Daniel Nedrow

CEG 3110 Project 4

**Test Plan**

We want to test the Cows and Bulls game with a focus on the part of the game that gives feedback to the user based on their guess. There are 14 total combinations of cows and bulls the user can get: (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3), (4,0), (4,1), (4,2), and (4,4). Notice that (4,3) is not possible because if three of the matching numbers were in the correct position, then the fourth one would necessarily be as well. Since this is a manageable number of feedback combinations, it is reasonable to test them all.

I have written the game code myself, and I implemented a 10-turn limit. Of course, I also included the ability to bypass the randomly assigned secret number and assign the “secret” number manually (so that testing is possible). The first 10 cow/bull combinations above can thus be put into one game (a losing effort, which by default will test the 10-turn limit), and the next four can be assigned to a second game (a winning effort). The second game will also include three more test cases that contribute to coverage of the assignment’s requirements: a test case will show that manually inputting a secret number with repeated digits is not allowed, another test case will show that manually inputting a secret number with a leading zero is indeed allowed, and finally a test case will show that repeated digits in the user’s guess are not allowed. Naturally, winning this game will prove that guesses with leading zeroes are allowed.

I also will include a third game to test a final requirement the first two games can’t test. The lone test case will show that the game proceeds normally when the option for a random secret number is chosen (I’m not testing randomness, but rather that the game works).

**Game A**

Precondition to all Game A test cases: the “secret” number 4 1 9 6 has been entered manually. The test cases need not necessarily be run in order, but the 10th test case must be run as the 10th turn in order to lose the game by the 10-turn rule.

|  |  |
| --- | --- |
| **Test Case 1a:** secret number 4 1 9 6  Input: 2 0 5 7  Expected Output: 0 Cows, 0 Bulls  Actual Output: 0 Cows, 0 Bulls  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 2a:** secret number 4 1 9 6  Input: 1 0 5 7  Expected Output: 1 Cows, 0 Bulls  Actual Output: 1 Cows, 0 Bulls  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 3a:** secret number 4 1 9 6  Input: 2 1 5 7  Expected Output: 1 Cows, 1 Bulls  Actual Output: 1 Cows, 1 Bulls  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 4a:** secret number 4 1 9 6  Input: 1 4 5 7  Expected Output: 2 Cows, 0 Bulls  Actual Output: 2 Cows, 0 Bulls  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 5a:** secret number 4 1 9 6  Input: 4 5 7 1  Expected Output: 2 Cows, 1 Bulls  Actual Output: 2 Cows, 1 Bulls  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 6a:** secret number 4 1 9 6  Input: 4 1 5 7  Expected Output: 2 Cows, 2 Bulls  Actual Output: 2 Cows, 2 Bulls  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 7a:** secret number 4 1 9 6  Input: 9 4 1 7  Expected Output: 3 Cows, 0 Bulls  Actual Output: 3 Cows, 0 Bulls  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 8a:** secret number 4 1 9 6  Input: 4 9 1 7  Expected Output: 3 Cows, 1 Bulls  Actual Output: 3 Cows, 1 Bulls  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 9a:** secret number 4 1 9 6  Input: 4 1 7 9  Expected Output: 3 Cows, 2 Bulls  Actual Output: 3 Cows, 2 Bulls  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 10a:** game lost by 10-turn rule  Input: 4 1 9 8  Expected Output: 3 Cows, 3 Bulls  Sorry, you are limited to 10 turns.  The actual answer was 4 1 9 6.  Actual Output: 3 Cows, 3 Bulls  Sorry, you are limited to 10 turns.  The actual answer was 4 1 9 6.  Pass/Fail: [ ] Pass [ ] Fail |

**Game B**

For Game B, the matter of choosing the secret number will be handled within the test cases. For proper game context, these test cases need to be run in their given order.

