Bugs 5, 6 and 7 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](#_Toc463821965)

[Static Review 3](#_Toc463821966)

[Bug 7 3](#_Toc463821967)

[Calculating the odds 3](#_Toc463821968)

[Bugs 5 and 6 4](#_Toc463821969)

[Hypothesis testing 7](#_Toc463821970)

[Hypothesis 1 7](#_Toc463821971)

[Hypothesis 2 8](#_Toc463821972)

[Hypothesis 3 8](#_Toc463821973)

[Resolution 9](#_Toc463821974)

[Risks 9](#_Toc463821975)

[Testing of solution 10](#_Toc463821976)

[Conclusion 12](#_Toc463821977)

# Description

Spades are not able to be rolled or guessed, and the winning percentage odds of the game are not correct.

# Static Review

## Bug 7

If we take bugs 5 and 6 at face value, without any testing, and assume that the game really does never roll or guess spades, this leads to an explanation for bug 7 (which is why they are placed together in the first place).

We can calculate the odds of winning in a game with no bugs, and not just take the 42% given in the initial bug report at face value. Then, we can calculate the odds of winning with the spades removed from the game and see if it is anywhere near the 48.8% found by testing.

### Calculating the odds

Let’s say the number of possible faces is ‘n’.

There are possible ways to roll the dice (first roll has n possibilities, second roll has n possibilities, third roll has n possibilities). There is a chance of losing (if you pick a particular symbol, the dice now has n-1 ways to come up that doesn’t match your chosen symbol, and there are three rolls).

So therefore there is a

chance to win, and the winning percentage should be

Let’s try this with n = 6 and see if we get 42.0%.

29ncee of losing get 42.0%.nninre are three rolls).ice now has n-1 ways to come up that doesn;t

This looks pretty good. The initial bug report reported it as 42% but it could have been rounding.

Now let’s try with n = 5 (all the faces without spades).

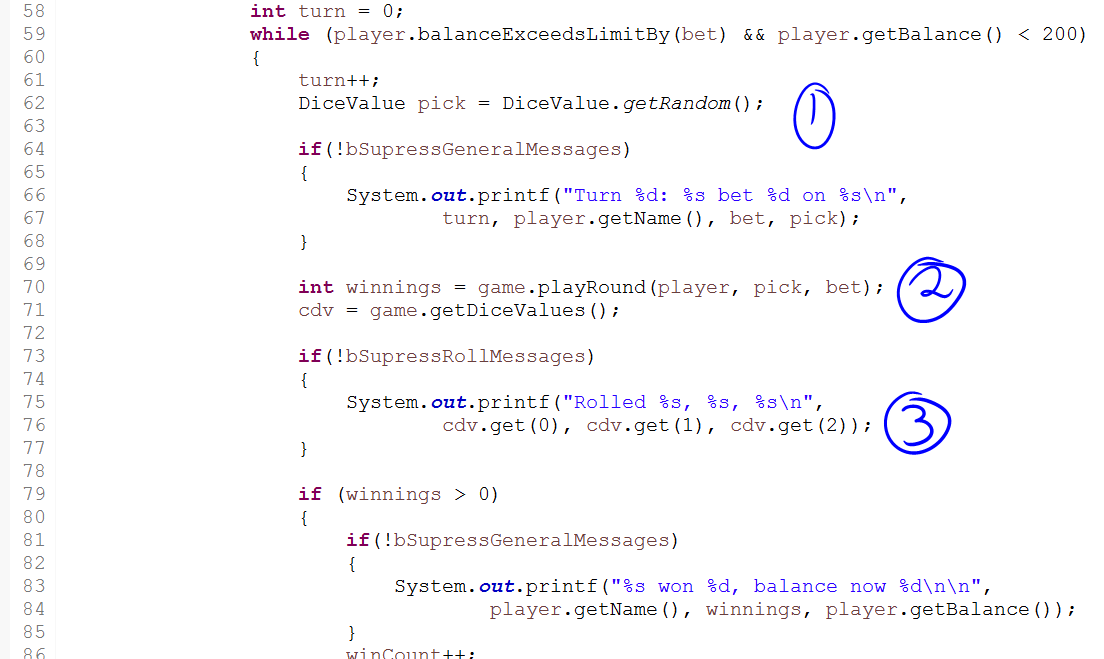
29ncee of losing get 42.0%.nninre are three rolls).ice now has n-1 ways to come up that doesn;t

Wow, 48.8% exactly! Would be a pretty huge coincidence if it wasn’t caused by the missing spades. But we will of course do plenty of testing. And we will test this bug again when we have solutions for bugs 5 and 6, to see if the winning percentage changed to 42.1%.

