

Part 2

by Rhysian Jones

FILE	12010_RHYSIAN_JONES_PART_2_159704_856999424.DOCX		
TIME SUBMITTED	10-JAN-2019 02:42PM (UTC+0000)	WORD COUNT	3145
SUBMISSION ID	98702466	CHARACTER COUNT	19305

PROGRAMMING AND SOFTWARE TESTING

ASSIGNMENT 1

Rhysian Jones – 1809710

Cohesion:

Cohesion is a measure of how similar the code is within a system^[1]. If a class has functions or methods which all contain a section of the same code, the class has high cohesion. This means that the code which appears multiple times can be put into a function and called multiple times, decreasing the class' complexity and increasing reusability and readability.

Having a system with high cohesion will decrease the amount of maintenance needed because there are fewer methods or functions to fix when they are broken by a change to the code. A highly cohesive system will allow developers to find and reuse methods easily, decreasing the amount of new code needed as well as the effort required. Readability is increased in a highly cohesive system because the main method can be laid out in a story-like way with each function name describing the next step in the program.

For example:

Low cohesion:
<pre>public class countingNumbers { public static void main(String[] args) { int userNumber = 10; int timesTen = userNumber *10; int divideTen = userNumber /10; int minusTwelve = userNumber -12; System.out.println(userNumber); System.out.println(timesTen); etc } }</pre>

High cohesion:
<pre>public class countingNumbers { public static void main(String[] args) { userNumber = takeInUserNumber(); multiplyTen(userNumber); divideTen(userNumber); ...etc } public static int multiplyTen(int userNumber) { int multiTen = userNumber *10; return multiTen; } Public static ... { ... } }</pre>

Coupling:

Coupling is a measure of how independent the functions or methods of a system are from each other^[2]. A tightly coupled system means that functions depend on each other so if you change one, it might break many other functions. Tight coupling leads to a decrease in reusability of code

due to the lack of generalisation. It also leads to difficulty in testing because each function can't be tested individually but instead needs the coupled functions to be included in the test.

A loosely coupled system will have functions which can function without the other functions in the system. This leads to easier testing, increased reusability of code, decreased maintenance and an easier development process.

For example:

Tightly coupled:
<pre>public class mathsyThing { public static void main(String[] args) { int x = functionForUserInput(); int y = functionForAddingOne(x); int z = functioForAddingTwo(y); int a = functionForSubtractingThree(z); } }</pre>

Loosely coupled:
<pre>public class mathsyThing { public static void main(String[] args) { int x = functionForUserInput(); int y = functionForAddingOne(x); int z = functionForAddingTwo(x); int a = functionForSubtractingThree(x); } }</pre>

Data Hiding:

Data hiding is a technique used in object-oriented programming to hide the details of an object^[3].

An object of a class needs to have access to data it is going to use but it does not need access to the design details of the system.

For example, a car (class) is split into parts (functions) and one part is a gear box. The gear box doesn't need to know the design details of the car; it only needs to know its own details. This means that if the manufacturers want to offer another choice of gear box, they don't need to redesign the whole car. Instead, they just swap the gear box module for the desired one, keeping the larger design the same.

In the case of a software program, if your functions are tied into the system design, the design must be changed every time you want to swap out functions. This system will also be vulnerable because anyone inspecting or doing maintenance on one function will also have access to the design of the whole system. This means that the wrong people might be able to gain access to data you don't want them to have.

In short, data hiding increases the flexibility of a system in the same way that loose coupling does, it reduces complexity by limiting dependency and it prevents unintended changes to objects when the system is changed.

Information Hiding:

Information hiding is used to prevent system design change[4]. If the design of the system is hidden (not accessible by the functions), some code can't be changed. This means that the functions don't get broken when the design of the system changes and vice versa. This leads to greater ease of use by users because they only see what they need.

Information is derived from data. An example would be knowing a person's date of birth (data) will tell you their age (information) or knowing their gender (data) and date of birth (data) might tell you the probability of them having a car accident (information). If the data in a system is hidden, the information will also be hidden because you can't derive information from nothing.