|  |  |
| --- | --- |
| **Test Case 1b:** illegal secret number  Input: 2 9 2 4  Expected Output:  A legal number consists of 4 space-separated digits (0-9) with no repeats!  Enter Secret Number:  Actual Output:  A legal number consists of 4 space-separated digits (0-9) with no repeats!  Enter Secret Number:  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 2b:** legal secret num w/ leading 0  Input: 0 3 8 2  Expected Output: Enter Guess:  Actual Output: Enter Guess:  (Note: the output merely shows the program proceeds as expected when a legal secret number is entered).  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 3b:** user enters illegal guess  Input: 5 1 9 1  Expected Output:  A legal number consists of 4 space-separated digits (0-9) with no repeats!  Enter Guess:  Actual Output:  A legal number consists of 4 space-separated digits (0-9) with no repeats!  Enter Guess:  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 4b:** secret number 0 3 8 2  Input: 3 0 2 8  Expected Output: 4 Cows, 0 Bulls  Actual Output: 4 Cows, 0 Bulls  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 5b:** secret number 0 3 8 2  Input: 0 8 2 3  Expected Output: 4 Cows, 1 Bulls  Actual Output: 4 Cows, 1 Bulls  Pass/Fail: [ ] Pass [ ] Fail | **Test Case 6b:** secret number 0 3 8 2  Input: 2 3 8 0  Expected Output: 4 Cows, 2 Bulls  Actual Output: 4 Cows, 2 Bulls  Pass/Fail: [ ] Pass [ ] Fail |
| **Test Case 7b:** game won with correct guess  Input: 0 3 8 2  Expected Output: 4 Cows, 4 Bulls  You Win!! (4 turns)  Actual Output: 4 Cows, 4 Bulls  You Win!! (4 turns)  Pass/Fail: [ ] Pass [ ] Fail |  |

**Game C**

The game begins with the following prompt: “Welcome to Cows and Bulls. Enter 'R' for a random number, enter 'I' to input the number yourself:”. A precondition for this test case is that ‘R’ has been chosen (and thus the guessing phase has begun). (A better way to test the basic functionality of the random secret number “mode” is to play the game honestly several times, which I was happy to do).

**Test Case 1c:** test that the game proceeds normally when a random secret number is chosen

Input: The tester should input 0 1 2 3 as her guess either one time (in the highly unlikely event that this is the secret number) or 10 times (continuously guessing wrong until defeat). For the output, we are only interested in the end-of-game message (not each turn’s cows and bulls).

Expected Output:

In the event of immediate victory:

You Win!! (1 turns)

In the event of defeat by the 10-turn rule:

Sorry, you are limited to 10 turns.

The actual answer was <a legal secret number>.

Actual Output:

Sorry, you are limited to 10 turns.

The actual answer was 4 8 5 6.

Pass/Fail: [ ] Pass [ ] Fail

**Flow Graph Section**

This section includes both the code blocks and the flow graph. Note that the code I’m choosing to graph is the entire main method of my program. The program is significantly modularized, however, and the Complexity Metric is only 5.

**Code Blocks**

Scanner input = new Scanner(System.in);

System.out.print("Welcome to Cows and Bulls. Enter 'R' for a random "

A + "number, enter 'I' to input the number yourself: ");

char result = input.next().charAt(0);

int[] secretNumber;

-----------------------------------------------------------------------------

B if (result == 'R') {

-----------------------------------------------------------------------------

C secretNumber = generateRandomSecretNumber();

-----------------------------------------------------------------------------

} else {

D secretNumber = userEnterNumber("Enter Secret Number: ");

}

-----------------------------------------------------------------------------

int[] guess;

E int[] cowsBulls;

boolean hasWon = false;

int i = 0;

-----------------------------------------------------------------------------

F while (gameContinues(hasWon, i)) {

-----------------------------------------------------------------------------

guess = userEnterNumber("Enter Guess: ");

cowsBulls = getCowsAndBulls(secretNumber, guess);

G System.out.println(cowsBulls[0] + " Cows, " + cowsBulls[1]

+ " Bulls");

i++;

-----------------------------------------------------------------------------

H if (cowsBulls[1] == 4) {

-----------------------------------------------------------------------------

hasWon = true;

I }

} // end while

-----------------------------------------------------------------------------

J if (hasWon) {

-----------------------------------------------------------------------------

K System.out.println("You Win!! (" + i + " turns)");

-----------------------------------------------------------------------------

} else {

L System.out.println("Sorry, you are limited to 10 turns.\nThe actual"

+ " answer was" + arrToString(secretNumber) + ".");

}

-----------------------------------------------------------------------------

END

**Flow Graph**

5

4

3

2

1

**END**

**L**

**J**

**K**

**F**

**G**

**H**

**I**

**E**

**D**

**C**

**B**

**A**

We can clearly see that Complexity Metric is 5 from the numbered regions of the flow graph. We can also use the equation M = E – N + 2, where E = 16 (the number of edges in the graph) and N = 13 (the number of nodes in the graph), and we still get 16 – 13 + 2 = 5.

end while

if false

if true

if true

if false

end while

if false

if true

**Statement and Branch Coverage**

**Statement Coverage**

Because of the presence of if/else branches in the code which proceed to mutually exclusive statements, we cannot get to all the statements in one pass. We need at least two.

