

Java Programming and Testing

Part 2

by Rhys Harding

FILE	12016_RHYS_HARDING_JAVA_PROGRAMMING_AND_TESTING_PART_2_159704_1048285120.DOCX		
TIME SUBMITTED	18-JAN-2019 11:29PM (UTC+0000)	WORD COUNT	6577
SUBMISSION ID	99458738	CHARACTER COUNT	34682

Programming and Software Testing
ACCA7000

**The Importance to Software Development of
High Cohesion, Loose Coupling, Data Hiding
and Information Hiding**

Rhys Harding
1809709

Contents

Introduction	2
Cohesion and Coupling	2
• Functional Cohesion.....	2
• Sequential Cohesion	2
• Communicational Cohesion	2
• Procedural Cohesion	2
• Temporal Cohesion	3
• Logical Cohesion	3
• Coincidental Cohesion	3
• Data Coupling.....	3
• Stamp Coupling.....	3
• Control Coupling	3
• External Coupling	3
• Common Coupling.....	3
• Content Coupling	3
Data Hiding and Information Hiding.....	4
Findings and Conclusion	4
References	5

Introduction

In Software Development, for software systems that are complex, often the system will be developed using a programming technique called Object Orientated Programming (OOP) [1]. OOP is used to handle more complex systems as oppose to what is called Procedural Programming, which includes multiple functions that perform on variables all in one piece of code [2]. As systems become more complex, the functions become inter-dependent on each other, so much so that when one part of code is changed in one function, other functions that depend upon it proceed to break [3]. OOP introduces structure to a software system through Classes and Objects [2]. Classes can be considered like blueprints and hold all the functionality and characteristics of a particular entity [1], [2]. An Object is an instance of a class, inheriting the functionality and properties of that class whilst possibly also having some that are specific to that instance [1], [2], [4]. This technique of compartmentalising is called Encapsulating and is a fundamental concept of OOP.

This report will detail the importance of high cohesion, loose coupling and data hiding. Further research will be carried out on online databases, journals and books in order to conduct the research necessary to report the findings and demonstrate an understanding of the given OOP terminology.

Cohesion and Coupling

Cohesion and coupling refer to a measure that is used to determine how the modules of an OOP system are structured. Cohesion measures the relationship between the elements of a module for a categorised system, it is the functional activity of how 'together' these elements are [5]. It is important for a system to have elements that have strong relationships and functionality, so that there is high connectivity and hence a fast and efficient executable program [6], [5], [7]. There are many types of cohesion and each are considered to attribute whether a system has high or low cohesion:

- **Functional Cohesion** refers to essential elements being grouped together 'contributing to a specific function' [8, p. 36], making the system more functionally related. Functional Cohesion is often associated with high cohesion [9].
- **Sequential Cohesion** refers to how the data within the module is handled by the different elements. Those elements are grouped together and the output for elements becomes the input for another element [8], producing high connectivity [9].
- **Communicational Cohesion** is when elements function on the same input data or when they contribute towards the same output data [8] and are so grouped together [9].
- **Procedural Cohesion** is when elements are '*grouped together*' because they are 'executed in order' [9]. These sorts of cohesion are not generally desirable as they are weakly connected [8] and are difficult to reuse [9].

- **Temporal Cohesion** refers to when elements are grouped together by their involvement in a particular task and within a particular time [9], [8].
- **Logical Cohesion** is when elements are not doing the same thing but can be categorised logically as doing similar things [9], [8].
- **Coincidental Cohesion** is when there is no relationship between the elements being grouped [8] apart from that they are being grouped. This is the lowest form of cohesion [8], [9].

A idealistic system will have high cohesion and a developer creating software using OOP should try to attain good practices of high cohesion [5]. Typically, the types that are placed high on the list above, like functional cohesion and sequential cohesion should be utilised, whilst avoiding forms like procedural, temporal, logical and coincidental as these prevent the system from having good connectivity [6]. Although a system should avoid lower forms of cohesion than the superior forms, circumstances often govern the extent to which cohesion types can be implemented [6], [5], [10]. A developer should make efforts to make the system as cohesive as possible [1], [10].

Coupling is a measure of the degree of interconnecting elements and how dependant they are upon each other [5], [7], [11]. It is important for a system to have elements that do not heavily depend on other elements, defined as loosely coupled [12] as tightly coupled systems can suffer from adverse effects to multiple parts of their system when one particular part defects or is changed [11]. There are multiple types of coupling, which attribute to whether a system has tight or loose coupling:

- **Data Coupling** is the most desirable and is considered as a loose coupling measure. It refers to the only dependency between modules is the passing of data from one to another [6], [9], [13].
- **Stamp Coupling** is the grouping together of modules that share 'data structure' [9]. This sort of coupling, however, tends to exhibit adverse effects when parts of the module is changed leading to the overall functionality of that module [6], [14].
- **Control Coupling** refers to when a module controls the flow of another [11], passing through information on what to do [9].
- **External Coupling** is when modules are grouped together, based on the fact that the software depends on particular, external hardware [9].
- **Common Coupling** is referred to when modules share 'global data' [9], [11]. This type of coupling introduces problems when changes are made that could possibly change the whole program or the way it executes [9].
- **Content Coupling** can be considered as the least desirable type of coupling. It is the type of coupling implies that modules can change or use the data from another and where the control is passed from module to module [9], [13].

