Bugs 1, 2, and 3 Investigation

Legend: **blue** words are classes, **green** words are methods, and **purple** words are instance variables. (If a word is not coloured, that means it is being used in its natural English sense).

Contents

[Description 2](#_Toc463822038)

[Static Review 3](#_Toc463822039)

[Hypothesis Testing 6](#_Toc463822040)

[Hypothesis 1 6](#_Toc463822041)

[Testing one match 6](#_Toc463822042)

[Testing two matches 7](#_Toc463822043)

[Testing three matches 8](#_Toc463822044)

[Hypothesis 2 9](#_Toc463822045)

[Hypotheses 3 and 4 10](#_Toc463822046)

[Resolution 13](#_Toc463822047)

[Risks 13](#_Toc463822048)

[Testing the solution 13](#_Toc463822049)

[Conclusion 16](#_Toc463822050)

# Description

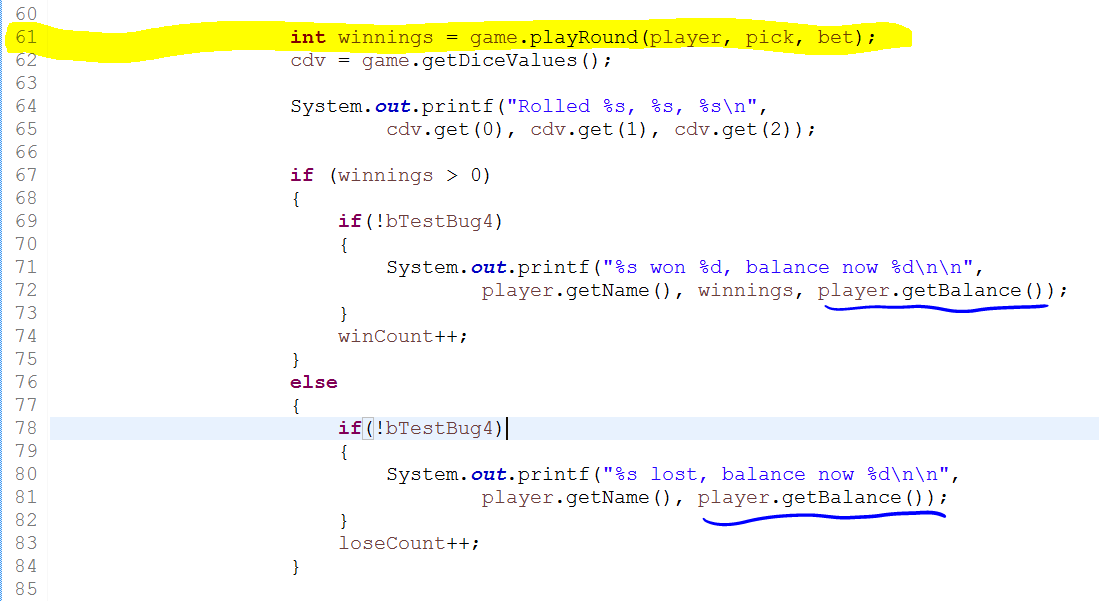
Whenever the player makes a match, he is not paid out correctly according to the rules sheet.

* When he makes one match, he just gets his money back instead of winning an amount equal to his bet.
* When he makes two matches, he wins only the amount that he bet, not twice the amount he bet.
* When he makes three matches, he wins only twice what he bets, not three times.

The reason all three of these scenarios are in the same investigation document, is because our initial hypothesis (before doing any investigation or static review) is that they all have the same cause, and therefore will all be resolved with the same change. Spoiler alert: if this document still exists at the time of marking, then this hypothesis turns out to be true (or otherwise they would have been split up into separate documents).

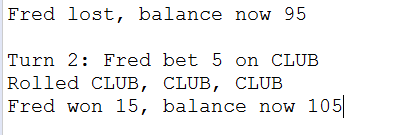
# Static Review

The first place to look in during main where the winnings get calculated. Looking here:



We see that winnings are calculated by **Game**’s **playRound**. After that, they are only displayed to the screen along with the player’s balance (from Player’s getBalance).