Importance:

A system where there is low cohesion, tight coupling and poor data and information hiding might look like this:

<pre>public class doingMathsThings { public static void main(String[] args) { String secretData = "Don't call me: 05559559559"; int x = functionForUserInput(); int y = functionForAddingOne(x); int z = functioForAddingTwo(y); int a = functionForSubtractingThree(z); int b = x * 13000 + 11 * 14; int c = x * x * x * 11; int d = functionThatFindsSquare(x, y); int e = addEleven(secretData, d); } public static int functionThatFindsSquare(int x, int y) { int plusOne = functionForAddingOne(x); int plusTwo = functionForAddingTwo(y); int somethingElse = plusOne + plusTwo; double anotherThing = somethingElse / 30; int square = x * x; return square; } public static int addEleven(String secretData, int d) { int x = d + 11; return x; } }</pre>	<pre><-- Functions take in return from previous functions where x would work. If functionForAddingOne is changed, it breaks all following functions. <-- Main is cluttered and intent is unclear. <-- Function takes in unneeded parameters. <-- Function does more than one thing. <-- Function is not reusable. <-- Testing will require multiple functions. <-- Function takes in unneeded parameter. <-- Everyone now has access to this secret data.</pre>
--	--

It is very clear that this program is difficult to read, pointlessly complicated, susceptible to change and not secure. In contrast, a program with better cohesion, coupling and data and information hiding might look like this:

<pre> public class doingMathsThings { private String secrets; private String moreSecrets; public static void main(String[] args) { int a = userInput(); int b = addTen(a); int c = multiplyTen(a); double d = squareRoot(a); } public static int addTen(int a) { int aTen = a + 10; return aTen; } public static void tellSecrets(String secrets) { String secretWord = "My word is: " + secrets; } public static void anotherSecret(String secrets, String moreSecrets) { String secretSentence = "My sentence is: " + secrets + " with " + moreSecrets; } } </pre>	<pre> <-- Private variable is hidden. It can't be accessed from outside the class doingMathsThings. <-- Functions don't all rely on each other. <-- Functions in main tell a story. <-- Public function is not hidden. It can be accessed from anywhere. It is also generalised and can be reused. <-- Public function is not hidden. It is used to access the private string 'secrets'. 'secrets' is hidden. <-- This function does not interfere with the private, hidden information so more functions with different functionalities can be added without changing the system design. </pre>
---	--

Conclusion:

To conclude, the use of high cohesion, loose coupling and data and information hiding is beneficial to software development because it produces a more robust and secure system while minimising the effort required to build and maintain the system.

References:

- [1] U. S. Poornima and V. Suma, "Impact of multiple inheritance on cohesion complexity in software design," in *2016 International Conference on Inventive Computation Technologies (ICICT)*, 2016, pp. 1–4.
- [2] R. Silva and H. Costa, "Graphical and statistical analysis of the software evolution using coupling and cohesion metrics — An exploratory study," in *2015 Latin American Computing Conference (CLEI)*, 2015, pp. 1–9.
- [3] "What is Data Hiding? - Definition from." [Online]. Available: <https://www.techopedia.com/definition/14738/data-hiding>. [Accessed: 13-Dec-2018].
- [4] "What is Information Hiding? - Definition from Techopedia." [Online]. Available: <https://www.techopedia.com/definition/3814/information-hiding>. [Accessed: 13-Dec-2018].

Bibliography (Code Part 2):

```
package MainForPart2;

import java.util.Scanner;

public class Assignment1Task2
{
    private static Scanner input = new Scanner(System.in);
    //-----MAIN
    public static void main(String[] args)
    {
        //MAIN VARIABLES:
        char[][] grid;
        char playerChar='m';
        boolean gameOver=false;
        String playerX="X", playerO="O";

        System.out.println();
        System.out.println("-----CONNECT 4-----");
        System.out.println();

        //FUNCTION TO INPUT PLAYERS:
        for (int i=0; i<2; i++)
        {
            if (i==0)
            {
                playerChar = 'X';
                playerX = createPlayers(playerChar);
            }
            else
            {
                playerChar = 'O';
                playerO = createPlayers(playerChar);
            }
        }
        //FUNCTION TO CREATE GRID:
        grid = createGrid();
        //FUNCTION TO RUN GAME LOOP:
        runGame(grid, playerChar, gameOver, playerX, playerO);
    }
    //-----MAIN
    //
```

```

//
//-----CREATE PLAYERS
public static String createPlayers(char playerChar)
{
    //FUNCTION TO TAKE USER INPUT NAMES
    //FUNCTION VARIABLES:
    String playerName;

    System.out.println("Welcome! Player " + playerChar + ", please enter your name:");
    playerName = input.nextLine();

    System.out.println(playerChar + ": " + playerName);
    return playerName;
}
//-----CREATE PLAYERS
//
//
//-----CREATE GRID
public static char[][] createGrid()
{
    //FUNCTION TO CREATE GRID
    //FUNCTION VARIABLES:
    char[][] grid = new char[7][7];

    for (int i = 0; i < grid.length; i++)
    {
        for (int j = 0; j < grid[i].length; j++)
        {
            grid[i][j] = '-';
        }
    }
    return grid;
}
//-----CREATE GRID
//
//
//-----RUN GAME
private static void runGame(char[][] grid, char playerChar, boolean gameOver, String
playerX, String playerO)
{
    //FUNCTION TO RUN GAME LOOP
    //FUNCTION VARIABLES:
    int column;
    int gameCounter =0;

