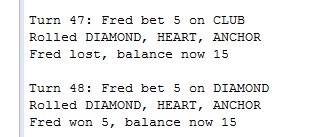
Bug Fixing Journal

# Bug 1 – Game Does Not Pay Out at Correct Levels

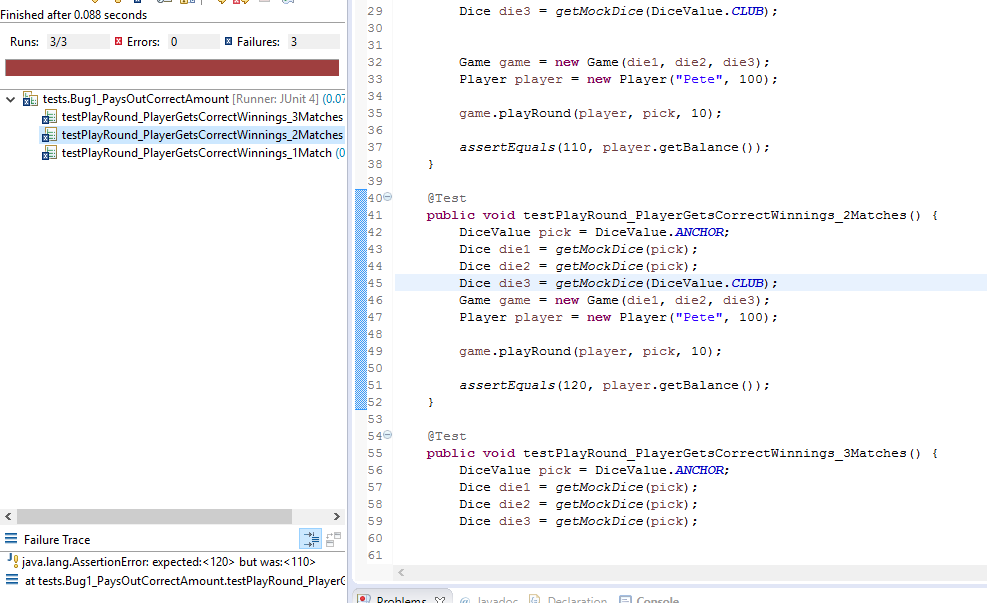
## User Test

Click Run and observe that when Fred wins, his balance increases appropriately. This test fails, for example:



At this point I briefly inspected the code to work out how it could be mocked/tested. The bug will most likely in the Game or Player classes. To test reliably, I need to create a test scenario in which the player always wins. The logical way to do this would be to rig the dice rolls.

## Test Output



As expected due to the bug report, the tests all failed.

## Debug log

Note that before I start, it is quite obvious from the results of the tests that the game is not giving back the bet amount when the player wins. I’m going to pretend it’s not that simple.

I start by looking at the code Player class to see how the balance can be effected. The first thing I want to ensure is that I’m not simply being shown the wrong balance.

**Hypothesis:** getBalance is not showing the balance correctly.

**Test (Junit):** Ensure a player created with $100 returns 100 from getBalance.

**Result:** pass

**Conclusion:** getBalance works as intended.

Other than the constructor which was proven to work in the first test, 2 methods can modify the balance: takeBet and receiveWinnings. Since both of these functions are used in the playRound method, I will start by ensuring that these 2 functions work properly.

**Hypothesis:** takeBet is taking too much away from the balance.

**Test (Junit):** Ensure that if the Player starts with $100 and a $10 bet is taken, they end up with $90.

**Result:** pass

**Conclusion:** takeBet works as intended.

**Hypothesis:** receiveWinnings is not adding to the balance properly.

**Test (Junit):** Ensure that if the Player starts with $100 and a $10 of winnings are received, they end up with $110.

**Result: pass**

**Conclusion:** receiveWinnings works as intended.

This leads me to believe that the player class is functioning properly, so the problem must be the input it is given. The first time it is used on line 33, player.takeBet(bet), the input comes straight from the method argument, so there’s not a lot of room for error.

The second time, player.receiveWinnings(winnings) is passed (or ‘dependent on’) the ‘winnings’ local variable which is defined with int winnings = matches \* bet;. This would result on the player receiving his bet back when he wins, which is what the bug is doing.