## Bugs 5 and 6

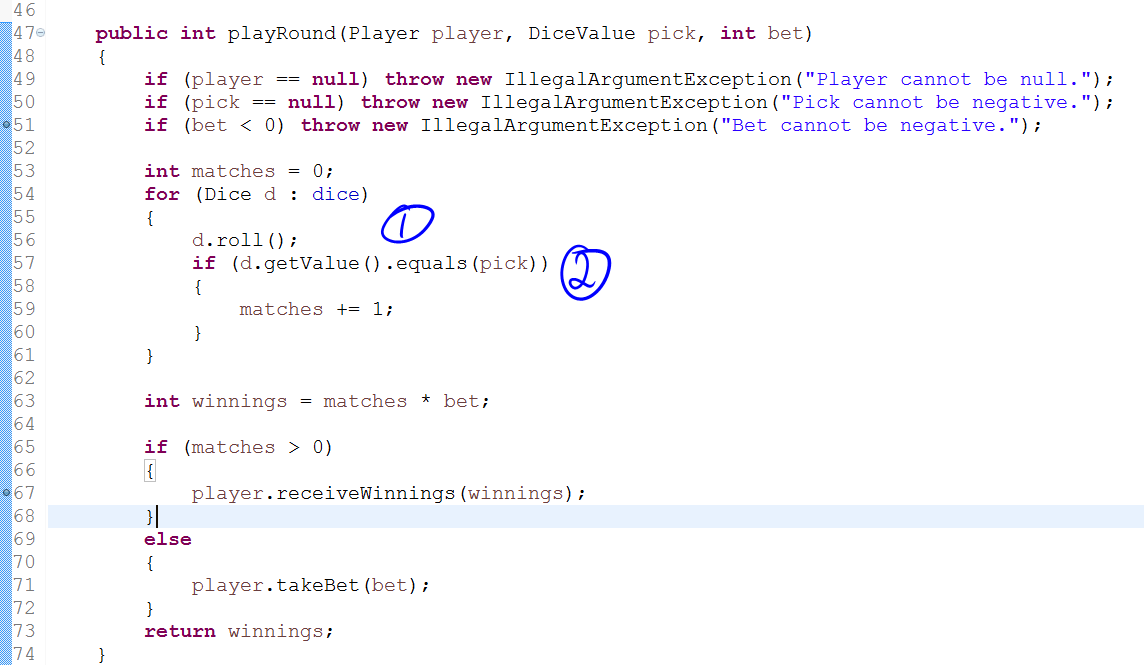
Since these bugs are about what can be rolled, it makes sense to start with where the dice are rolled for each game (for bug 5 – spades never being rolled) and where the player choses the pick (for bug 6 – player never chooses spades).

If we have a look at main we can see that the actual game happens inside **Game**’s **playRound**:



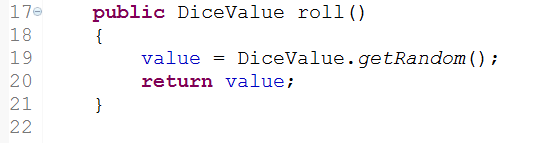
At (1), the player picks the face they wish to bet on (we will come back to this for bug 6), at (3) the rolls are being displayed, so therefore the rolls and such must be determined during (2) (which is **Game**’s **playRound**).

Now let’s look at **playRound** and try to determine when the dice are rolled to get new values:

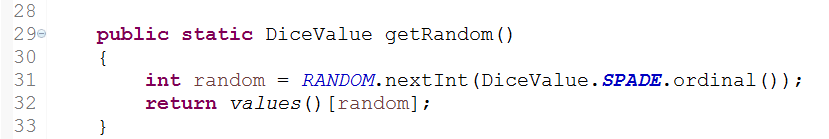


At (1), it looks like the dice are being rolled (because of the helpful method name). But at (2), the values of the dice are being compared against the pick, so we can be fairly sure the dice really did get rolled at (1) (otherwise they’d be being compared to the old values).

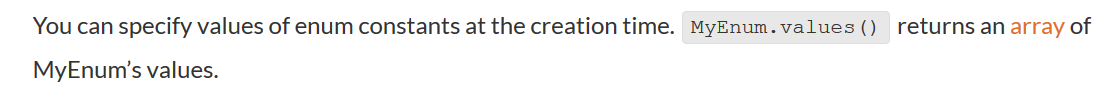
So let’s have a look inside roll.



