| **Test Name** | | | Game odds do not appear to be correct | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Use Case Tested:** | | | Crown and Anchor Game | | | |
| **Test Description:** | | | Crown and Anchor game have an approximate 8% bias to the house. The win+lose ratio should equal approximately 0.42. This does not appear to be the case | | | |
| **Pre-conditions** | | | Default game setup | | | |
| **Post-conditions** | | | Once the game has finished the reported win ratio should be around 0.42. Because this value is only an approximate and the game is random we are going to say that an acceptable difference is minus/plus two percent in either direction. This means that the win ratio can be anywhere between 0.4 and 0.44 to be acceptable. | | | |
| **Notes:** | | **In order to test this bug the game must be played a few times. We know the bug does exist as it was reported so the game needs to be player until the ratio is off. In order to confirm the bug we will also say that if the bug does not show up after 20 games then the bug doesn’t actually exist. A pass will indicate that the bug does in fact exist** | | | | |
| **Result (Pass/Fail/Warning/Incomplete)** | |  | | | | |
|  | **TEST STEP** | | | **EXPECTED TEST RESULTS** | P | F |
|  | Run program | | |  | x |  |
|  | Look through all the results and count how many times the house has win/lose. Work out the ratio that the house is winning | | | The ratio that the house is winning/losing at should not be close to 0.42 | x |  |
|  |  | | |  |  |  |

# Test Result

Output from game

**Run One**

195 turns later.

End Game 99: Fred now has balance 200

Win count = 9037, Lose Count = 13279, 0.40

The first run is in the acceptable range, so we need to play the game again in order to find the bug

**Run two**

113 turns later.

End Game 99: Fred now has balance 0

Win count = 10746, Lose Count = 15772, 0.41

**Run Three**

359 turns later.

End Game 99: Fred now has balance 200

Win count = 8890, Lose Count = 13112, 0.40

**Run Four**

153 turns later.

End Game 99: Fred now has balance 0

Win count = 9594, Lose Count = 14129, 0.40

**Run Five**

112 turns later.

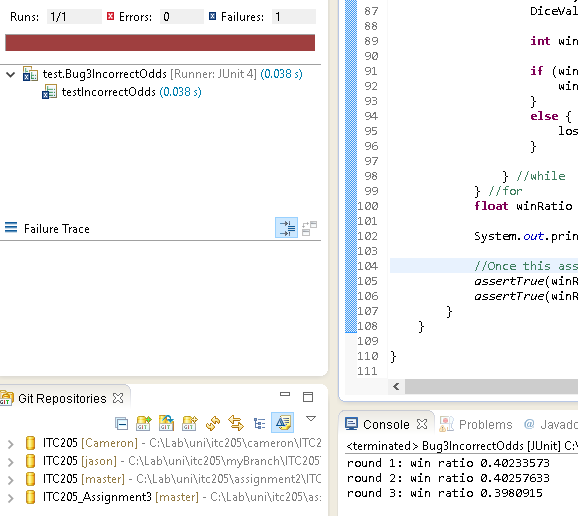
End Game 99: Fred now has balance 200

Win count = 6279, Lose Count = 4279, 0.59

Diagnose

In order to reproduce the bug so that the win ratio was not in the acceptable range I had to run the game five times. From looking at the results the first few runs had the win ratio hang around 0.4, this is a little low from what the game is meant to be at, but as the game is random this is still an acceptable range. The last run had the win ratio at 0.59 which is way too high to be acceptable and is an indication that the game is not placing a bias onto the house to insure that the house wins more than the player does.

# Automated Test with Junit to show the bug



As the automated test shows the bug can be reproduced automatically. It might not show up right away but it is clearly there. This also is another indication that the game at present does not take into account any kind of bias. In order to test this kind of bug I felt it would be best to place the test code into an endless loop until the bug showed its self up. Once the assert fails then the loop finished. The test code was copyed directly out of Main.java so the game would play out as it currently does. I removed a few output statements as these where not needed and removed a few variables that the test didn’t need. In order to help visualize the bug in an automated process I also added in one text output that would let me know the win ratio for each round and what round we are currently on. This has the added benefit that if the bug doesn’t appear to be showing up after say 1000 rounds then we can determine the test case won’t show the bug. Or that the bug doesn’t exist in what we are testing.

# Tracing the code back to find the bug

From looking at the results from both tests it would appear that the game doesn’t take any bias into account when picking the houses dice. In order to confirm that this is the case we are going to have to walk through some of the code in order to see if we can find any bias calculations. We are going to start the trace at Main.java if we look through all the lines of code we notice the following which is line 49

**int** winnings = game.playRound(player, pick, bet);

The line seems to indicate that it is what actually plays the game. So at this point it would be worthwhile looking at this function. So let’s open up Game.java and find this function. This function can be found on line 28, we are going to walk through this function line by line and see if we can spot something that indicates some kind of bias to the house. We can see that line 33 takes the players bet. This doesn’t indicate any kind of bias. Line 36 to line 41 does look a little interesting though. At first glance there doesn’t seem to be any kind of bias within this loop, but one thing it does do is roll the dice. So if we are looking for a bias we would expect to see it here as this is where we are getting the dice value from. If we look at line 37 we can see

d.roll();

Seems interesting, it might be worth while taking a look at the roll() function. We can find this function inside Dice.java (d is the variable name for the Dice object). So let’s open up Dice.java and locate this function.