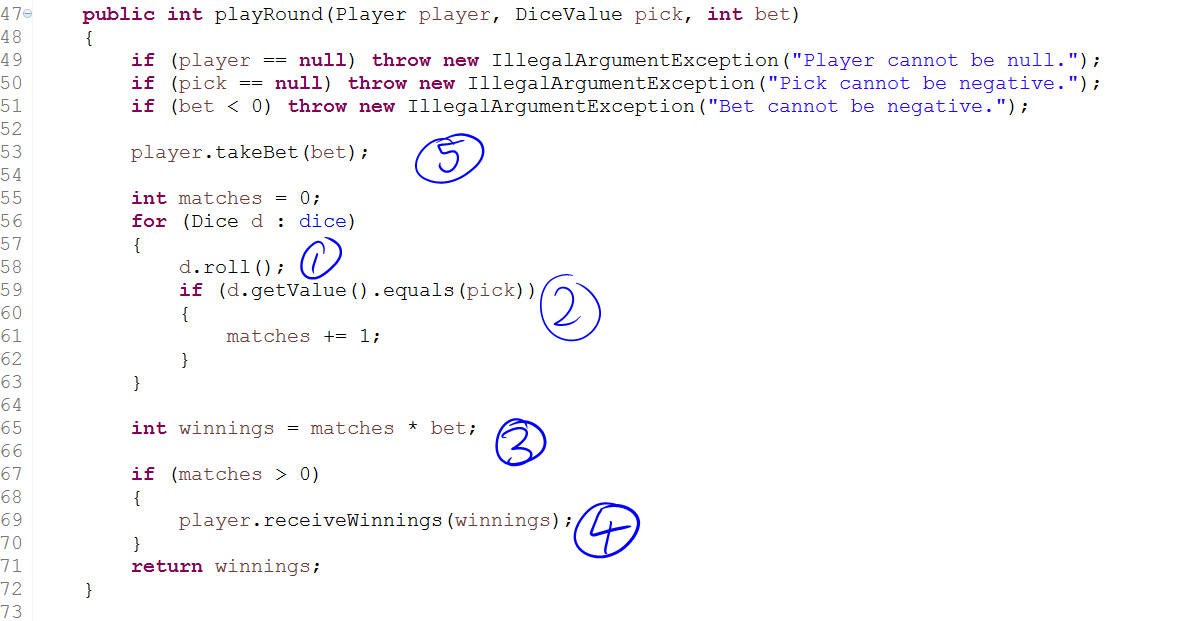
Let’s look at an example output to see what might be wrong.



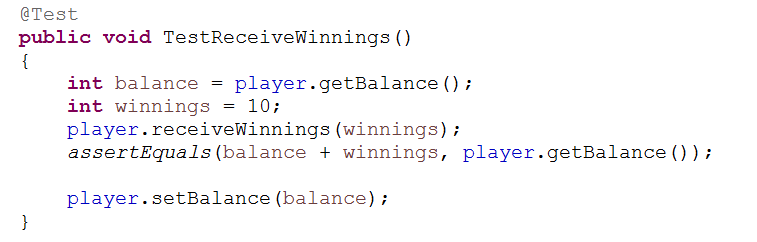
We can see that the winnings amount displayed is correct (15 = 5X3 and there were three matches) but the balance is incorrect (95+15 = 110, not 105).

There are two possibilities. Either **playRound** is not changing the player’s balance correctly, or getBalance is not displaying the balance correctly (and maybe it is correct internally). If getBalance was not displaying the amount correctly, and it was off by 5 in every case, we would expect the first line in the above screen snip to also be 5 off (and thus the change in balance would look correct – 5 off on both cases for a 15 total change). But it isn’t. Therefore, it’s can’t be getBalance’s fault.

So that leaves us just with **playRound**. Let’s have a look.

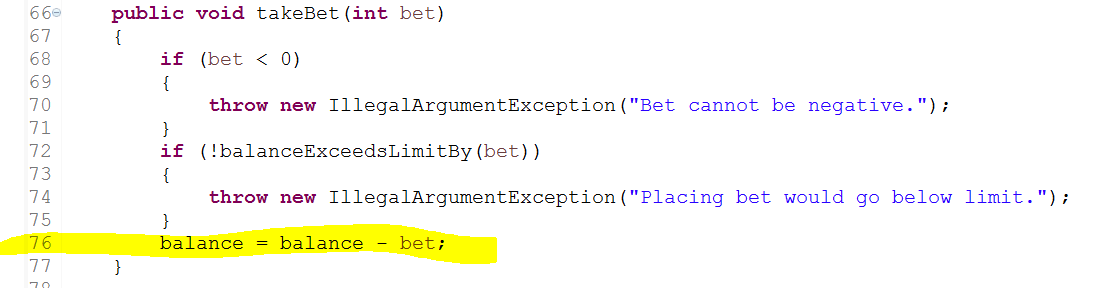


This is the whole **playRound** function. At (1) the dice are being rolled, and from fixing Bug 4 we know that this works as intended. At (2) we are comparing the value of the dice with the pick. One hypothesis is that this works fine. We can test it to confirm (see next section). At (3), if (2) produces the correct number of matches there is no reason why this wouldn’t work (both matches and bet are ints, so there should be no issue with them being multiplied). At (4), the method **receiveWinnings** adds the winnings onto the player’s balance. This has been unit-tested in **PlayerTest** so should work:



As you can see, the balance is recorded before anything happens, the winnings are set, the method is called, and there is an assert that the player’s balance has changed as expected. Just in case there is any wriggle-room for doubt, we will make this a hypothesis and test it (that the player’s balance increases by the ‘winnings’ amount).