A idealistic software system will have loose coupling [11], a developer should aim to incorporate coupling types like Data Coupling or Stamp Coupling whilst avoiding Content and Common Coupling [1], [13]. A system that has loose coupling will be a lot easier to test, to change and to reuse the different modules within [11]. Tightly coupled modules become so dependent on one another that the building and understanding of the system becomes incredibly complex and time consuming. Also, any changes made to one module can consequently change the way another may function [11]. In similarity to cohesion, a developer should aim to achieve the best possible types of coupling for the given situation.

Data Hiding and Information Hiding

Encapsulation in OOP is also referred to as the way in which modules and elements are constructed so that the structure and properties of a particular element are only accessed by that element [2], [15], [10]. Data Hiding is a technique used to hide the particular attributes and functionalities of an element [12], [10]. Data Hiding helps enforce encapsulation, with the aid of *'getter'* and *'setter'* functions [13]. These are used to help protect the attributes of an element, whilst controlling what is exactly being accessed, preventing other elements from handling the internal data [13].

It is important for a system to incorporate data hiding [13], as it ensures that the system is in a reliable and stable state, as internal data cannot be manipulated, [6], [12], which could result in inconsistent code [1], [2], [13], [16]. Furthermore, developing code can be written with the assurance that new code will not tamper with existing code, and so it will avoid breaking code this way. Additionally, testing becomes easier and more effective as the functions that perform on hidden data can be extensively tested, assuring that the data is handled as expected [11].

Information hiding is a similar technique that is adopted. This mechanism ensures that the attributes and functionalities of an element should be hidden from the end-user [13]. In addition, the attributes of an element should only be accessed by the element itself and its functionalities. An element that adopts this type of hiding uses *'private'* attributes and methods to ensure that no other elements can handle them [1]. Information hiding makes it easier for a developer to amend a program whilst avoiding changes to unexpected parts of the system [10], [13].

Findings and Conclusion

Research conducted from various sources through online databases, journals and books has facilitated the understanding of high cohesion, loose coupling, data hiding and information hiding and the importance of these measures and concepts. High cohesion and loose coupling are both measures of a software system that are considered that of a good system. They indicate the high strength of relationships between modules that help achieve greater connectivity, and the loose inter-dependencies between modules and elements that help avoid rippling effects to a system as a result of a change of one element. Data hiding is the act of protecting the inner workings of an element and is achieved through encapsulation and information hiding, packaging the elements of system so that they remain hidden to each other and protect the integrity of an elements attributes and functionalities.

References

- [1 P. Tilmanis and G. White , "Further Programming in Java," 2006 RMIT University, Melbourne, 2006.
- [2 K. S. Chauveau, "Working with C++," in *Proceedings of NORTHCON'93 Electrical and Electronics Convention*, Portland, 1993.
- [3 R. Buyya, S. T. Selvi and X. Chu, Object-oriented Programming with Java: Essentials and Applications, New Delhi: Tata McGraw Hill , 2009.
- [4 Oracle, "The Java™ Tutorials," 1994-2019. [Online]. Available:
] <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html>. [Accessed 3 January 2019].
- [5 R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, New Jersey: Pearson Education Inc., 2009.
- [6 TutorialsPoint, "Cohesion and Coupling," 2018. [Online]. Available:
] https://www.tutorialspoint.com/software_engineering_basics/cohesion_and_coupling.asp?fbclid=IwAR0ez4T3NErr3KW7_Pu2SmJz4bN7bh62AJiZqrQV0M2aOfSqqSO40vYCbgo. [Accessed 8 January 2019].
- [7 TutorialsPoint, "Coupling in Java," 2018. [Online]. Available:
] <https://www.tutorialspoint.com/Coupling-in-Java?fbclid=IwAR2bUjbjIEOeNAOfPYmITxYS1PCvU7obDUNew70cpORSW4U1Q7xO-n1Plt8>. [Accessed 7 Jan 2019].
- [8 A. Lakhotia, "Rule-based approach to computing module cohesion," in *ICSE '93 Proceedings of the 15th international conference on Software Engineering*, Baltimore, Maryland, 1993.
- [9 A. Chakrabort, "TutorialsPoint," Tutorials Point (India) Pvt. Ltd., 2018. [Online]. Available:
] https://www.tutorialspoint.com/software_engineering_basics/cohesion_and_coupling.asp?fbclid=IwAR0ez4T3NErr3KW7_Pu2SmJz4bN7bh62AJiZqrQV0M2aOfSqqSO40vYCbgo. [Accessed 9 January 2019].
- [1 D. SAMANTA, OBJECT-ORIENTED PROGRAMMING WITH C++ AND JAVA, New Delhi: PHI Learning Pvt. Ltd., 2006.
- [1 M. Hitz and B. Montazer, "Measuring Coupling and Cohesion In Object-Oriented Systems," Proc. Int'l Symp. Applied Corporate Computing, Monterrey, Mexico, 1995.
- [1 N. Ashrafi and H. Ashrafi, Object Oriented Systems Analysis and Design, Essex: Pearson Education Edition, 2014.
- [1 M. Weisfeld, The Object-Oriented Thought Process: ObjectOr Thought Process, 4 ed., New Jersey: Developer's Library, 2013.
- [1 H. M. Kaung, N. S. Kham and N. L. Thein, "To Visualize the Coupling among Modules," in 4] *IEEE Conferences*, Yangon, 2005.
- [1 TutorialsPoint, "Data Encapsulation in C++," 2019. [Online]. Available:
5] https://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm?fbclid=IwAR3jilwWYvcEenDXP8uwdZuYM5wa6ZhBRCmi54nQe2bRjq08hICBoRTnHiQ. [Accessed 10 January 2019].
- [1 K Steinbrugge, Frant, "Using C++ in an object oriented paradigm. A practical approach," 6] in *Proceedings of NORTHCON '94*, Seattle, 1994.