Path 1: A B D E F G H I F J K END

Path 2: A B C E F G H F J L END

Notice that all statement nodes A-L (as well as END) are touched in these two paths.

**Branch Coverage**

It turns out that for the way my code is constructed, statement and branch coverage can both be achieved with the two paths listed above. There are four decision nodes in the code. Decision node B branches to D in Path 1, and to C in Path 2. Decision node F is a while loop, and in both Path 1 and Path 2 the paths of entering the loop and exiting the loop are both explored. Decision node H is an if statement that is part of the last block of code in the while loop: in Path 1, we take the “true” path to node I, whereas in Path 2 we bypass this and go directly back to the loop (node F). Decision node J branches to K in Path 1, and to L in Path 2. As you can see, Path 1 and Path 2 above collectively achieve branch coverage.

**Did the Test Cases Achieve Statement Coverage and Branch Coverage?**

We need only to show that the test cases collectively traversed Path 1 and Path 2 above. Game B is a 4-turn winning effort in which the “secret” number was entered manually. We start by executing the code in node A, and the manual entry of the “secret” number takes us from decision node B to node D. We move to node E and then enter the while loop at decision node F. The “G H” part of Path 1 is actually now repeated three times as G H F G H F G H F because we are guessing wrong three times and decision node H is taking us back to the loop node F. We then rejoin Path 1 above by executing node G, H, and now finally I (because it is true that we have guessed correctly). We exit the loop at node F, proceed to decision node J, branch to node K (because we have won), and then exit at the END node. Game B travels through all of Path 1 (as well as looping three extra times).

Path 2 represents the program’s flow for Game C. After executing node A and arriving at node B, we now choose a randomly generated secret number which takes us to node C. We move on to node E and hit loop node F. Without further belaboring the issue, we loop through F, G, and H 10 times (because we guess incorrectly 10 times) before exiting the loop at node F and proceeding to decision node J. This time we have lost the game, so we branch to node L before exiting at the END node.

What happens if we “accidentally” win Game C by somehow matching “0 1 2 3” to the randomly generated secret number? Well, we have still achieved total coverage of the nodes in Path 1 and Path 2 (and thus statement and branch coverage) if you also consider Game A’s test cases. If Game C is miraculously a win, we have merely failed to take the “false” branch at decision node J, and we have failed to execute statement L. We get everything else we needed out of Path 2. Well, Game A guarantees us a loss by the 10-turn rule, so this single-branch and single-statement deficit are guaranteed to be covered in Game A. Thus, we have achieved both statement coverage and branch coverage with our original test cases in the three games in our test plan.

**Complete Source Code**

/\*

\* Daniel Nedrow

\* CEG 3110

\* Project 4

\* Professor John Reisner

\*/

package ceg3110project4;

import java.util.Scanner;

/\*\*

\* @author Daniel

\* This program simulates a Cows and Bulls game.

\*/

public class CEG3110Project4 {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

// prompt for random or manually entered secret number

System.out.print("Welcome to Cows and Bulls. Enter 'R' for a random "

+ "number, enter 'I' to input the number yourself: ");

char result = input.next().charAt(0);

int[] secretNumber;

// based on user's choice, get random or manually entered secret number

if (result == 'R') {

secretNumber = generateRandomSecretNumber();

} else {

secretNumber = userEnterNumber("Enter Secret Number: ");

}

int[] guess;

int[] cowsBulls;

boolean hasWon = false;

int i = 0;

// continue guessing until victory or 10 turns have been played

while (gameContinues(hasWon, i)) {

// get user's guess

guess = userEnterNumber("Enter Guess: ");

// get number of cows and bulls and display these numbers

cowsBulls = getCowsAndBulls(secretNumber, guess);

System.out.println(cowsBulls[0] + " Cows, " + cowsBulls[1]

+ " Bulls");

i++; // increment turn number

// check for victory

if (cowsBulls[1] == 4) {

hasWon = true;

}

} // end while loop

// If player won, display victory msg. Otherwise, display defeat msg.

if (hasWon) {

System.out.println("You Win!! (" + i + " turns)");

} else {

System.out.println("Sorry, you are limited to 10 turns.\nThe actual"

+ " answer was" + arrToString(secretNumber) + ".");

}

} // end main method

/\*\*

\* Generates a secret number with 4 random digits (none of which can be

\* repeated). The number is stored as an array.