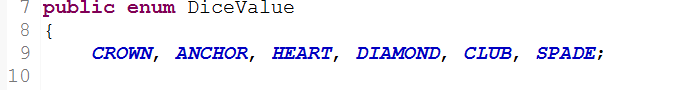
So **DiceValue**’s **getRandom** is being called and the result is being set to value (which would be what **Dice**’s **getValue** will return). Let’s have a look at **getRandom** now.



According to <http://crunchify.com/why-and-for-what-should-i-use-enum-java-enum-examples/>:



So on line 32, we are retrieving the enum value according to its index. And our enum values are:



So CROWN will be 0, ANCHOR 1, etc.

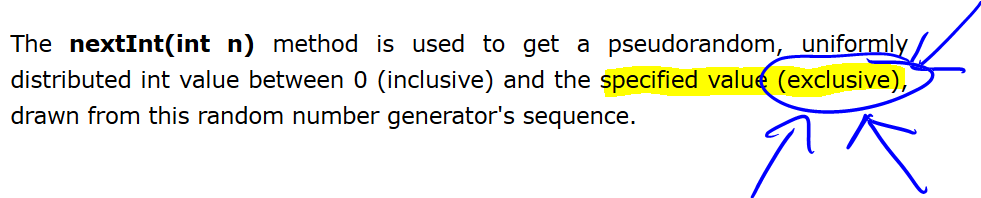
And ‘random’ is this index. Now let’s see how random is determined.

First of all the RANDOM variable is defined as:



So let’s see what Random.nextInt does.

According to <https://www.tutorialspoint.com/java/util/random_nextint_inc_exc.htm>:



Random.nextInt(n) will get a random number between 0 and **less than** n (as highlighted above – exclusive of n). So this means our **getRandom** method must be getting all the ordinal values not including spades (as it is the last one).

This explains bug 5 perfectly. The roll method called **DiceValue**’s **getRandom**, and **getRandom** can return any **DiceValue** except for spades.

Now back to bug 6. As we noted earlier, the pick came from point (1) in the screen snip of main where we had the line:



Through investigating bug 5, we found that roll called **DiceValue**’s **getRandom**, which infected **Dice**’s value (which is of the class **DiceValue**). Here, **DiceValue**’s **getRandom** is being called directly and the result placed in pick. We already know that **getRandom** infects the value it returns, so we can see it infects pick (so our initial hypothesis that bugs 5, 6, and 7 are related is holding up so far).

Here are our hypotheses:

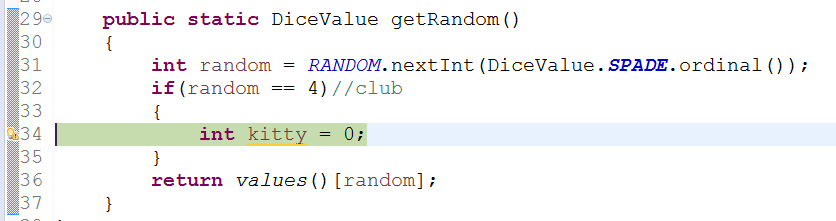
1. **getRandom** can’t ever return spades
2. Bugs 5 and 6 are caused by the same programming error
3. Bug 7 is caused by the same programming error as bugs 5 and 6

# Hypothesis testing

## Hypothesis 1

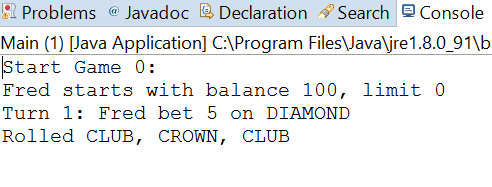
To test this one, we will make a little debug section in the **getRandom** method that will trigger if the method would ever return spades. Then, if this section does get triggered, we know it is not **getRandom**’s fault and there is another problem somewhere else in the code where the spades are getting lost. If, however, this section is never triggered, then we know that **getRandom** really cannot produce spades.

We will start off by triggering when we get clubs, to make sure the trigger point actually works (just in case we do something wrong and it can never be triggered, and then we blame **getRandom** unnecessarily).

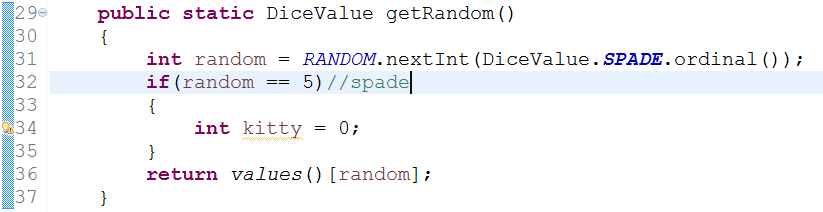


(There is actually a breakpoint there – Eclipse has a silly design where it puts warnings over the top of your breakpoints.) We can see the debug section does get triggered when we hit a club.