Connect 4 Code

```
1.  /*
2.   * To change this license header, choose License Headers in Project Properties.
3.   * To change this template file, choose Tools | Templates
4.   * and open the template in the editor.
5.   */
6.  package connect4;
7.
8.  import java.util.Scanner;
9.
10. /**
11.  *
12.  * @author Rhys Harding
13.  */
14. public class Connect4 {
15.
16.     private static Scanner stdin = new Scanner(System.in);
17.
18.     /**
19.      * @param args the command line arguments
20.      */
21.     public static void main(String[] args) {
22.         displayGraphic();
23.         System.out.println("-----Welcome to Connect4-----");
24.         displayRules();
25.
26.         String[] players = selectPlayers();
27.         char[][] board = createBoard(); // D&I a multidimensional array that will act as the game board.
28.         turnHandler(board, players[0], players[1]); // Calling the procedure that cycle through the game.
29.
30.     }
31.
32.     /**
33.      * A procedure that displays the Connect4 graphic.
34.      */
35.     public static void displayGraphic() {
36.         System.out.println("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\n");
37.         + "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXX\n"
38.         + "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXX XXX\n"
39.         + "XXX X XXX\n"
40.         + "XX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX X XXX\n"
41.         + "X XX XXX XX X X X X X X XXXX XXX XX XXX X XXX\n"
42.         + "X X XXX X XXXX X X X X X XXXX XXX XX XXX X XXX\n"
43.         + "X X XXXXX XXXX X X X X X XXXX XXXXXX X XXX XXXX X\n"
44.         + "X X XXXXX XXXX X X X X X XXX XXXXXX X XXX XXXX X\n"
45.         + "X X XXX X XXXX X XX X XX X XXXX XXX XX X X X\n"
46.         + "X XX XXX XX XXX X XXX X XX XXX X X\n"
47.         + "XX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXX X\n"
48.         + "XXX Mr. Dacey & Dr Bashford's Version X X X\n"
49.         + "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXX XXX\n"
50.         + "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXX\n"
51.         + "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
52.
53.     }
54.
55.     /**
56.      * A method that displays the rules in a format that is in line with overall
57.      * layout
58.      */
59.     public static void displayRules() {
```

```

60.         System.out.println("The object of the game is to match 4 of your counters in a connected lin
e,\n"
61.         + "either horizontally, diagonally or vertically.                \n"
62.         + "'-' denotes a free space.                                    \n"
63.         + "'X' denotes the first player's marker.                      \n"
64.         + "'O' denotes the second player's marker.                      \n"
65.         + "'*' denotes the placement of a 'time bomb' for player 1.      \n"
66.         + "'#' denotes the placement of a 'time bomb' for player 2.      \n"
67.         + "                                                                \n"
68.         + "Select a column for which you wish to drop you counter. The counter will \n"
69.         + "fall to the bottom most free space.                            \n"
70.         + "Each player has use of two special abilities which they can only use once:\n"
71.         + "                                                                \n"
72.         + "BLITZ    Activated by pressing 'b' followed by a column. The user can \n"
73.         + "clear any column of its existing markers.                             \n"
74.         + "TIME BOMB Activated by pressing 't' followed by a column. This places a \n"
75.         + "a 'time bomb' in the selected column which clear the surrounding\n"
76.         + "markers in 2 moves time                                                \n");
77.     }
78.
79.     /**
80.      * A method that takes the names of the players and which player goes first
81.      *
82.      * @return The names of the two players, in the order that they will play
83.      */
84.     public static String[] selectPlayers() {
85.         System.out.println("-----Player Selection-----");
86.         String[] playerNames = new String[2];
87.         System.out.println("X's will go first! Please enter the player names");
88.         playerNames[0] = readInputString("'X' Player name: ");
89.         playerNames[1] = readInputString("'O' Player name: ");
90.         System.out.printf("Ok, so %s will go first, followed by %s, who will go second.\r\n",
91.             playerNames[0],
92.             playerNames[1]);
93.         return playerNames;
94.     }
95.
96.     /**
97.      * A method that will create the empty board
98.      *
99.      * @return A 2 dimensional array representing a 6x7 connect4 board
100.     */
101.     public static char[][] createBoard() {
102.         char specialMoveCount = '1'; //specify amount of special moves available here.
103.         /**
104.          * A multidimensional array follows that will act as a standard 6x7
105.          * classic Connect4 board. The board has an added row that is intended
106.          * to be hidden from most functions and acts to hold the special move
107.          * counters and the time bomb countdowns.
108.          */
109.         char[][] emptyBoard = new char[7][7];
110.         for (int i = 0; i < 7; i++) {
111.             for (int j = 0; j < 7; j++) {
112.                 if (i == 6 && j < 4) { //hidden row
113.                     emptyBoard[i][j] = specialMoveCount;
114.                     /**
115.                      * Could have indicated here without the use of a variable
116.                      * but wanted to easily indicate using a variable so that
117.                      * any alterations to the special moves count could be
118.                      * changed with clarity.
119.                      */
120.                 } else {
121.                     emptyBoard[i][j] = '-';
122.                 }
123.             }
124.         }
125.         return emptyBoard;
126.     }