```

```

int choice;
int bombNo0=1, bombNoX=1, blitzNo0=2, blitzNoX=2;
int[] special;

//GAME LOOP:
do
{
    //GAME COUNTER FOR TAKING TURNS:
    gameCounter++;
    if (gameCounter % 2 == 0)
    {
        playerChar = 'O';
    } else
    {
        playerChar = 'X';
    }
    //CALL FUNCTION TO PRINT GRID:
    printGrid(grid);
    //SPECIAL COUNTER CHECK AND UPDATE:
    special = specialCounter(bombNo0, bombNoX, blitzNo0, blitzNoX);
    System.out.println();
    System.out.println("Specials X :: bomb: " + special[1] + ", blitz: " + special[3]);
    System.out.println("Specials O :: bomb: " + special[0] + ", blitz: " + special[2]);
    System.out.println();
    //MOVE CHOICE INPUT:
    do
    {
        System.out.println(playerChar + "'s move");
        System.out.println("Select 'n' for a normal move, 'b' for a blitz, 't' for a
time bomb or '-' to rage quit:");
        choice = input.next().charAt(0);
    } while (choice != 'n' && choice != 'b' && choice != 't' && choice != '-');
    //MOVE SWITCH:
    switch (choice)
    {
        case 'n':
            System.out.println("Normal move selected");
            //CALL FUNCTION TO TAKE A TURN:
            column = selectColumn(playerChar);
            //CALL FUNCTION TO DROP COUNTER:
            dropCounterCaller(grid, column, playerChar);
            //CALL FUNCTION TO CHECK FOR TIME BOMBS:
            bombTimer(grid);
            //CALL FUNCTION TO PRINT GRID:

```

```

        //printGrid(grid);
        //CALL FUNCTION TO CHECK FOR WINS:
        gameOver = checkWin(grid, gameOver, playerChar, playerO, playerX);
        //CALL FUNCTION TO CHECK FOR DRAW:
        gameOver = checkDraw(gameOver, grid);
        break;

    case 'b':
        System.out.println("Blitz selected");
        //CHECK BLITZ COUNTER:
        if ((playerChar == 'X' && blitzNoX < 1)|| (playerChar == 'O' && blitzNoO
<1))

        {
            System.out.println("You don't have a blitz to use.");
            System.out.println("Miss a turn for pushing your luck.");
            break;
        }
        //REDUCE BLITZ COUNTER:
        if (playerChar == 'X')
        {
            blitzNoX--;
        }
        else
        {
            blitzNoO--;
        }
        //CALL FUNCTION TO TAKE A TURN:
        column = selectColumn(playerChar);
        //CALL FUNCTION TO DROP COUNTER:
        blitzCaller(grid, column);
        //CALL FUNCTION TO CHECK FOR TIME BOMBS:
        bombTimer(grid);
        //CALL FUNCTION TO PRINT GRID:
        //printGrid(grid);
        //CALL FUNCTION TO CHECK FOR WINS:
        gameOver = checkWin(grid, gameOver, playerChar, playerO, playerX);
        //CALL FUNCTION TO CHECK FOR DRAW:
        gameOver = checkDraw(gameOver, grid);
        break;

    case 't':
        System.out.println("Time bomb selected");
        //CHECK BLITZ COUNTER:
        if ((playerChar == 'X' && bombNoX < 1)|| (playerChar == 'O' && bombNoO <1))