So then if everything we just looked at works as expected (assuming testing passes our two hypotheses so far), where, then, does the bug come from? It must be at number (5), with the **takeBet** method. I don’t even know what this is for, so let’s have a look:



The program proceeds in normal fashion each time we have run it, so we should be able to presume that neither of the conditions for exceptions being thrown are ever met. Therefore, we can concentrate on the last line (highlighted) and treat it as the only line in the method.

So, **takeBet** deletes one lot of the bet amount from the player’s balance.

We can see immediately what is wrong here. Looking at (3) from the **playRound** screen snip, the winnings are calculated as number of matches times the bet amount. According to the rules, this is correct – this amount is meant to be added onto the player’s balance from before the round. However, the balance is changed before the winnings are added on. One lot of bet is removed before the winnings are added on, leading to the player receiving one less lot of bet for each win. For example, one match results in no change to balance (should have increased by one lot of bet, so is one lot of bet worse off), two matches results in one lot of bet added to the balance (should be two lots, so again one lot worse off), and three matches results in two lots of bet added to the balance (instead of three).

If our guesses are right, then all three of these hypotheses need to be tested true:

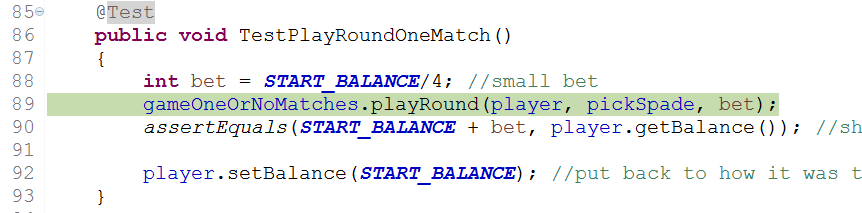
1. The value of the dice is compared successfully to pick and the correct number of matches is calculated.
2. The player’s balance correctly increases by the winnings amount after **receiveWinnings** is executed.
3. The balance is incorrect at the time of executing **receiveWinnings**.
4. The moment the balance becomes incorrect is just after **takeBet** is called. (i.e. to use correct terminology the player’s balance becomes ‘infected’ by **takeBet**).

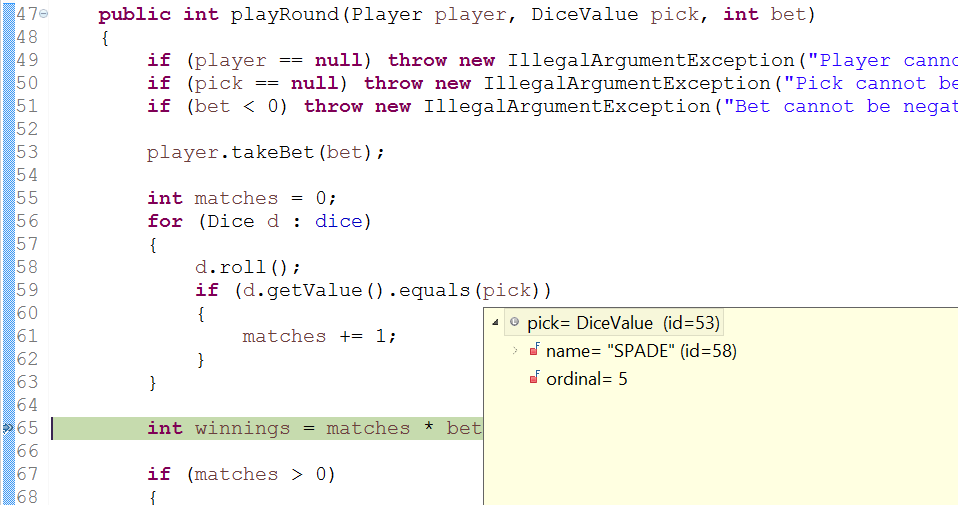
# Hypothesis Testing

## Hypothesis 1

For this one we can use the unit test of the **Game** class, and place a breakpoint in the appropriate test (TestPlayRoundOneMatch, etc). This way the dice are guaranteed to come up the way we want, and we don’t have to step through main for long periods of time until we get the match that we want.