```

```

127.     }
128.
129.     /**
130.      * A procedure that will print the current board state to the console
131.      *
132.      * @param board current board state
133.      */
134.     public static void drawBoardCentralized(char[][] board) {
135.         System.out.println("-----Current Board State-----
136.         ---");
137.         System.out.println("          0 1 2 3 4 5 6");
138.         for (int x = 0; x < 6; x++) {
139.             System.out.print("          |"); //helps depicts a board structure i
140.             n the console
141.             for (int y = 0; y < 7; y++) {
142.                 System.out.print(board[x][y]);
143.                 if (y != 6) { //anywhere other than the last value of each row we wish to separate t
144.                     he values with a space to help depict the board
145.                     System.out.print(" ");
146.                 }
147.             }
148.             System.out.print("\r\n");
149.         }
150.         System.out.println("          [=====]"); // helps depicts a boa
151.         rd structure in the console
152.     }
153.
154.     /**
155.      * A procedure that cycles through the player turns, always checking for
156.      * wins or a draw, in which case, will stop the game and present the winner.
157.      *
158.      * @param board (char[][] The empty board.
159.      * @param player1 (String) The player who will go first.
160.      * @param player2 (String) The player who will go second.
161.      */
162.     public static void turnHandler(char[][] board, String player1, String player2) {
163.         System.out.println("-----Game Initialized-----
164.         ---");
165.         boolean turn = false; //will emulate the flipping of turns between each player.
166.         while (!isWin(board) && !isDraw(board)) { //checks before each turn, whether the game has be
167.             en won or drawn
168.             turn = !turn; //swaps the turn at each loop. For the first loop, will switch the turn v
169.             ariable to true to set the first turn as the first player
170.             drawBoardCentralized(board); //draws the board for the player to see before he/she selec
171.             ts their move
172.             if (turn) {
173.                 System.out.printf("%s, it is your turn.\r\n", player1);
174.                 board = makeMove(board, turn); //calls the method to make the move
175.             } else {
176.                 System.out.printf("%s, it is your turn.\r\n", player2);
177.                 board = makeMove(board, turn);
178.             }
179.         }
180.         drawBoardCentralized(board); //draws the board one last time once the while loop is broken i
181.         .e. the game has either been won or drawn
182.         if (isWin(board)) { //case win
183.             if (turn) {
184.                 System.out.printf("%s, you have won the game\r\n", player1);
185.             } else {
186.                 System.out.printf("%s, you have won the game\r\n", player2);
187.             }
188.         } else { //draw
189.             System.out.println("The game has ended in a draw!");
190.         }
191.         displayGraphic();
192.     }
193.
194.     /**
195.      * A method that will handle the move making process

```

```

187.      *
188.      * @param board (char[][] the latest version of the board
189.      * @param turn (boolean) the turn of the player
190.      * @return The board with a new move that has been made
191.      */
192.      public static char[][] makeMove(char[][] board, boolean turn) {
193.          boolean inputValid = false; //bool to ensure valid input
194.          timeBombCountdown(board); //to check if there are any active time bombs
195.          System.out.println("Please select which column you wish to drop your counter in.");
196.          do {
197.              int[] input = readInputMove("Column: ");
198.              switch (input[0]) { //input[0] comes from above function and indicates the move type
199.                  case 1: //Blitz
200.                      if (turn && charToInt(board[6][0]) > 0) { //if ability count allows
201.                          board[6][0] = intToChar(charToInt(board[6][0]) - 1); //one less use
202.                          input[1] = readInputColumn("Select the column you wish to Blitz: ");
203.                          blitzClear(board, input[1]);
204.                          inputValid = true;
205.                      } else if (!turn && charToInt(board[6][1]) > 0) {
206.                          board[6][1] = intToChar(charToInt(board[6][1]) - 1);
207.                          input[1] = readInputColumn("Select the column you wish to Blitz: ");
208.                          blitzClear(board, input[1]);
209.                          inputValid = true;
210.                      } else {
211.                          System.out.println("Special ability already used!");
212.                      }
213.                      break;
214.                  case 2: //TimeBomb
215.                      if (turn && charToInt(board[6][2]) > 0) {
216.                          board[6][2] = intToChar(charToInt(board[6][2]) - 1);
217.                          input[1] = readInputColumn("Select the column you wish to place the time bom
218.                          b in: ");
219.                          timeBombPlace(board, input[1], turn);
220.                          inputValid = true;
221.                      } else if (!turn && charToInt(board[6][3]) > 0) {
222.                          board[6][3] = intToChar(charToInt(board[6][3]) - 1);
223.                          input[1] = readInputColumn("Select the column you wish to place the time bom
224.                          b in: ");
225.                          timeBombPlace(board, input[1], turn);
226.                          inputValid = true;
227.                      } else {
228.                          System.out.println("Special ability already used!");
229.                      }
230.                      break;
231.                  case 0: //Standard Move
232.                      if (!isPossible(board, input)) { //ensure the selected move is possible
233.                          System.out.println("Column full! Please try a different column.");
234.                      } else { //move is possible
235.                          System.out.println("Column Accepted!");
236.                          inputValid = true;
237.                          dropCounter(board, input[1], turn); //once the loop is broken, indicating th
238.                          e move is valid, a method is called to drop the counter and make the move requested
239.                      }
240.                  }
241.              } while (!inputValid);
242.              return board;
243.          }
244.      }
245.      /**
246.       * A procedure that checks if there is an active time bomb in the board and
247.       * handles the countdown and the clearing of counters
248.       *
249.       * @param board latest board
250.       */
251.      public static void timeBombCountdown(char[][] board) {
252.          if (charToInt(board[6][4]) > 0) { //if countdown active
253.              board[6][4] = intToChar(charToInt(board[6][4]) - 1); //pass the turn
254.              if (charToInt(board[6][4]) == 0) { //if last count of countdown
255.                  timeBombClear(board, true);