```

```

        {
            System.out.println("You don't have a bomb to use.");
            System.out.println("Miss a turn for unsuccessfully starting a war.");
            break;
        }
        //REDUCE BOMB COUNTER:
        if (playerChar == 'X')
        {
            bombNoX--;
        }
        else
        {
            bombNoO--;
        }
        //CALL FUNCTION TO TAKE A TURN:
        column = selectColumn(playerChar);
        //CALL FUNCTION TO DROP COUNTER:
        bombCaller(grid, column);
        //CALL FUNCTION TO CHECK FOR TIME BOMBS:
        bombTimer(grid);
        //CALL FUNCTION TO PRINT GRID:
        //printGrid(grid);
        //CALL FUNCTION TO CHECK FOR WINS:
        gameOver = checkWin(grid, gameOver, playerChar, playerO, playerX);
        //CALL FUNCTION TO CHECK FOR DRAW:
        gameOver = checkDraw(gameOver, grid);
        break;

    case '\n':
        System.out.println("QUITTING???");
        System.out.println("...");
        System.out.println("...");
        System.out.println("...");
        System.out.println("...PATHETIC...");
        break;
    }
} while (!gameOver && choice != '\n');
}
//-----RUN GAME
//
//
//-----SPECIAL
COUNTER~~~~~EDITING
    public static int[] specialCounter(int bombNoX, int bombNoO, int blitzNoO, int blitzNoX)

```

```

{
    //FUNCTION TO KEEP TRACK OF NUMBER OF SPECIAL MOVES
    //FUNCTION VARIABLES:
    int[] specials = new int[4];

    specials[0] = bombNoO;
    specials[1] = bombNoX;
    specials[2] = blitzNoO;
    specials[3] = blitzNoX;

    return specials;
}
//-----SPECIAL
COUNTER~~~~~EDITING
//
//
//-----PRINT GRID
private static void printGrid(char[][] grid)
{
    //FUNCTION TO PRINT GRID
    for (int i = 0; i < grid.length; i++)
    {
        for (int j = 0; j < grid[i].length; j++)
        {
            System.out.print(grid[i][j] + " ");
        }
        System.out.println();
    }
    System.out.println();
}
//-----PRINT GRID
//
//
//-----SELECT COLUMN
public static int selectColumn(char playerChar)
{
    //FUNCTION TO SELECT COLUMN
    //FUNCTION VARIABLES:
    int column=-10;

    System.out.println(playerChar + ": Please select a column number:");
    do
    {
        System.out.println("(Positive integer from 1 to 7)");
    }
}

```

```

        if(input.hasNextInt())
        {
            column=input.nextInt()-1;
        }
        else
        {
            input.nextLine();
        }
    } while (column<0 || column>6);
    return column;
}
//-----SELECT COLUMN
//
//
//-----DROP COUNTER
private static void dropCounterCaller(char[][] grid, int column, char playerChar)
{
    //FUNCTION TO INSERT THE COUNTER INTO THE GRID
    switch (column)
    {
        case 0:
            dropCounter(grid, column, playerChar);
            break;
        case 1:
            dropCounter(grid, column, playerChar);
            break;
        case 2:
            dropCounter(grid, column, playerChar);
            break;
        case 3:
            dropCounter(grid, column, playerChar);
            break;
        case 4:
            dropCounter(grid, column, playerChar);
            break;
        case 5:
            dropCounter(grid, column, playerChar);
            break;
        case 6:
            dropCounter(grid, column, playerChar);
            break;
    }
}

```

```

private static void dropCounter(char[][] grid, int column, char playerChar)
{
    //IF TOP OF COLUMN IS EMPTY:
    if (grid[0][column] == '-')
    {
        //ADD COUNTER TO TOP:
        grid[0][column] = playerChar;
        //CHECK NEXT ROWS FOR EMPTY:
        for (int i = 1; i < 7; i++)
        {
            if (grid[i][column] == '-')
            {
                grid[i][column] = playerChar;
                grid[i - 1][column] = '-';
            }
        }
    } else //NOT EMPTY:
    {
        System.out.println();
        System.out.println("The counter drops to the floor.");
        System.out.println();
    }
}

//-----DROP COUNTER
//
//
//-----BLITZ
private static void blitzCaller(char[][] grid, int column)
{
    //FUNCTION TO INSERT THE BLITZ INTO THE GRID
    switch (column)
    {
        case 0:
            blitz(grid, column);
            break;
        case 1:
            blitz(grid, column);
            break;
        case 2:
            blitz(grid, column);
            break;
        case 3:
            blitz(grid, column);
            break;
    }
}