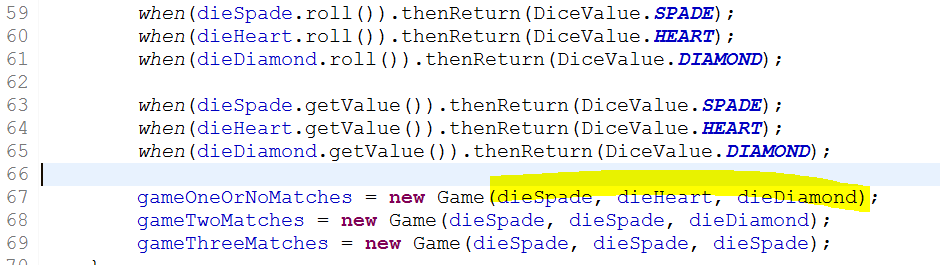
### Testing one match



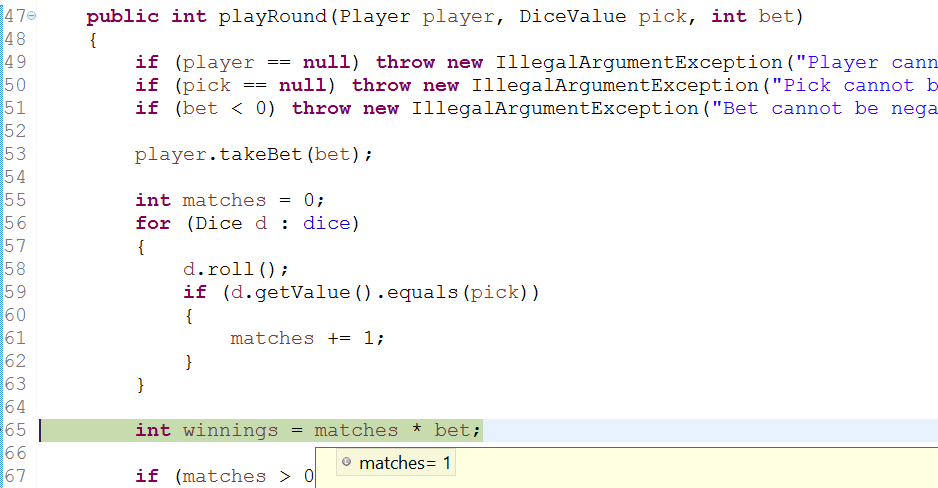
Going into the right test:

The pick is a Spade.

We can’t actually show the values of dice because they are mocked objects. But in the constructor of **GameTest**:



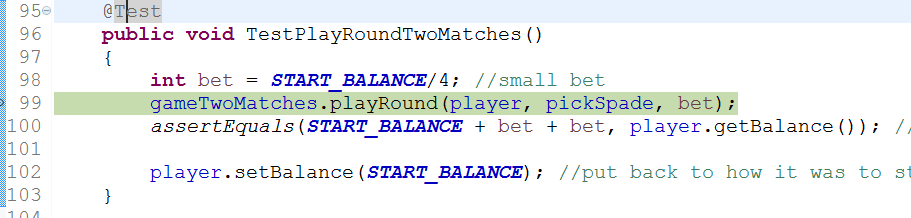
You can see that **gameOneOrNoMatches** will roll a spade, a heart, and a diamond, and the player picks spade. So, we should get one match here.



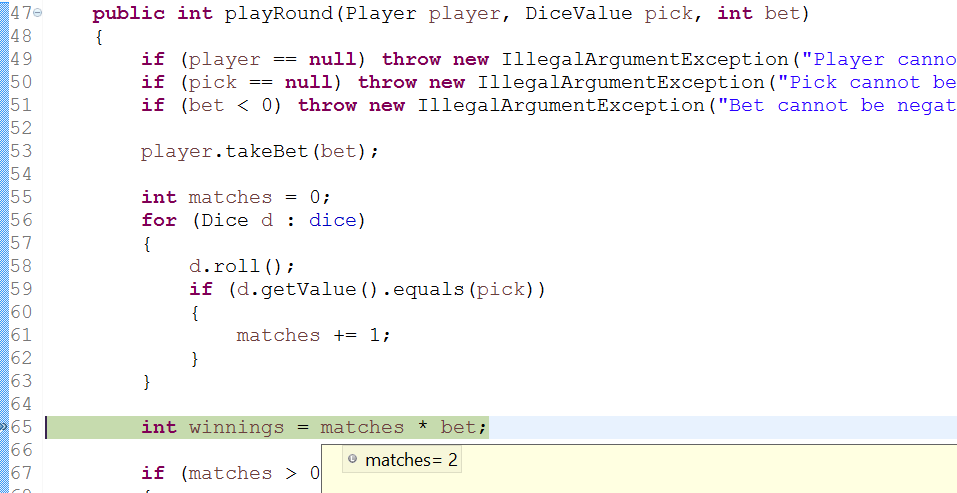
And indeed we do get one match.

### Testing two matches

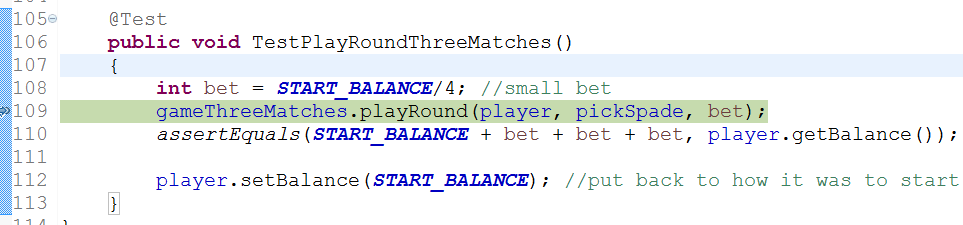
Going into the right test:

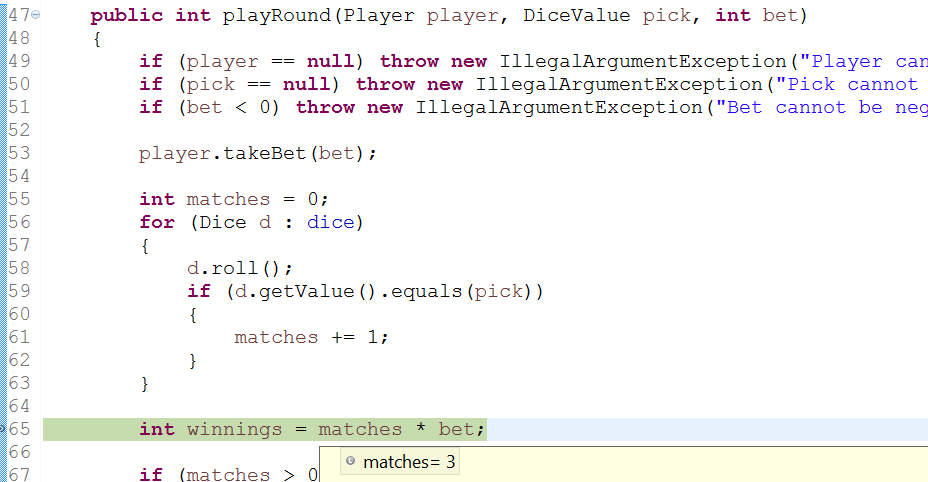


The right number of matches:



### Testing three matches

Going into the right test:

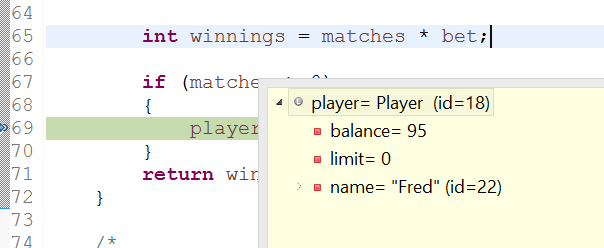
Right number of matches:

So hypothesis 1 is verified. **Game’s** **playRound** can determine the correct number of matches.

## Hypothesis 2

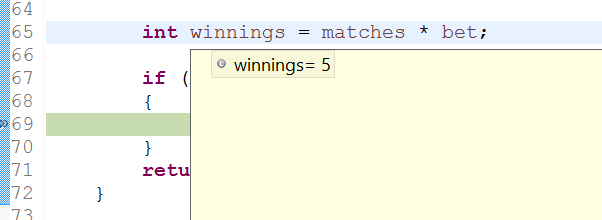
For this one, we will put a breakpoint before **receiveWinnings** is called, run the program normally and see what the balance and the winnings amount is. We will then step to after it is called, and check that the balance increased by the right amount.

Here is the program right before it calls **receiveWinnings**:

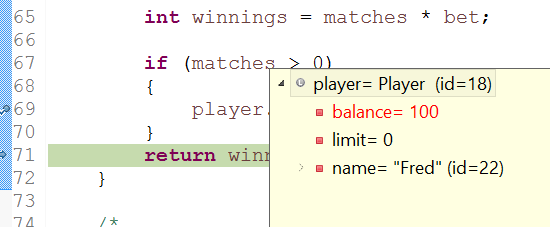


As you can see, the player’s balance is 95.

We also need to know the winnings:



So we would expect that the balance after calling **receiveWinnings** would be 100:



And indeed it is. This verifies hypothesis 2.

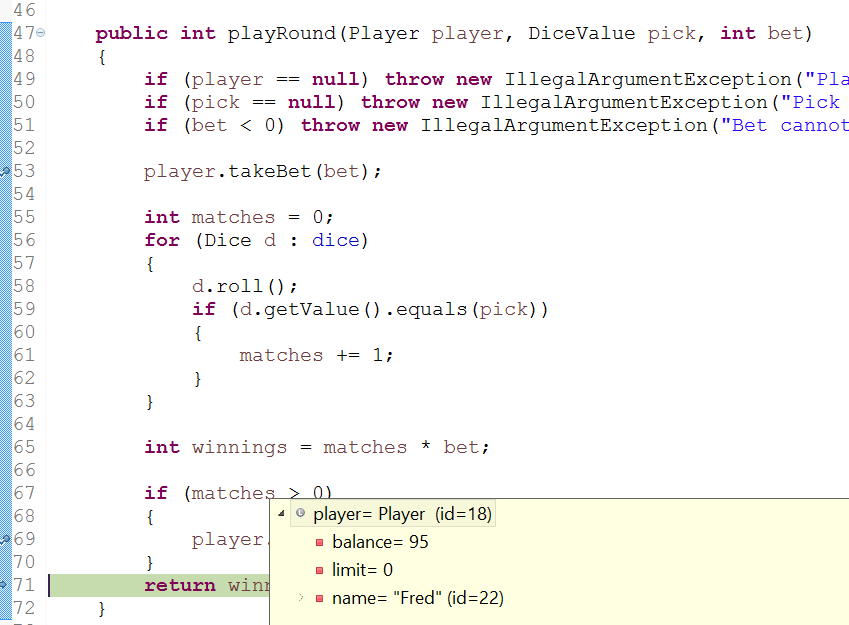
## Hypotheses 3 and 4

To test these ones, we put breakpoints/step as follows:

1. At the end of **playRound**, and have a look at the player’s balance.
2. Then, step to just before **takeBet** and make sure the balance is still the same (the previous breakpoint is to make sure that it hasn’t been changed by anything in between the last round and this one – so it is certain that the infection happens inside the **playRound** method and not outside it).
3. Then, step to just after to confirm that the balance has changed to the incorrect value.
4. Then, step to just before **receiveWinnings**, to make sure the balance hasn’t changed

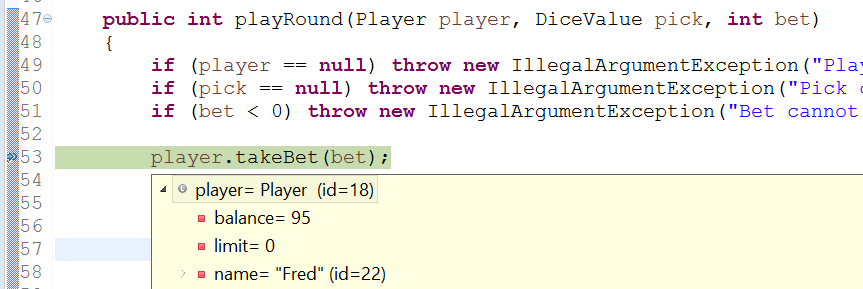
After this, we have already confirmed **receiveWinnings** to be working correctly. At this point, if the program behaves as expected we will have shown that it is **takeBet**, and **takeBet** only, that infects the player’s balance.

1. End of **playRound**:



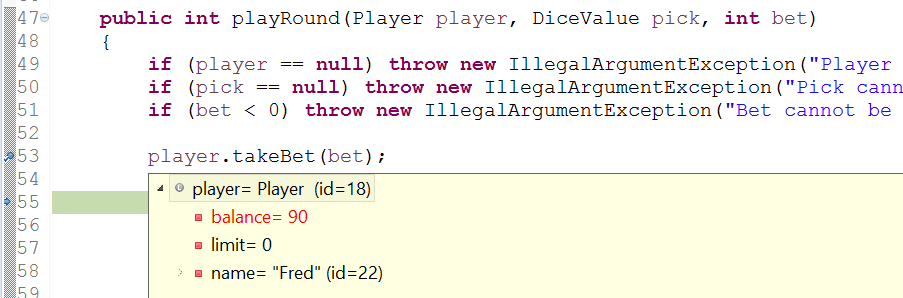
Player’s balance is 95.

1. Just before **takeBet**:



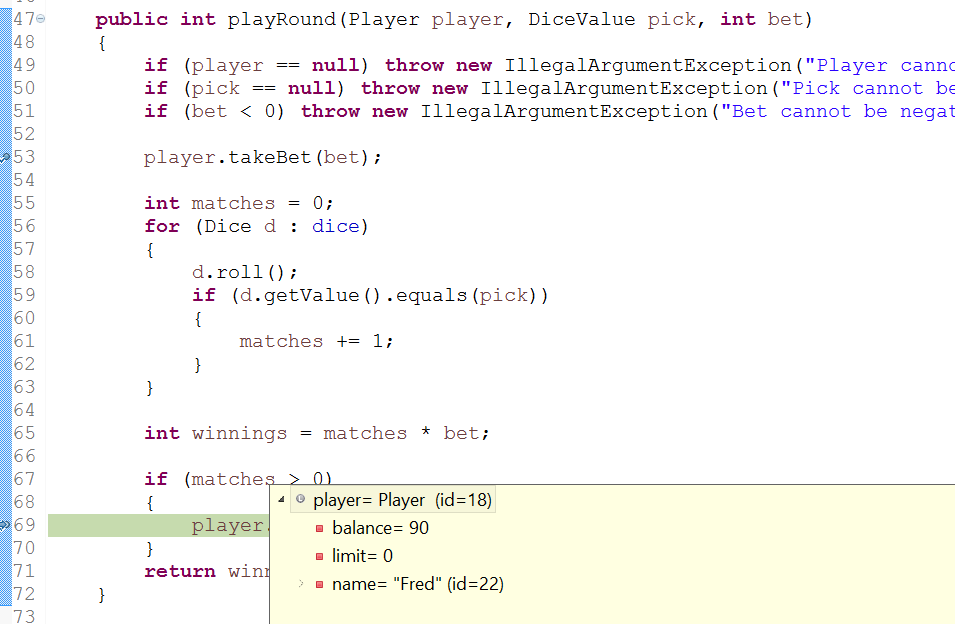
The player’s balance is still 95.