```

```

253.         fallCounters(board);
254.     }
255. }
256. if (charToInt(board[6][5]) > 0) {
257.     board[6][5] = intToChar(charToInt(board[6][5]) - 1);
258.     if (charToInt(board[6][5]) == 0) {
259.         timeBombClear(board, false);
260.         fallCounters(board);
261.     }
262. }
263. }
264.
265. /**
266.  * A method that checks to see if the attempted move is legal.
267.  *
268.  * @param board (char[][] ) The board.
269.  * @param y (int) The column coordinate.
270.  * @return True if legal, false if illegal.
271.  */
272. public static boolean isPossible(char[][] board, int[] y) {
273.     boolean isPossible = false;
274.     for (int x = 0; x < 6; x++) { //iterate through rows of selected column
275.         if (board[x][y[1]] == '-') { // if empty
276.             isPossible = true;
277.         }
278.     }
279.     return isPossible;
280. }
281.
282. /**
283.  * A method that writes the new move onto the board
284.  *
285.  * @param board latest version of the board
286.  * @param y the column selected
287.  * @param turn player whose turn it is
288.  */
289. public static void dropCounter(char[][] board, int y, boolean turn) {
290.     for (int x = 0; x < 6; x++) {
291.         if (board[5 - x][y] == '-') { //if space unoccupied, starting from bottom
292.             if (turn) {
293.                 board[5 - x][y] = 'X';
294.                 break;
295.             } else {
296.                 board[5 - x][y] = 'O';
297.                 break;
298.             }
299.         }
300.     }
301. }
302.
303. /**
304.  * A method that performs the special ability 'blitz'. All existing markers
305.  * in specified column will be omitted.
306.  *
307.  * @param board current board
308.  * @param y chosen column
309.  */
310. public static void blitzClear(char[][] board, int y) {
311.     for (int x = 0; x < 6; x++) {
312.         //Here, if a Blitz is used over a timebomb in time, it cancels out the bomb
313.         if (board[x][y] == '*') {
314.             board[6][4] = '0'; //stops the countdown
315.         }
316.         if (board[x][y] == '#') {
317.             board[6][5] = '0';
318.         }
319.         board[x][y] = '-'; //reduce to empty spaces
320.     }
321. }

```

```

322.
323. /**
324.  * A method that performs the special ability 'Time Bomb'. Places a * / #
325.  * into the board
326.  *
327.  * @param board latest board
328.  * @param y chosen column
329.  * @param turn
330.  */
331. public static void timeBombPlace(char[][] board, int y, boolean turn) {
332.     int timeBombTimer = 2;
333.     if (turn) { //for *
334.         board[6][4] = intToChar((timeBombTimer) + 1);
335.         for (int x = 0; x < 6; x++) {
336.             if (board[5 - x][y] == '-')
337.                 board[5 - x][y] = '*';
338.             break;
339.         }
340.     }
341.     else { //for #
342.         board[6][5] = intToChar((timeBombTimer) + 1);
343.         for (int x = 0; x < 6; x++) {
344.             if (board[5 - x][y] == '-') {
345.                 board[5 - x][y] = '#';
346.                 break;
347.             }
348.         }
349.     }
350. }
351.
352. /**
353.  * A procedure that will find the time bomb and clear the surrounding
354.  * counters
355.  *
356.  * @param board latest board
357.  * @param turn player
358.  */
359. public static void timeBombClear(char[][] board, boolean turn) {
360.     int[] bomb = new int[2]; //An array that is intended to hold the x and y coordinates of the
    bomb position on the board
361.     for (int i = 0; i < 6; i++) {
362.         for (int j = 0; j < 7; j++) {
363.             if (turn) {
364.                 if (board[i][j] == '*') {
365.                     bomb[0] = i;
366.                     bomb[1] = j;
367.                 }
368.             } else {
369.                 if (board[i][j] == '#') {
370.                     bomb[0] = i;
371.                     bomb[1] = j;
372.                 }
373.             }
374.         }
375.     }
376.     // This next set of code is to ensure that we clear the surrounding counter spaces given tha
    t they remain in the array bounds
377.     if (bomb[0] < 5) { //last row
378.         board[bomb[0] + 1][bomb[1]] = '-';
379.     }
380.     if (bomb[0] > 0) { //first row
381.         board[bomb[0] - 1][bomb[1]] = '-';
382.     }
383.     if (bomb[1] < 6) { //last column
384.         board[bomb[0]][bomb[1] + 1] = '-';
385.     }
386.     if (bomb[1] > 0) { //first column
387.         board[bomb[0]][bomb[1] - 1] = '-';