We can find this function on line 14 of Dice.java, the function in its self is a little boring all it does is call DiceValue.getRandom() with that said it would appear that the purpose of this function is to return a random value and nothing else, we still haven’t found the bias so we might as well take a look at the DiceValue class and the getRandom function, as the bias must be in there.

Once we open up DiceValue.java we can find the getRandom function on line 25, once again this function doesn’t seem to have bias within it. All it is doing is returning a random value out of the enum

**int** random = *RANDOM*.nextInt(DiceValue.*SPADE*.ordinal());

So with all that said and done it would appear that we just followed a dead end. I would like to think of it as more eliminating possible section of code, so let’s go back a few steps and look at Game.java again, we will start with line 38 (as we left on line 37), From looking at the rest of the lines in this function there isn’t anything that is a bias, the game just applies the winning if needed and returns.

With all that said and done the only conclusion I can come up with is that the game does not add any kid of bias to the results which is why the win ratio is off. The solution to this issue would be to place a bias into the game, this within its self wouldn’t be very difficult and the best place to do it would be inside the playRound function of the Game class. In order to add a bias to the game we will have to tell the Game class to keep track of how many times the player wins and how many times the player loses. Once we know these values we can then work out if we need to rig the outcome so that the house wins or the player wins. The house should win if the players win ratio is above 0.44, and the house should lose if the win ratio is less than 0.40. In order to be able to do this we will also have to change the Dice class to allow us to specify a dice value, this is so we can rig the outcome of any game at any point depending on the current win ratio.

**Update One:**

When I went to add in the code to allow for the dice value to be specified I did notice that the dice value never changed. This was another bug in its self but would not of been the cause to the bug I was looking for. I have updated the code to allow for the new dice value to be assigned to the dice before returning.

**Update Two:**

The bias code worked better than expected, after running a few tests it appears that I can set the bias to be the exact percent that I want the player to win at, this means that I don’t have to say 0.4 to 0.44 is good enough

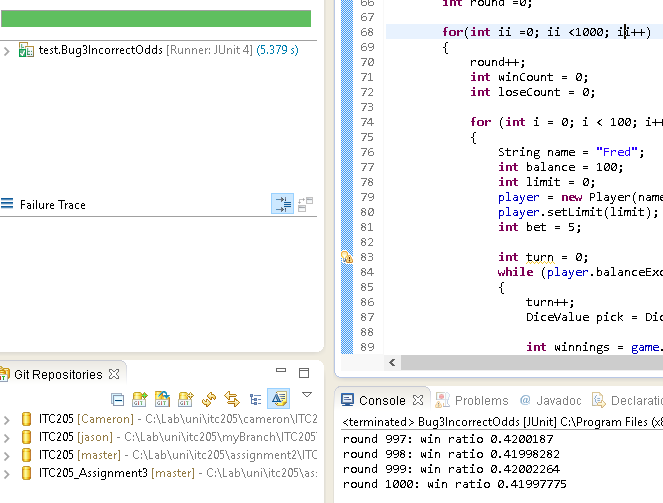
**Update Three**

During the changing of the code another bug was noticed within the DiceValue code, the getRandom value will never return the spade as a value, we need to plus one onto the

DiceValue.*SPADE*.ordinal()

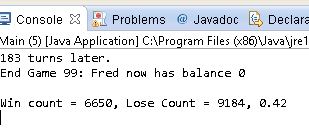
Call in order to account for the spade

**Conclusion**, because the Junit test that I currently have setup will never exit if I did fix the bug I am going to change the endless while loop so that it is a for loop instead and have it run the game 1000 times, if the test dosnt fail then the bug has been fixed. After running the test I can confirm that I have managed to add a bias to the game, it does drop a little from 0.42, but the amount it drops is nothing compared to before. The screen shot below shows the output from the test after 1000 runs

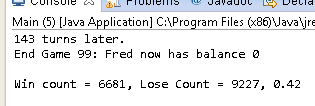


| **Test Name** | | | Game odds should stay around 0.42 | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Use Case Tested:** | | | Crown and Anchor Game | | | |
| **Test Description:** | | | Crown and Anchor game have an approximate 8% bias to the house. The win+lose ratio should equal approximately 0.42. This does not appear to be the case | | | |
| **Pre-conditions** | | | Default game setup | | | |
| **Post-conditions** | | | Once the game has finished the reported win ratio should be around 0.42. Because this value is only an approximate and the game is random we are going to say that an acceptable difference is minus/plus two percent in either direction. This means that the win ratio can be anywhere between 0.4 and 0.44 to be acceptable. | | | |
| **Notes:** | | **In order to test that the odds are correct we will run the game 5 times and check the output from each run. If the win ratio stays around 0.4 to 0.44 then the test has passed.** | | | | |
| **Result (Pass/Fail/Warning/Incomplete)** | |  | | | | |
|  | **TEST STEP** | | | **EXPECTED TEST RESULTS** | P | F |
|  | Run program | | |  | x |  |
|  | Look through all the results and count how many times the house has win/lose. Work out the ratio that the house is winning | | | The ratio that the house is winning/losing at should not be close to 0.42 | x |  |
|  |  | | |  |  |  |

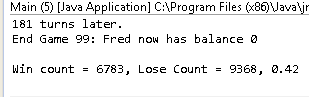
**Run 1**

**Run 2**

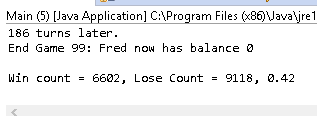
**Run 2**



**Run 3**



**Run 4**



**Run 5**