If we step through a bit further we see the clubs be printed:



So now we replace the 4 with a 5 (for spades – it is the 6th enum value and so has an ordinal value of 5) and see if our breakpoint gets triggered.



(Again, there is actually a breakpoint there but you can’t see it.) Nope – the breakpoint does not get triggered! Therefore, we have verified that **getRandom** cannot return spades as an answer.

## Hypothesis 2

Through the static review we determined that both **Dice**’s roll (which is the beginning of the sequence that leads to bug 5 being expressed) and **DiceValue**’s **getRandom** (which is the beginning of bug 6 being expressed) both end up at **DiceValue**’s **getRandom**, which is a common method. Through verifying hypothesis 1 (see above) we have shown that **getRandom** causes an infection in the **DiceValue** that it returns, so we can conclude that both bug 5 and bug 6 are caused by the same error.

## Hypothesis 3

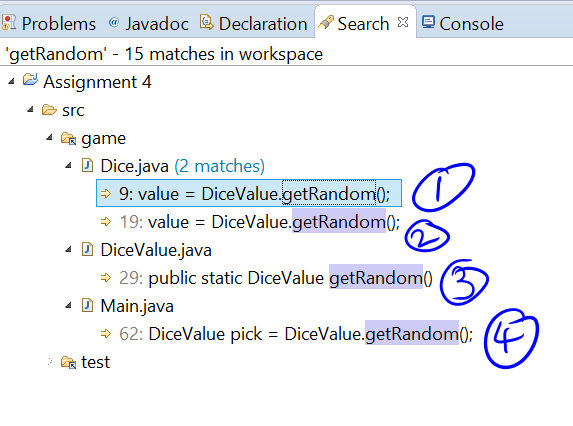
We can’t really test this one directly. We have to wait until we have implemented a solution, then retest with Bug7Replication.java to determine if the odds have changed.

# Resolution

**getRandom** is calling Random’s nextInt with an argument of DiceValue.SPADE.ordinal(). According to the static review, nextInt(int n) cannot return n, it can only return values less than n (and 0 or higher). So what we should do is add 1 to the argument in nextInt. That way, it will not return DiceValue.SPADE.ordinal() + 1 (since it can’t return the value of its argument) but it can return DiceValue.SPADE.ordinal() (being one less than the argument). We add 1, instead of 0.1 or something, because the argument to nextInt must be an integer. The next biggest integer means adding 1. This way we don’t get any results that are NOT one of the ordinals of **DiceValue** (which would cause exceptions).

### Risks

There could be places in the code that are calling **getRandom** and expecting to not receive spades as an answer, and maybe now receiving spades could cause a problem for their logic. We will do a search of **getRandom** and determine if there is anywhere besides the two places we identified (in roll and in main to determine the pick).

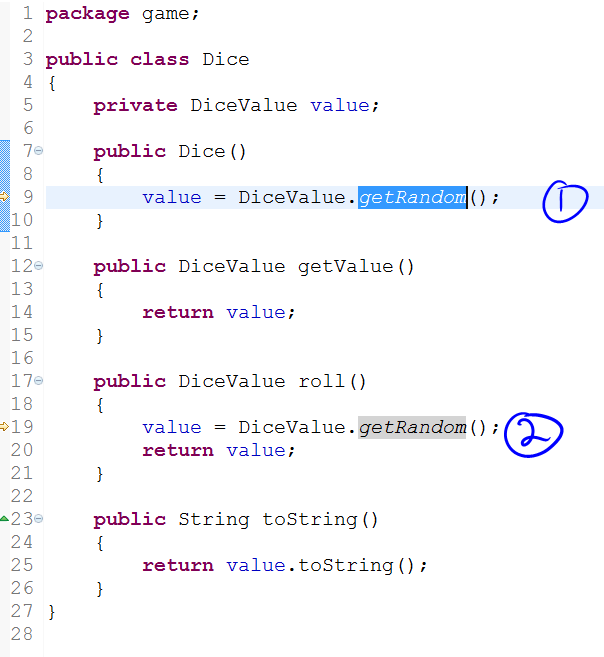


For (1) and (2), one of these is the roll method, and the other one is something else. We will have to investigate both to see what the other one is.

For (3), this is the method definition.

For (4), this is the pick determination.