**Hypothesis:** winnings variable is not being calculated correctly.

**Test:** Set a breakpoint on line 33 for 10 winnings at line 46 (player.receiveWinnings(winnings)) with 2 matches and a bet of 10, and ensure winnings == 30. If winnings = 20, the hypothesis is correct.

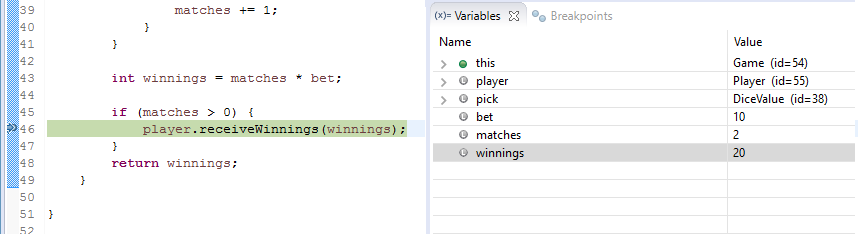
**Result: FAIL. Winnings = 20.**

**Conclusion:** winnings is not being calculated correctly.

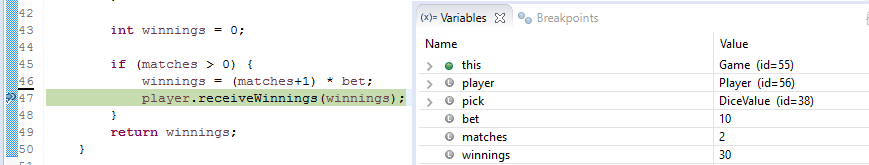
Winnings should be 0 if there are no matches, or (matches+1) \* bet if there is any. To ensure I don’t mess up the logic, at this point I’ll add an automated test for when there are no matches.

## Before and After Screenshots

The following is what the code and values look like before the bug is fixed:

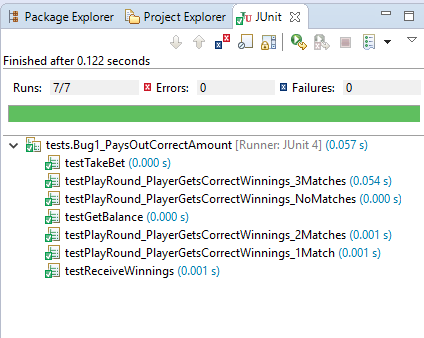


And after the bug is fixed:



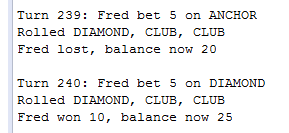
## Test Output from Fixed Bug

All tests ran successfully with the bug fix included:



## Output of Fixed Bug

Output for a winning bet is given below:



# Bug 2 – Player cannot reach betting limit

## User Test

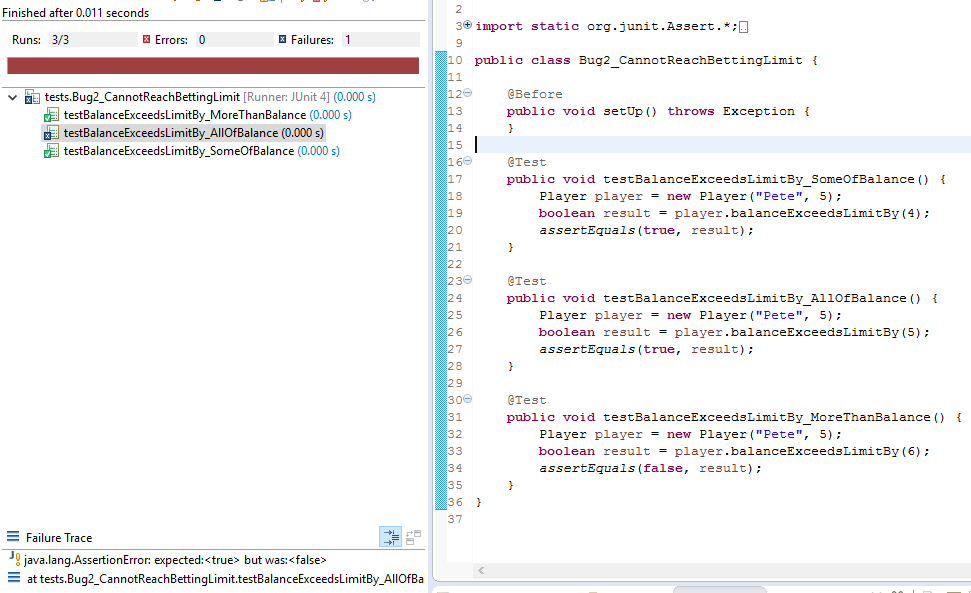
Run the program repetitively until the player stops because he has reached his limit (and not doubled his money). Ensure that the players balance is 0. Test fails because the balance is 5 when the player stops for this reason:



Because the game runs in a while loop using (player.balanceExceedsLimitBy(bet) && player.getBalance() < 200), it’s clear that there’s an issue with the Player class not checking whether the balance is exceeded correctly. Initial tests to show the bug will be to test player.balanceExceedsLimitBy.