```

```

388.     }
389.     if (bomb[0] > 0 && bomb[1] > 0) { //top left corner
390.         board[bomb[0] - 1][bomb[1] - 1] = '-';
391.     }
392.     if (bomb[0] > 0 && bomb[1] < 6) { //top right corner
393.         board[bomb[0] - 1][bomb[1] + 1] = '-';
394.     }
395.     if (bomb[0] < 5 && bomb[1] > 0) { //bottom left corner
396.         board[bomb[0] + 1][bomb[1] - 1] = '-';
397.     }
398.     if (bomb[0] < 5 && bomb[1] < 6) { //bottom right corner
399.         board[bomb[0] + 1][bomb[1] + 1] = '-';
400.     }
401.     board[bomb[0]][bomb[1]] = '-'; //don't forget the bomb space
402. }
403. /**
404.  * A procedure that will ensure that every counter falls to the bottom (once a time bomb has cleared)
405.  * @param board latest board
406.  */
407. public static void fallCounters(char[][] board){
408.     for (int y = 0; y < 7; y++) {
409.         for (int x=0; x < 5; x++) {
410.             if (board[x][y]!='-' && board[x+1][y]=='-')
411.                 board[x+1][y]=board[x][y]; //move the counter into the new position below
412.             board[x][y]='-'; //erase the counter from the original position
413.             x--;
414.         }
415.     }
416. }
417. }
418. /**
419.  * public static boolean isBreak(char[][] board, int column){
420.     for (int x=0; x<6; x++){
421.         if (board[x][y]!='-'){
422.
423.         }
424.     }
425. }
426. */
427.
428.
429. /**
430.  * A method that checks if the game has been won. The function contains many
431.  * counters that look to find a connected 4.
432.  *
433.  * @param board The state of the board.
434.  * @return true if the game has been won.
435.  */
436. public static boolean isWin(char[][] board) {
437.     int rowCount = 0;
438.     int colCount = 0;
439.     int[] rightDiag = new int[6]; //all right diagonals
440.     int[] leftDiag = new int[6]; //all left diagonals
441.     for (int y = 0; y < 7; y++) {
442.         for (int x = 0; x < 6; x++) {
443.             switch (board[x][y]) { //switch statement that checks the mark
444.                 case 'X':
445.                     if (colCount < 0) {colCount = 0;} //if previously, the counter was in favour
446.                     colCount++;
447.                     break;
448.                 case 'O':
449.                     if (colCount > 0) {colCount = 0;} //similar to above but for if the counter
450.                     colCount--;
451.                     break;

```

```

452.         default: //if neither of the above then the space must be unoccupied
453.             colCount = 0; //set count back to zero since an unoccupied space is as bad as
an opposing mark for a connected line of marks
454.             break;
455.         }
456.         if ((colCount * colCount) == 16) { //if during the column, the counter is weighted 4
for either mark then a connect 4 has been obtained
457.             //System.out.println("column win, colCount is:"+colCount);
458.             return true;
459.         }
460.     }
461.     colCount = 0; //at the end of the column, if not already returned, the count will be res
et as to not count for a win in a column over many columns (to not leak into another column)
462.
463. }
464. for (int x = 0; x < 6; x++) {
465.     for (int y = 0; y < 7; y++) {
466.         switch (board[x][y]) {
467.             case 'X':
468.                 if (rowCount < 0) {rowCount = 0;}
469.                 rowCount++; //here instead we use the row counter instead of the column coun
ter
470.                 break;
471.             case 'O':
472.                 if (rowCount > 0) {rowCount = 0;}
473.                 rowCount--;
474.                 break;
475.             default:
476.                 rowCount = 0;
477.                 break;
478.         }
479.         if ((rowCount * rowCount) == 16) {
480.             //System.out.println("row win");
481.             return true;
482.         }
483.         //Right Diagonals
484.         if (x == y + 2) { //equation of a right diagonal line
485.             switch (board[x][y]) {
486.                 case 'X':
487.                     if (rightDiag[2] < 0) {rightDiag[2] = 0;}
488.                     rightDiag[2]++;
489.                     /**
490.                      * index of the counter somewhat refers to the equation of the line as I
may want to utilise the index later.
491.                      * Possibly an AI, and so the index (2) could refer to the EoFL (x == y+
(2)), and so value carries two pieces of info
492.                      * 1.the value (how many counters in connection) and
493.                      * 2.the index (a reference to the line in question)
494.                      */
495.                     break;
496.                 case 'O':
497.                     if (rightDiag[2] > 0) {rightDiag[2] = 0;}
498.                     rightDiag[2]--;
499.                     break;
500.                 default:
501.                     rightDiag[2] = 0;
502.                     break;
503.             }
504.         }
505.         if (x == y + 1) { //equation of a right diagonal line
506.             switch (board[x][y]) {
507.                 case 'X':
508.                     if (rightDiag[1] < 0) {rightDiag[1] = 0;}
509.                     rightDiag[1]++;
510.                     break;
511.                 case 'O':
512.                     if (rightDiag[1] > 0) {rightDiag[1] = 0;}
513.                     rightDiag[1]--;
514.                     break;