Looking at (1) and (2), we’ll have a look at the whole Dice class (it’s only small):

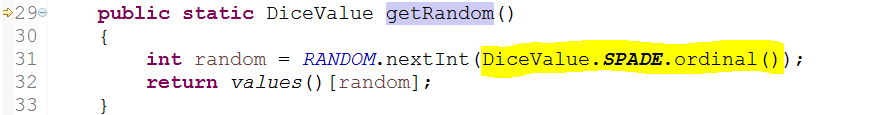


(2) is our roll method, so that’s fine. (1) is the constructor. This must just be so that the dice gets a random face to begin with. There should be no reason why we wouldn’t want spades on the initial roll of the dice, but we do want spades on every subsequent roll. So this is fine. (This should really be roll not **getRandom** to reduce class coupling, but as it doesn’t affect anything, we will leave it. Better to only change things that relate to the debugging effort, not randomly “improve” code for no good reason)

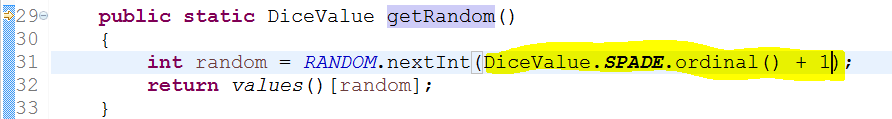
Therefore, there are no risks to changing the **getRandom** method as specified.

### Testing of solution

We are going from this (parts pending change highlighted):

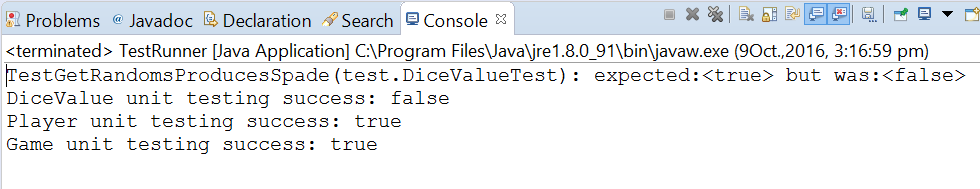


To this (changed parts highlighted):



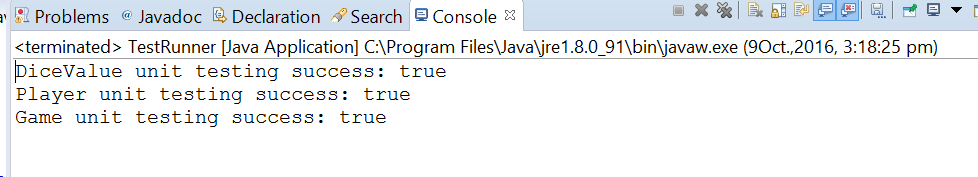
Now let’s test each of the bugs. We will run through the unit testing for **DiceValueTest**, since we have a test for **getRandom** to determine whether each one of the faces comes up in 100 rolls. This will test both bug 5 and bug 6 (since they both stem from the problem in **getRandom**).

This is the result prior to implementing the solution:



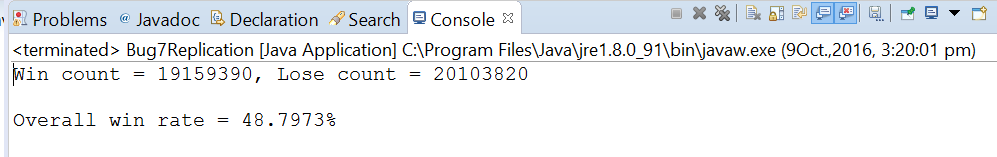
As you can see, **DiceValueTest** fails because the test **TestGetRandomsProducesSpade** fails.

Now after implementing the solution:

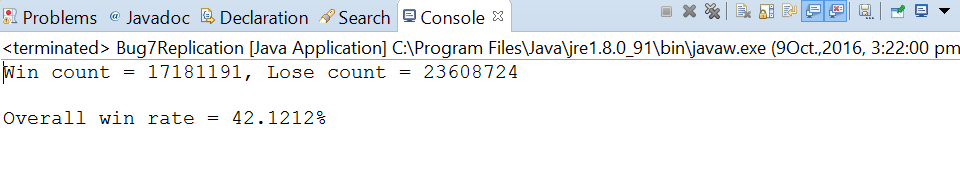


**DiceValueTest** now passes! So the solution works for bugs 5 and 6.

Now we can finally test whether this fixes bug 7. We run Bug7Replication.java which will run main 2000 times to be really sure of the winning percentage. To remind everybody, this is the result before the solution is implemented:



And this is the result afterwards:



We were expecting a theoretical 42.1296%, but this is certainly within a very small margin of error. We can say this test passes (and hypothesis 3 is now verified).

# Conclusion

With the implemented solution, all the tests pass. So bugs 5, 6 and 7 are now resolved.