Note that part of the issue here seems to be design; we don’t want to know if the balance exceeds a limit, we want to know if the balance exceeds OR EQUALS a limit. Something like Player.canBet(int amount) might be more appropriate, but I believe changing it is beyond the scope of the assignment.

## Test Output



Again, as expected from the bug report, the test fails.

## Debug Log

*Because the faulty function is only 1 line, it is immediately clear what is wrong.*

**Hypothesis:** balanceExceedsLimit’s calculations are incorrect.

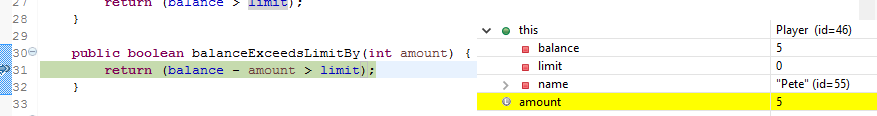
**Test (Junit):** testBalanceExceedsLimitBy\_AllOfBalance

**Result: FAIL. Returns false.**

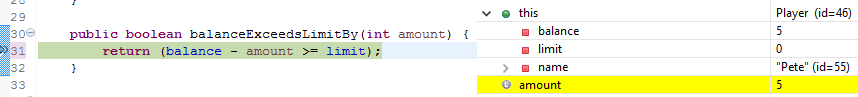
**Conclusion:** balanceExceedsLimitBy is not being calculated correctly so the hypothesis is correct.

## Before and After screenshots

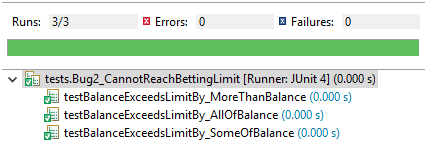
As the function is only 1 line, there is not a very appropriate place to pause the script and inspect variables. (5-5 > 0) == false:



But (5-5 >= 0) == true:



## Test output



## User Test Output



At this point there is an obvious nearby bug, but because it has not yet ‘been reported’ or caused any issues, I’ll leave it to discover using more ‘scientific’ methods.

# Bug 3 – Odds in the game do not appear to be correct

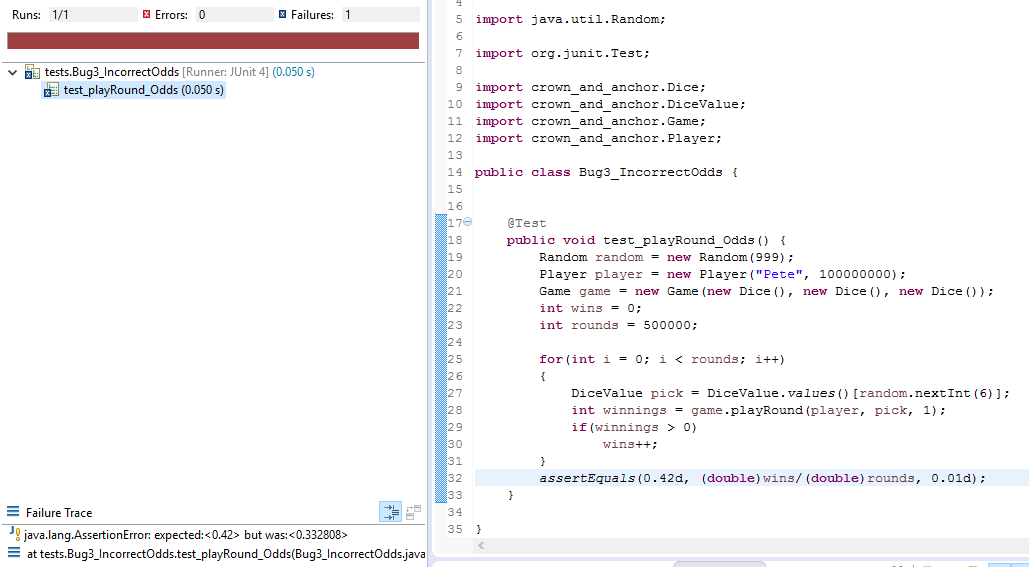
## User Test

Run the program repetitively and observe that the resulting win/loss ratio is close to the expected win/loss ratio of 0.42.

