**Step 1: Replication**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bug Number 3** | | Odds in the game do not appear to be correct. | | |
| **Bug Description** | | Crown and Anchor games have an approximate 8% bias to the house. So the win : (win+lose) ratio should approximately equal 0.42. This does not appear to be the case. | | |
| **Reproduction Test Description:** | | Run simulation, find instance where player wins on one match and compare balance to previous balance. | | |
| **Pre-conditions** | | N/A | | |
| **Post-conditions** | | N/A | | |
|  | **TEST STEP** | | **EXPECTED TEST RESULTS** | |
|  | Run the simulation (execute Main.main()). | | Simulation starts & prints output of simulation (100 games and stats) to console. |  |
|  | Scroll to end (end of simulation) of output in console. | | Output ends with stats on the overall simulation – with the last line showing the win count, lose count and win+lose ratio. |  |
|  | Evaluate the win+lose ratio | |  |  |
|  | Re-run the simulation several times, noting the dice values rolled in the last round – and the win+lose ratio – for each run | | There appears a relatively high occurrence of win+lose ratios of 0.60+ (may have to rerun the simulation many times).  There is also perhaps a correlation between the values rolled in the last round and the ratio – when there are no duplicate values the ratio appears high ~ 0.60, but when there are 2 dice with the same values the ratio is approximately normal ~ 0.40.  Regardless, the ratio always appears to be one of a limited set of significantly differing values whenever the simulation is run – e.g. ~0.40 or ~0.60… |  |

**Step 2: Simplification**

Note – rather than jumping to conclusions, here I decided to just quickly test 2 different aspects – first consistency of the win ratio across games, and then the distribution of the values rolled

|  |  |
| --- | --- |
| **Automated Test 1** | TestBug3  hundredRunsOfHundredThousandRounds\_ratioShouldBeCorrectAndConsistent() |
| **Test Description** | Given many games with many rounds, when we evaluate the win ration then the ratio should be approximately correct AND the ration should be consistent across the games. |
| **Test Output** | The test fails as the win ratio is still significantly higher than expected (but is consistent). |

|  |  |
| --- | --- |
| **Automated Test 2** | TestBug3  threeHundredThousandRounds\_ratioDiceValuesHouldBeEqual() |
| **Test Description** | Given many dice rolls, when we count the rolls of each symbol, each symbol should have been rolled and the rolls should be approximately evenly distributed across each symbol. |
| **Test Output** | The test fails as one of the symbols is never rolled (see bottom part of screenshot) |

**Step 3: Tracing**

|  |  |
| --- | --- |
| **Debugging Log** | For Bug3: using TestBug3 |

**Note:** as the second test failure showed that one symbol is never rolled, that is clearly the test to focus on first.

**Debugging Preparation:** Observation of TestBug3 threeHundredThousandRounds\_ratioDiceValuesHouldBeEqual()

|  |  |
| --- | --- |
| **Null Hypothesis** | The test works! |
| **Prediction** | After many dice rolls, each symbol should have been rolled. |
| **Observation** | After 300,000 one symbol was not rolled at all. |
| **Conclusion** | The hypothesis is rejected. |

**Hypothesis 1**

|  |  |
| --- | --- |
| **Null Hypothesis** | The invocation of d1.roll() in line 67 does not return a new value, and thus does not assign a new value to dv |
| **Prediction** | dv remains the same through all of the iterations, thus some symbols are never returned. |
| **Observation** | After a couple of iterations, it is clear that d1.roll() does in fact update the value of dv  dv is sane, although uninitialized at the start of the loop (first screenshot).  In the first iteration, dv is sane – with a value of ‘CLUB’, and in a later iteration it is also sane, as it has a new value of ‘ANCHOR’ |
| **Conclusion** | The hypothesis is rejected. |

**Hypothesis 2**

|  |  |
| --- | --- |
| **Null Hypothesis** | As we know that d1.roll() returns new values, the next null hypothesis must be that d1.roll() must return all possible values for a dice. |
| **Prediction** | Over many iterations dv will have been set to all potential values, thus the distribution count will reflect that. |
| **Observation** | Not all values are returned (30 iterations+). At the start of the loop, the state is sane – the loop variable is not initialized and there have been no increments to the counts for the different values.    After 30 iterations, it is likely that the state is not sane – one of the counts has not been incremented as dv has still not been set to one of the potential values. |
| **Conclusion** | The hypothesis is rejected. |

**Hypothesis 3**

|  |  |
| --- | --- |
| **Null Hypothesis** | As we know that d1.roll() returns new values, but not all potential values, the next null hypothesis must be that the invocation of DiceValue.getRandom() within d1.roll() allows the return of all potential values |
| **Prediction** | It is possible for the variable random to be assigned all potential values from the expression RANDOM.nextInt(DiceValue.SPADE.ordinal() |
| **Observation** | Not all values can be returned.  The value to be returned by DiceValue.getRandom() is determined by the statement  int random = RANDOM.nextInt(DiceValue.SPADE.ordinal()) - see below.    This produces a valid ‘random’ value – – see screenshot below.    However, we can see in the enum that there are 6 potential values. If we set the variable random to 0 (the index of the first value in the array) then the value ‘CROWN’ is returned – as expected – see screenshot below.    If we set the variable random to 5 (the index of the last value in the array) then the value ‘SPADE is returned – as expected – see screenshot below.    However, if the expression DiceValue.SPADE.ordinal() is evaluated – it equates to ‘5’.    This means that the statement  int random = RANDOM.nextInt(DiceValue.SPADE.ordinal());  equates to  int random = RANDOM.nextInt(5);  And that can not result in the value 5 being produced – this is a classic case of the off-by-one-error.  In order to allow a range of values from 0 to 5, RANDOM.nextInt(6) is required. |

**Step 4: Resolution**

|  |  |
| --- | --- |
| **Design** | From the debugging step above, we know that the expression  int random = RANDOM.nextInt(DiceValue.SPADE.ordinal());  Cannot produce the value 5, which in turn means that DiceValue.getRandom() cannot return the value ‘SPADE’    Rather than simply updating the statement to:  int random = RANDOM.nextInt(DiceValue.SPADE.ordinal() +1);  It would be better to use the size of the Map that stores the values (thus if new values are added this method will automatically allow the return of those new values) |
| **Confirm automated test shows resolution of bug** | Both tests in BugTest3 now pass after making that change – see first screenshot below.    In this first test, we can that the trace for the a game of 100,000 rounds shows a win ratio approximately equal to the expected one (0.42).    And in the second test we can see that a value has the approximate expected distribution count |
| **Confirm user test shows resolution of bug** | Carrying out the user-reproduction test (running the simulation by executing Main.main()) now shows that the win ratio is no longer one of a small set of values (i.e. 0.40 or 0.60).  However, as the win ratio is not close to 0.42, this still warrants further attention – see next bug – Bug 4 Simulation turns are not random. |