1. Stepping to just after **takeBet**:



Yes, the balance has changed to 90. Tsk, tsk.

1. Just before **receiveWinnings**:



Balance still 90.

Everything that we required to verify our hypothesis has come true, so hypotheses 3 and 4 are verified (that the balance is already incorrect before **receiveWinnings** is called, and it is **takeBet** that causes this infection).

# Resolution

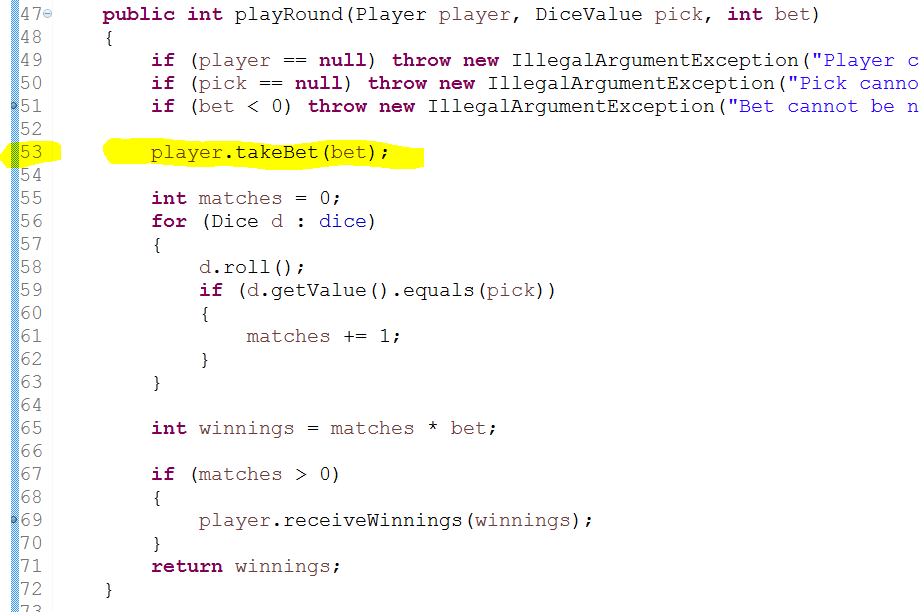
If the player loses, we do want **takeBet** to be called. But if the player wins, it shouldn’t be called. It also doesn’t need to be called so early in **playRound**. As long as it happens before the end of the method, the balance will be correct. There is a section at the end of the method that is a conditional on matches being greater than 0, which calls **receiveWinnings**. This isn’t really necessary, because if matches is 0, then winnings will be zero as well. However, we can use this by adding an ‘else’ statement that calls **takeBet** (so if the player doesn’t get any matches, the bet amount is removed from his balance).

## Risks

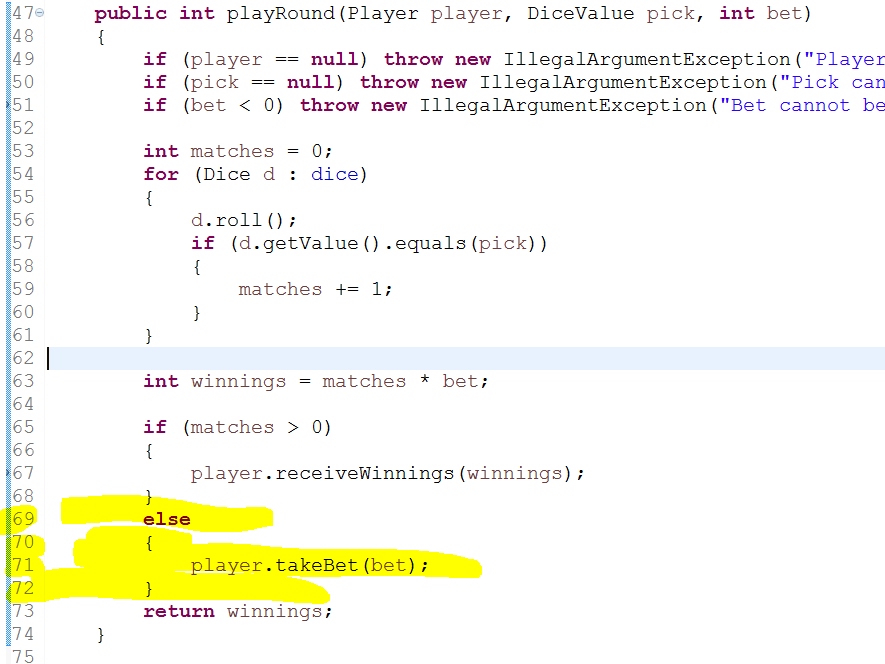
There shouldn’t be any risks for changing the location of **takeBet** in this way. There isn’t anything that relies on the player’s balance during this method (besides **receiveWinnings**, and we’ve seen that this influence is negative).