```

```

        case 4:
            blitz(grid, column);
            break;
        case 5:
            blitz(grid, column);
            break;
        case 6:
            blitz(grid, column);
            break;
    }
}

private static void blitz(char[][] grid, int column)
{
    //EMPTY OUT COLUMN:
    for (int i = 1; i < 7; i++)
    {
        grid[i][column] = '-';
    }
}

//-----BLITZ
//
//
//-----BOMB DROP
private static void bombCaller(char[][] grid, int column)
{
    //FUNCTION TO INSERT THE BLITZ INTO THE GRID
    switch (column)
    {
        case 0:
            bomb(grid, column);
            break;
        case 1:
            bomb(grid, column);
            break;
        case 2:
            bomb(grid, column);
            break;
        case 3:
            bomb(grid, column);
            break;
        case 4:
            bomb(grid, column);
            break;
    }
}

```

```

        case 5:
            bomb(grid, column);
            break;
        case 6:
            bomb(grid, column);
            break;
    }
}

public static void bomb(char[][] grid, int column)
{
    //IF TOP OF COLUMN IS EMPTY:
    if (grid[0][column] == '-')
    {
        //ADD COUNTER TO TOP:
        grid[0][column] = 'c';
        //CHECK NEXT ROWS FOR EMPTY:
        for (int i = 1; i < 7; i++)
        {
            if (grid[i][column] == '-')
            {
                grid[i - 1][column] = '-';
                grid[i][column] = 'c';
            }
        }
    } else //NOT EMPTY:
    {
        System.out.println();
        System.out.println("The bomb drops to the floor.");
        System.out.println("How careless.");
        System.out.println();
    }
}

//-----BOMB DROP
//
//
//-----BOOM!!!

public static char[][] bombBlow(char[][] grid)
{
    //FUNCTION TO BLOW UP THE BOMBS
    System.out.println();
    System.out.println("*tick*...");
    System.out.println();
    System.out.println("...*Tick*...");
}

```

```
System.out.println();
System.out.println(".....*TICK*...");
System.out.println();
System.out.println("*BOOM!*");
//SCAN THROUGH GRID TO FIND READY BOMBS:
for (int row=0; row<7; row++)
{
    for (int col=0; col<7; col++)
    {
        if (grid[row][col] == '@')
        {
            //CLEAR 3x3 CELLS:
            grid[row][col] = '-';
            if (col > 0)
            {
                grid[row][col - 1] = '-';
            }
            if (col < 6)
            {
                grid[row][col + 1] = '-';
            }
            if (row > 1)
            {
                grid[row - 1][col] = '-';
            }
            if (row < 6)
            {
                grid[row + 1][col] = '-';
            }
            if ((row > 1)&&(col > 0))
            {
                grid[row - 1][col - 1] = '-';
            }
            if ((row > 1)&&(col < 6))
            {
                grid[row - 1][col + 1] = '-';
            }
            if ((row < 6)&&(col > 1))
            {
                grid[row + 1][col - 1] = '-';
            }
            if ((row < 6)&&(col < 6))
            {
                grid[row + 1][col + 1] = '-';
            }
        }
    }
}
```

```

        }
    }
}

//DROP FLOATING COUNTERS:
//CHECK NEXT ROWS FOR EMPTY:
for (int count=0; count<7; count++)
{
    //DROP COUNTERS DOWN:
    for (int row=0; row<6; row++)
    {
        for (int col=0; col<7; col++)
        {
            if (grid[row][col] != '-' && grid[row+1][col] == '-')
            {
                //IF FLOATER FOUND,
                //SET NEXT ROW TO COUNTER,
                grid[row+1][col] = grid[row][col];
                //REMOVE FLOATER,
                grid[row][col] = '-';
            }
        }
    }
}

return grid;
}

//-----BOOM!!!
//
//
//-----BOMB TIMER
public static char[][] bombTimer(char[][] grid)
{
    //FUNCTION TO CHECK FOR TIME BOMBS AND STEP THROUGH THE TICKING
    //SCAN GRID FOR BOMBS:
    for (int row=0; row<7; row++)
    {
        for (int col=0; col<7; col++)
        {
            if (grid[row][col] == '@')
            {
                //IF READY TO BLOW, CALL BLOW FUNCTION:
                bombBlow(grid);
            }
            else if (grid[row][col] == 'a')