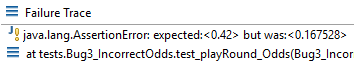
**The test currently fails because the ratio fluctuates wildly, from 0.2 to 0.6.**

****

## Test Output







## Debug Log

Interestingly, no matter how many iterations of playRound are run with random picks, certain values come up for the win/loss ratio. 0.16, 0.33, 0.50 are the ones I have observed. This makes me think that the results aren’t entirely random, and that something is happening during initialization that changes the result of playRound.

*Being a dice game, the randomness is obviously at least partly controlled by the dices. Because of this, the first thing to test is that the dices work.*

**Hypothesis: Dices aren’t random.**

**Test (Junit): Assert that the chance of each of the 6 faces rolling on the dice is 1/6.**

**Result: FAIL. The odds for all faces except Spade is 0.20. The odds for Spade are 0.**

**Conclusion:** Hypothesis is correct; the bug is somewhere in Dice.roll() which is a single line that calls DiceValue.getRandom();

**Hypothesis:** DiceValue.getRandom doesn’t return a random value**.**

**Test (Junit): Assert that the chance of each of the 6 faces being returned by** DiceValue.getRandom **is 1/6.**

**Result: FAIL. The odds for all faces except Spade is 0.20. The odds for Spade are 0.**

**Conclusion:** Hypothesis is correct; the bug is somewhere in DiceValue.getRandom(); which is a single line that calls DiceValue.getRandom();

The problem line is obviously in line 26, int random = RANDOM.nextInt(DiceValue.SPADE.ordinal());. According to (Oracle, 2016A), ordinal() will return the zero-based index of the enum. In this case it would be 5. According to (Oracle, 2016B), Random.nextInt(value) returns a value between 0 (inclusive) and the specified integer, 5 (exclusive). This means that 4 can never be returned, which explains the results of the previous hypothesis results.

**Hypothesis:** Changing line 26 to int random = RANDOM.nextInt(values().length) will result in Dices functioning correctly.

**Test (Junit): Assert that the chance of each of the 6 faces rolling on the dice is 1/6 with the line changed.**

**Result: SUCCESS. The Dices now work correctly.**

**Conclusion:** While the dices now work correctly, the game still gives the same results.

Looking at how the playRole() interacts, with the Dice, it doesn’t actually use the value returned by roll(), but instead gets the value using Dice.getValue(). For this reason, the next step is to check that the value returned by getValue() is just as random as that returned by roll().

**Hypothesis: getValue is not returning the random values.**

**Test (Junit): Assert that getValue returns the value that was rolled.**

**Result: FAIL. getValue returns the correct value 1/6 of the time.**

**Conclusion:** The issue is in getValue.

The next step is to look at where the value being returned by getValue() is being set. It is only being set in the constructor, which explains my initial suspicions regarding something happening during initialization that is determining the results in some way.

**Hypothesis:** Setting value to the random face before returning it from Dice.roll() will make the game work properly.

**Test (Junit): Assert that the chance of winning the game is 0.42 after making the change.**

**Result: SUCCESS. The Dices now work correctly.**

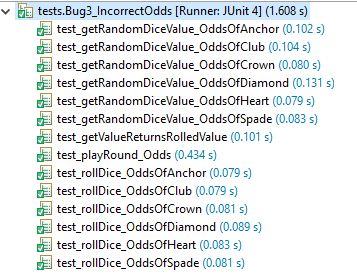
**Conclusion:** While the dices now work correctly, the game still gives the same results.

The odds of the game now appear to be correct.

## Before and After Screenshots

With first change:

With both changes:



# References

Oracle. (2016A, October 1). *Class Enum<E extends Enum<E>>.* Retrieved from Oracle Help Center: https://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html

Oracle. (2016B, October 1). *Class Random.* Retrieved from Oracle Help Center: https://docs.oracle.com/javase/7/docs/api/java/util/Random.html