\* @return an array of size 4 containing each digit of the secret number

\*/

public static int[] generateRandomSecretNumber() {

int[] result = {-1, -1, -1, -1}; // dummy values for secret number

// generate each of the 4 digits

for (int i = 0; i < result.length; i++) {

// get a random number 0-9

int potentialPick = (int) (Math.random() \* 10);

// while we have a repeated digit, continuously repick a new number

while (contains(result, potentialPick)) {

potentialPick = (int) (Math.random() \* 10);

}

result[i] = potentialPick; // assign unique pick to secret number

}

return result;

}

/\*\*

\* Figures out how many cows and bulls the user got by comparing the user's

\* guess to the secret number.

\* @param secretNumber the secret 4-digit number

\* @param guess the user's guess of the 4-digit number

\* @return an array of size 2 containing the number of cows, bulls

\*/

public static int[] getCowsAndBulls(int[] secretNumber, int[] guess) {

int[] result = {0, 0}; // cows and bulls start at 0

// examine every digit of guess to see if it is a cow, bull, or miss

for (int i = 0; i < guess.length; i++) {

// check if the number is a cow (is it anywhere in the secret num)

if (contains(secretNumber, guess[i])) {

result[0] = result[0] + 1;

// check the cow to see if it is also a bull (position matches)

if (secretNumber[i] == guess[i]) {

result[1] = result[1] + 1;

}

}

}

return result;

}

/\*\*

\* Allows the user to enter a 4-digit number. This method is useful for

\* letting the user enter either a manual secret number or a guess.

\* @param prompt an appropriate prompt (asking for secret number or guess)

\* @return an array with user-selected 4-digit number with no repetition

\*/

public static int[] userEnterNumber(String prompt) {

int[] result = {-1, -1, -1, -1}; // dummy values for number

Scanner input = new Scanner(System.in);

boolean legalNum = true;

// continue prompting for 4-digit number until user makes legal choice

do {

System.out.print(prompt); // display selected prompt for number

// examine each of the 4 digits for range (0-9) and no repetition

for (int i = 0; i < result.length; i++) {

int potentialDigit = input.nextInt(); // get next digit

// check for illegal digit: repeat or out of range (0-9)

if (contains(result, potentialDigit) || potentialDigit < 0

|| potentialDigit > 9) {

legalNum = false; // we'll need user to choose another num

System.out.println("A legal number consists of 4 "

+ "space-separated digits (0-9) with no repeats!");

// reset the state of result array for new repetition check

for (int j = 0; j < result.length; j++) {

result[j] = -1; // avoid false positive on repeat check

}

input.nextLine(); // clear input buffer of remaining digits

break; // found illegal digit, stop examing this number

}

result[i] = potentialDigit; // assign the legal digit to result

legalNum = true; // note that we have a legal number (so far)

} // end for loop

} while (!legalNum);

return result;

}

/\*\*

\* Checks if an array of numbers contains a given number.

\* @param checkArray the array to check

\* @param checkNum the number we are looking for

\* @return true if the number is in the array, false otherwise

\*/

public static boolean contains(int[] checkArray, int checkNum) {

// check each number of the array

for (int i = 0; i < checkArray.length; i++) {

if (checkArray[i] == checkNum) {

return true;

}

}

return false;

}

/\*\*

\* Turns an array of numbers into a single String for display.

\* @param arr the array of numbers

\* @return a String of space-separated numbers

\*/

public static String arrToString(int[] arr) {

String result = "";

// append each number in the array onto the String

for (int i = 0; i < arr.length; i++) {

result += " " + arr[i];

}

return result;

}

/\*\*

\* Determines if a Cows and Bulls game should continue based on whether the

\* player has already won and the turn number. This method is only included

\* to avoid an "&&" in the main method (for easier code graphing).

\* @param hasWon boolean indicating if player has won or not

\* @param turn the current turn number

\* @return true if the game should continue, false otherwise

\*/

public static boolean gameContinues(boolean hasWon, int turn) {

return !hasWon && turn < 10;

}

}