; i< 100; i++) {

  if (Board[i].getBackground() == Color.GREEN){
  Board[i].setBackground(Color.BLACK);
  }

  }
 }
```

chkBoundary(Piece[], int index): boolean

This method is to check the selected button is a KING piece or not, returns true if it is and false otherwise.

```
if ((Board[index].getText() == "K")){
      //KING OR NO
   return true;
  } else {
   return false;
  }
```

canMove(Piece[], int index): boolean

This method is to check if the piece is KING or not, and then regulate the moves.

-Check if the piece is a KING piece or not,

if (chkBoundary(thisBoard, index) == true) // it is true

-Check if it is now red’s turn, and background colour = red , and piece is KING piece

((gameTurn == 0) && (Board[index].getBackground() == Color.RED) && (Board[index].getText() == "K"))

-if all true, the piece can only move to the positions where are either (-9 or -11) or (+9 or +11)(due to its a KING piece) on the board

```
if (thisBoard[index-9].getBackground() == Color.BLACK) {
    showPiece(thisBoard, index-9);
    }
    if (thisBoard[index-11].getBackground() == Color.BLACK) {
    showPiece(thisBoard, index-11);
    }
    if (thisBoard[index+9].getBackground() == Color.BLACK) {
    showPiece(thisBoard, index+9);
    }
```

```
        if (thisBoard[index+11].getBackground() == Color.BLACK) {
        showPiece(thisBoard, index+11);
        }
    else{
  // same logic for yellow KING pieces
  }
```

- Regular pieces rules are similar, the only difference is that they are only moving upward for red's and downward for yellow's. therefore, +9 or +11 for yellow pieces, and -9 or -11 for red pieces

canJump(Piece[], int index): boolean
Construct a method of jump-moves for KING pieces and regular piece.

mustJump(Piece[]): void
This method is to apply rules for jumping when the piece is capable.

-Check if the piece is a KING piece,
if (thisBoard[index].getText() == "K")
-Check the turn, selected piece's background colour.
((gameTurn == 0) && (Board[index].getBackground() == Color.RED))
-Check the position where the piece is jumping to. The background colour for that position must be opposite colour and the position itself must be taken, otherwise it is not a legal jump.
(Board[index-9].getBackground() == Color.YELLOW)
-Check -9-9 position if it is empty for jumping and the background must be black colour so that it is empty for landing.
(thisBoard[index-9-9].getBackground() == Color.BLACK)
-Same rules for the other sides. The position for legal-jump must be either -9-9 or -11-11, as well as that the background colour must be black.

reachEnd(Piece[]): void
This method is to check if any pieces made to other end for both sides. It also assigns the piece to a KING piece.

```
int count = 0;
  for (int i = 1; i < 9; i++) {
    count = i + 10; //will check 11-18   //check row by row
    if (thisBoard[count].getBackground() ==Color.RED) { thisBoard[count].setText("K"); }
  }  // assign the piece to KING once found
  for (int i = 1; i < 9; i++) {
    count = i + 80; //will check 81-88
    if (thisBoard[count].getBackground() ==Color.YELLOW) {
thisBoard[count].setText("K"); }
   }
```

}

MODULE: Pieces

This module is used to initiate Pieces objects on the board. Total 24 Pieces will be instantiated by gameBody. Pieces will have a name and index number attached to it.

USES: NONE

Access Program:

Pieces(): Pieces
This method initiate a Pieces class with a name for it.

setIndex(i: int): void
This method sets up an Index number for a Pieces object.

getIndex(): int
This method returns the index number that is given to a Pieces object.

piecesChangeName(): void
This method changes the name of a Pieces object.

myName(): String
This method returns the name of a Pieces object.

IMPLEMENTATION

USES: NONE

Variables:
myName: String
isOccupied: boolean
greenSource: int
myIndex: int

Access Programs:

Pieces(): Pieces
Create a Pieces object with a name.
myName=”something”

setIndex(i: int): void
Set index number for Pieces objects.
this.myIndex=i

getIndex(): int
Return index number.
Return this.myIndex

piecesChangeName(): void
Change myName for a Pieces object.
myName=”something else”

myName(): String
Returns the myName.
Return myName.

MODULE: Game

This module is the main body of the game. It test if all classes are instantiated, then initiate the game.

USES: Board.java, GameControl.java, GameBody.java, Pieces.java

Access Programs:

main(args: String[]): void
This is the main method of the checker game. It creates a gameBody and starts the game.

testClasses(): void
This method tests if all classes can be properly accessed and instantiated.

gameStart(): void
This method construct a new GameBody object to start the game.

IMPLEMENTATION

USES: Board.java, GameControl.java, Gamebody.java, Pieces.java

Variables: NONE

Access Programs:

main(args: String[]) void
This is the main method of the game that calls gameStart to start the game.
```
gameStart();
```

testClasses(): void
This method test if other modules can be properly called.

```
aBoard= new Board();
print(aBoard.myName);
aBoard.BoardChangeName();
print(aBoard.myName);

aGamebody= new Gamebody();
print(aGamebody.myName);
aGamebody.GamebodyChangeName();
print(aGamebody.myName);

aPiece= new Pieces();
print(aPiece.myName);
aPiece.PiecesChangeName();
print(aPiece.myName);

aGamecontrol= new Gamecontrol();
print(aGamecontrol.myName);
aGamecontrol.GamecontrolChangeName();
print(aGamecontrol.myName);
```

gameStart(): void
Create a new game interface.
```
aBody= new Gamebody();
```

Test Cases

Each test returns True on pass, False on fail, and prints test data to standard out

INTERFACE

USES:

Gamecontrol

ACCESS PROGRAMS:

testCompile: Boolean
Tests the compilation of the program

testPieces: Boolean
Tests for all the pieces are available on the board

testKing: Boolean
Tests if the regular piece can change into a king piece

testMoveReg: Boolean
Tests if the regular piece can move to a certain location

testMoveKing: Boolean
Tests if the King piece can move to a certain location

testEliminate: Boolean
Tests if the piece can be eliminated (jumped)

testBoundary: Boolean
Tests if the piece can be placed outside the board