```

```

515.         default:
516.             rightDiag[1] = 0;
517.             break;
518.     }
519. }
520. if (x == y) { //equation of the right diagonal line
521.     switch (board[x][y]) {
522.         case 'X':
523.             if (rightDiag[0] < 0) {rightDiag[0] = 0;}
524.             rightDiag[0]++;
525.             break;
526.         case 'O':
527.             if (rightDiag[0] > 0) {rightDiag[0] = 0;}
528.             rightDiag[0]--;
529.             break;
530.         default:
531.             rightDiag[0] = 0;
532.             break;
533.     }
534. }
535. if (x == y - 1) { //equation of a right diagonal line
536.     switch (board[x][y]) {
537.         case 'X':
538.             if (rightDiag[3] < 0) {rightDiag[3] = 0;}
539.             rightDiag[3]++;
540.             break;
541.         case 'O':
542.             if (rightDiag[3] > 0) {rightDiag[3] = 0;}
543.             rightDiag[3]--;
544.             break;
545.         default:
546.             rightDiag[3] = 0;
547.             break;
548.     }
549. }
550. if (x == y - 2) { //equation of a right diagonal line
551.     switch (board[x][y]) {
552.         case 'X':
553.             if (rightDiag[4] < 0) {rightDiag[4] = 0;}
554.             rightDiag[4]++;
555.             break;
556.         case 'O':
557.             if (rightDiag[4] > 0) {rightDiag[4] = 0;}
558.             rightDiag[4]--;
559.             break;
560.         default:
561.             rightDiag[4] = 0;
562.             break;
563.     }
564. }
565. if (x == y - 3) { //equation of a right diagonal line
566.     switch (board[x][y]) {
567.         case 'X':
568.             if (rightDiag[5] < 0) {rightDiag[5] = 0;}
569.             rightDiag[5]++;
570.             break;
571.         case 'O':
572.             if (rightDiag[5] > 0) {rightDiag[5] = 0;}
573.             rightDiag[5]--;
574.             break;
575.         default:
576.             rightDiag[5] = 0;
577.             break;
578.     }
579. }
580. //Left Diagonals
581. if (x + y == 6 + 2) {
582.     switch (board[x][y]) {
583.         case 'X':

```

```

584.         if (leftDiag[2] < 0) {leftDiag[2] = 0;}
585.         leftDiag[2]++;
586.         break;
587.     case 'O':
588.         if (leftDiag[2] > 0) {leftDiag[2] = 0;}
589.         leftDiag[2]--;
590.         break;
591.     default:
592.         leftDiag[2] = 0;
593.         break;
594.     }
595. }
596. if (x + y == 6 + 1) {
597.     switch (board[x][y]) {
598.     case 'X':
599.         if (leftDiag[1] < 0) {leftDiag[1] = 0;}
600.         leftDiag[1]++;
601.         break;
602.     case 'O':
603.         if (leftDiag[1] > 0) {leftDiag[1] = 0;}
604.         leftDiag[1]--;
605.         break;
606.     default:
607.         leftDiag[1] = 0;
608.         break;
609.     }
610. }
611. if (x + y == 6) {
612.     switch (board[x][y]) {
613.     case 'X':
614.         if (leftDiag[0] < 0) {leftDiag[0] = 0;}
615.         leftDiag[0]++;
616.         break;
617.     case 'O':
618.         if (leftDiag[2] > 0) {leftDiag[0] = 0;}
619.         leftDiag[0]--;
620.         break;
621.     default:
622.         leftDiag[0] = 0;
623.         break;
624.     }
625. }
626. if (x + y == 6 - 1) {
627.     switch (board[x][y]) {
628.     case 'X':
629.         if (leftDiag[3] < 0) {leftDiag[3] = 0;}
630.         leftDiag[3]++;
631.         break;
632.     case 'O':
633.         if (leftDiag[3] > 0) {leftDiag[3] = 0;}
634.         leftDiag[3]--;
635.         break;
636.     default:
637.         leftDiag[3] = 0;
638.         break;
639.     }
640. }
641. if (x + y == 6 - 2) {
642.     switch (board[x][y]) {
643.     case 'X':
644.         if (leftDiag[4] < 0) {leftDiag[4] = 0;}
645.         leftDiag[4]++;
646.         break;
647.     case 'O':
648.         if (leftDiag[4] > 0) {leftDiag[4] = 0;}
649.         leftDiag[4]--;
650.         break;
651.     default:
652.         leftDiag[4] = 0;