## Testing the solution

**playRound** is going from this (proposed deletions highlighted):

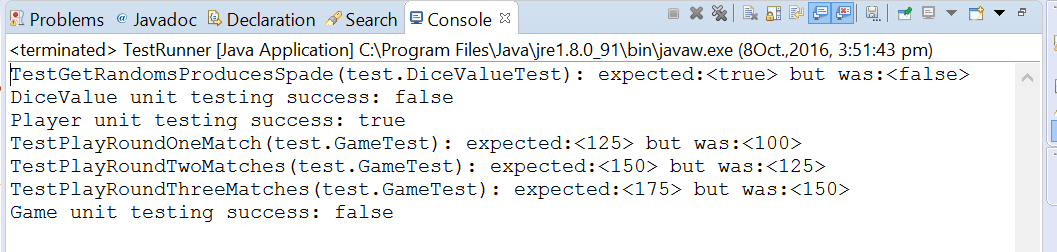


To this (additions highlighted):

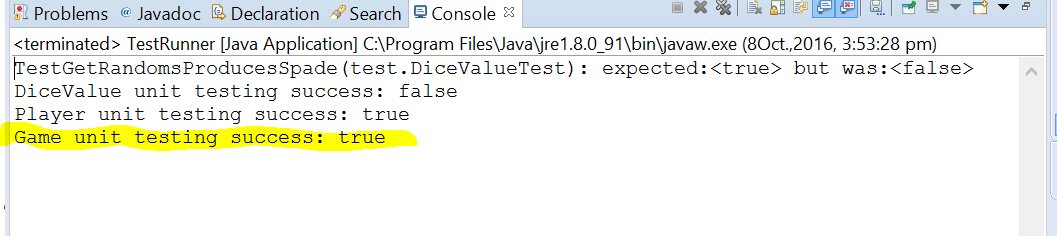


Previously, the **GameTest** unit test failed, because the player’s balance did not change appropriately. Let’s run it again and see if that has changed.

This is before the proposed solution was implemented:

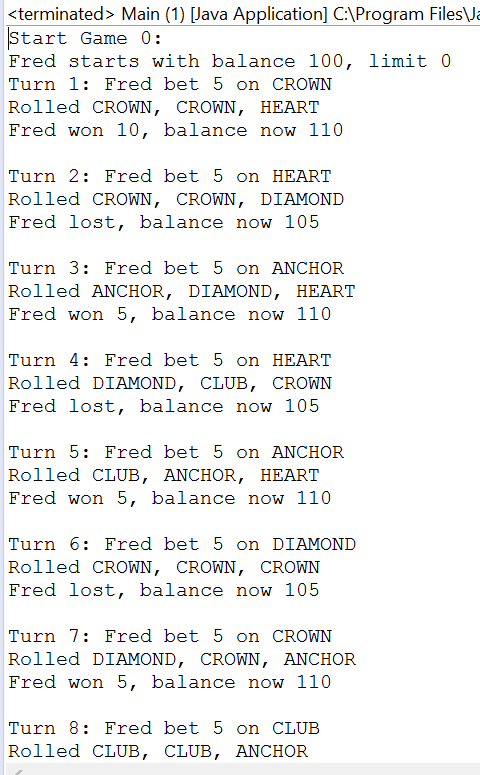


And after the solution is implemented:



Success!

We don’t have to take the unit testing’s word for it though. We can run through the program and highlight some examples of matches that are paid out correctly (and of course if any are not paid out correctly, we will take note of those as evidence of a failed solution to this bug).



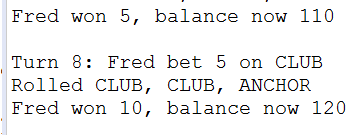
Here we can see on Turn 1, Fred started with $100, bet $5 on Crown, and rolled two Crowns and a Heart. He made two matches so should make $10. And he did: after the roll, his balance is now $110.

On turn 3, Fred has $105, and bet $5 on Anchor. He rolled an Anchor, Diamond and Heart. He made one match so should win $5. And he did: after the roll, his balance is now $110.

On turn 5, Fred has $105 again, and bets $5 on Anchor again. He rolls Club, Anchor, Heart. He made one match so should win $5. And he did: his balance after the roll is $110.

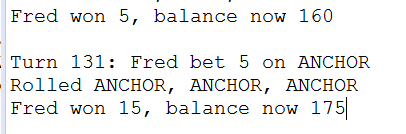
Let’s find an example where he matches two or three times so we can be really sure.

Here, he makes two matches:



He bet $5, so should win $10. He started with $110 and ends with $120, so he does win his $10.

Here’s an example of matching three:



He matched three, and bet $5, so should win $15. He started with $160 and ends with $175, so he does win his $15.

# Conclusion

The bug is now resolved, and no new defects have been introduced by the resolution. Sample successful results can be seen in Bug123Solution.txt.