```

```

        {
            //IF MID, ADD TICK:
            grid[row][col] = '@';
            System.out.println();
            System.out.println("...tick*...");
        }
        else if (grid[row][col] == 'c')
        {
            //IF FRESH, ADD TICK:
            grid[row][col] = 'a';
            System.out.println();
            System.out.println("...tick*...");
        }
    }
}

return grid;
}

//-----BOMB TIMER
//
//
//-----CHECK WIN
public static boolean checkWin(char[][] grid, boolean gameOver, char playerChar, String
player0, String playerX)
{
    //FUNCTION TO CHECK FOR WIN
    gameOver = rowWin(grid, gameOver);
    gameOver = colWin(grid, gameOver);
    gameOver = diagWin(grid, gameOver);

    if (gameOver)
    {
        System.out.println("WIN");
        System.out.println("Player " + playerChar + " is the winner!");
        if (playerChar == 'X')
        {
            System.out.println("Congrats, " + playerX + "!");
        }
        else if (playerChar == 'O')
        {
            System.out.println("Congrats, " + player0 + "!");
        }
    }
    return gameOver;
}

```

```

public static boolean rowWin(char[][] grid, boolean gameOver)
{
    for (int row=0; row<grid.length; row++)
    {
        for (int col=0; col<4; col++) //row<4 BECAUSE WE ONLY NEED TO SCAN THE TOP TO GET
THE LINE
        {
            //ALL POSSIBLE ROW WINS:
            if (((grid[row][col]!='-')
                &&(grid[row][col]==grid[row][col+1])
                &&(grid[row][col]==grid[row][col+2])
                &&(grid[row][col]==grid[row][col+3])))
            {
                gameOver=true;
            }
        }
    }
    return gameOver;
}

```

```

public static boolean colWin(char[][] grid, boolean gameOver)
{
    for (int col=0; col<grid.length; col++)
    {
        for (int row=0; row<4; row++) //AGAIN row<4
        {
            //ALL POSSIBLE ROW WINS:
            if (((grid[row][col]!='-')
                &&(grid[row][col]==grid[row+1][col])
                &&(grid[row][col]==grid[row+2][col])
                &&(grid[row][col]==grid[row+3][col])))
            {
                gameOver=true;
            }
        }
    }
    return gameOver;
}

```

```

public static boolean diagWin(char[][] grid, boolean gameOver)
{
    //DIAGONAL DOWN \:
    for (int row=0; row<4; row++) //AGAIN row<4

```

```

    {
        for (int col=0; col<4; col++)
        {
            //ALL POSSIBLE ROW WINS:
            if (((grid[row][col]!='-')
                &&(grid[row][col]==grid[row+1][col+1])
                &&(grid[row][col]==grid[row+2][col+2])
                &&(grid[row][col]==grid[row+3][col+3])))
            {
                gameOver=true;
            }
        }
    }
    //DIAGONAL UP /:
    for (int row=0; row<4; row++)
    {
        for (int col=6; col>2; col--)
        {
            //ALL POSSIBLE ROW WINS:
            if (((grid[row][col]!='-')
                &&(grid[row][col]==grid[row+1][col-1])
                &&(grid[row][col]==grid[row+2][col-2])
                &&(grid[row][col]==grid[row+3][col-3])))
            {
                gameOver=true;
            }
        }
    }
    return gameOver;
}
//-----CHECK WIN
//
//
//-----CHECK DRAW
public static boolean checkDraw(boolean gameOver, char[][] grid)
{
    //FUNCTION TO CHECK FOR DRAW
    //FUNCTION VARIABLES:
    int counter = 0;
    //LOOP THROUGH GRID, COUNTING FOR '-'s:
    for (int i=0; i<6; i++)
    {
        for (int j=0; j<6; j++)
        {

```

```
        if (grid[i][j] == '-')
        {
            counter++;
        }
    }
}
if (counter == 0)
{
    System.out.println("DRAW");
    gameOver = true;
}
return gameOver;
}
//-----CHECK DRAW
}
```

Part 2

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

61/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