```

```

653.             break;
654.         }
655.     }
656.     if (x + y == 6 - 3) {
657.         switch (board[x][y]) {
658.             case 'X':
659.                 if (leftDiag[5] < 0) {leftDiag[5] = 0;}
660.                 leftDiag[5]++;
661.                 break;
662.             case 'O':
663.                 if (leftDiag[5] > 0) {leftDiag[5] = 0;}
664.                 leftDiag[5]--;
665.                 break;
666.             default:
667.                 leftDiag[5] = 0;
668.                 break;
669.         }
670.     }
671.     for (int i = 0; i < rightDiag.length; i++) { //iterate through the array of counters
        at each stage
672.         if ((rightDiag[i] * rightDiag[i]) == 16 || (leftDiag[i] * leftDiag[i]) == 16) {
673.             //System.out.println("diag win on"+i);
674.             return true;
675.         }
676.     }
677. }
678.     rowCount = 0; //after each row, if no win has been found then this line of code is reach
        ed and so the counter is reset to 0 ready for the next row
679. }
680.     return false; //if all attempts to check for a win yeild no return (no win) then return fals
        e and indicate that a win has not been reached
681. }
682.
683. /**
684.  * A method that checks if the game is a draw by checking the board is full AND the game hasn't
        been won.
685.  *
686.  * @param board The state of the board.
687.  * @return True if the game is a draw.
688.  */
689. public static boolean isDraw(char[][] board) {
690.     return turnCounter(board) == (7 * 6) && !isWin(board);
691. }
692.
693. /**
694.  * A method that counts the number of turns made on the board.
695.  *
696.  * @param board (char[][]) The game board.
697.  * @return (int) the number of turns made.
698.  */
699. public static int turnCounter(char[][] board) {
700.     int movesMade = 0;
701.     for (int x = 0; x < 6; x++) {
702.         for (int y = 0; y < 7; y++) {
703.             if (board[x][y] == 'X' || board[x][y] == 'O') { //if a mark exists, then the move has
                been made
704.                 movesMade++;
705.             }
706.         }
707.     }
708.     return movesMade;
709. }
710.
711. /**
712.  * A method that prompts the user to input a string and then takes the input with the aid of a p
        rompt.
713.  *
714.  * @param message prompt

```

```

715.     * @return (string) user input
716.     */
717.     public static String readInputString(String message) {
718.         System.out.print(message); //prompt
719.         String input = stdin.nextLine(); //reads input
720.         return input;
721.     }
722.
723.     /**
724.      * A method that interprets the move trying to be made by the user, and in the case where
725.      * it is not a special move, will take the desired column.
726.      * @param message prompt
727.      * @return an array of two elements:
728.      *         1.First element indicates the type of move
729.      *         2.Second element indicates the column selected but is only utilised
730.      *         once in this function.
731.      */
732.     public static int[] readInputMove(String message) {
733.         boolean validInput = false;
734.         int[] input = new int[]{0, 0};
735.         do {
736.             String userInput = readInputString(message);
737.             switch (userInput) {
738.                 case "b": //Blitz
739.                     input[0] = 1;
740.                     validInput = true;
741.                     break;
742.                 case "t": //TimeBomb
743.                     input[0] = 2;
744.                     validInput = true;
745.                     break;
746.                 default: //Standard Move
747.                     input[0] = 0;
748.                     try {
749.                         input[1] = Integer.parseInt(userInput); //changes the string input to an int
750.                         if (input[1] > -1 && input[1] < 7) { //Within the bounds of the board
751.                             validInput = true;
752.                         } else {
753.                             System.out.println("Invalid input! No such column exists. Select from 0-6.");
754.                         }
755.                     } catch (NumberFormatException e) {
756.                         System.out.println("Invalid input! Please input an integer or select a special ability.");
757.                     }
758.                 } while (!validInput);
759.             return input;
760.         }
761.     }
762.
763.     /**
764.      * A method that will read the input from the user after prompted.
765.      * @param message prompt
766.      * @return user input
767.      */
768.     public static int readInputInteger(String message) {
769.         boolean validInput = false;
770.         int input = 0;
771.         while (!validInput) {
772.             System.out.println(message);
773.             try {
774.                 input = Integer.parseInt(stdin.nextLine());
775.                 validInput = true;
776.             } catch (NumberFormatException e) {
777.                 System.out.println("Invalid input! Please input an integer.");
778.             }
779.         }
780.         return input;

```

```

781.     }
782.
783.     /**
784.      * A method that, incorporating the function above, reads the input for the column, ensuring
785.      * that the input keeps within the bounds
786.      * @param message prompt
787.      * @return user input
788.      */
789.     public static int readInputColumn(String message) {
790.         boolean validInput = false;
791.         int input;
792.         do {
793.             input = readInputInteger(message);
794.             if (input > -1 && input < 7) { //within bounds of the board
795.                 validInput = true;
796.             } else {
797.                 System.out.println("Invalid input! No such column exists. Select from 0-6.");
798.             }
799.         } while (!validInput);
800.         return input;
801.     }
802.
803.     /**
804.      * A method that makes it easier, by calling, to change data types from char to int
805.      * @param character character
806.      * @return integer
807.      */
808.     public static int charToInt(char character) {
809.         int numeric = Character.getNumericValue(character);
810.         return numeric;
811.     }
812.
813.     /**
814.      * A method that makes it easier, by calling, to change data types from int to char
815.      * @param integer
816.      * @return character
817.      */
818.     public static char intToChar(int integer) {
819.         char character = Character.forDigit(integer, 10);
820.         return character;
821.     }
822. }

```

Java Programming and Testing Part 2

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

83/100

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

