## Asg 4 - Debugging

Value: 20%

Due date: 16-Oct-2015

Return date: 06-Nov-2015

## Submission method options

EASTS (online)

Alternative submission method

Bug 1: Game does not pay out at correct level

When player wins on 1 match, balance does not increase.

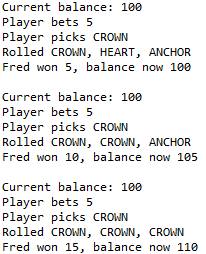
# Task 1 – Replication

| **Test Name** | Winning wager payout increase balance Test | | | |
| --- | --- | --- | --- | --- |
| **Test Description:** | Tests the increase of the Player object’s balance variable when a wager is won for all payout levels; 3 to 1, 2 to 1, and 1 to 1. | | | |
| **Pre-conditions** | * Main object exists * Dice object exists * Player object exists * Game object exists * DiceValue list exists * One winning bet has occurred | | | |
| **Post-conditions** | * The Player’s balance increases by 5 (standard bet) multiplied by the odds, resulting in an increase by 15 for 3 to 1 odds, 10 for 2 to 1 odds and 5 for 1 to 1 odds * Game’s turn variable increases by 1 * Game’s matches variable = the appropriate matches * Turn details are displayed in console | | | |
| **Notes** | When betting with odds, a winner is expected to get back their initial bet plus the winnings.  This test demonstrates the known bug 1, it is expected to fail. | | | |
| **Result (Pass/Fail/Warning/Incomplete)** | Fail, bug exists. | | | |
| **Expected bug output** | Player balance does not increase after winning a bet. | | | |
| **TEST STEP** | | **EXPECTED TEST RESULTS** | P | F |
| Test 1 – Bet won with 1 to 1 odds, balance increases by 5 | | | | |
| 1. Run program with the user picking crown, and the results being crown, heart and anchor. | | * Player’s balance variable increases by Game’s winnings variable (5) * Game’s turn variable increases by 1 * Game’s matches variable = 1 * Turn details are displayed in console | X |  |
| Test 2 – Bet won with 2 to 1 odds, balance increases by 10 | | | | |
| 1. Run program with the user picking crown, and the results being crown, crown and anchor. | | * Player’s balance variable increases by Game’s winnings variable (10) * Game’s turn variable increases by 1 * Game’s matches variable = 2 * Turn details are displayed in console |  | X |
| Test 3 – Bet won with 3 to 1 odds, balance increases by 15 | | | | |
| 1. Run program with the user picking crown, and the results being crown, crown and anchor. | | * Player’s balance variable increases by Game’s winnings variable (15) * Game’s turn variable increases by 1 * Game’s matches variable = 3 * Turn details are displayed in console |  | X |

# Task 2 – Simplification

Junit bug simplification tests in project.

## **Simplification Test Outputs Demonstrating the Bug**



# Task 3 – Tracing

## **Debugging Log**

*Hypothesis 1* – Main’s winnings variable is infected and incorrect

*Prediction 1* – The initializing method (Game’s playRound()) causes winnings to be incorrect

*Note:* If tracing the winnings variable origin and calculation does not reveal the bug, it is in the variables used to call the playRound() method (player, pick or bet).

*Test 1* – Backward observation from Main’s winnings variable

* The winnings variable is created by calling the playRound() method which returns a value, therefore we can hypothesize that the playRound() method’s return value is infected, we will observe this first.
* Tracing backwards through the playRound() method we observe that the return value is a variable called ‘winnings,’ this we hypothesize is infected.
* Secondly we encounter an if statement:

**if** (matches > 0) {

player.receiveWinnings(winnings);

}

For this hypothesis, tracing through to the Player class is out of scope, however if we determine that Main’s winning variable is not infected, we will create another hypothesis that the Player’s recieveWinnings() method causes the defect.

* Thirdly, we observe the initialization of the winnings variable:

**int** winnings = matches \* bet;

We must examine the matches and bet variables.

* By demonstrating that the matches and bet variables are sane in Hypothesis 2 and 4, we should be able to conclude that the immediately above statement is the origin of the defect, however the winnings variable is only the additional winnings, not including the return of the initial bet to the players balance also. We observed before that the recieveWinnings() method might cause the defect, *jump to Hypothesis 5.*

*Hypothesis 2* – Game’s playRound() method matches variable is infected and incorrect

*Prediction 2* – matches is incorrectly determined

*Test 2* – Backward observation from the initialization of the winnings variable

* Tracing backwards we encounter a for statement:

**for** ( Dice d : dice) {

d.roll();

**if** (d.getValue().equals(pick)) {

matches += 1;

}

}

Inside the for statement is an if statement which if true will increase the matches variable by 1.

As a result of Hypothesis 3, we determine that there is no error in the if statement. By simply observing we also determine that the matches +=1; statement can cause no error, therefore Hypothesis 2 is disproven and the matches variable is sane.

*Hypothesis 3* – Dice’s getValue() method controlling the for statement that determines how many matches are found when comparing a dice face with the players pick causes the bug

*Prediction 3* – The getValue() method returns an incorrect dice face which was determined by the above Dice roll() method

*Test 3* – Backward observation from the d.getValue() method controlling the if statement

* Inside the Dice class we find the getValue() method:

**public** DiceValue getValue() {

**return** value;

}

It simply returns the value which is determined by the Dice constructor:

**public** Dice() {

value = DiceValue.*getRandom*();

}

Which in turn initializes the value variable by calling the DiceValue getRandom() method

* Inside the DiceValue class we find the getRandom() method:

**public** **static** DiceValue getRandom() {

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

**return** *values*()[random];

}

By observing this method, we find that it picks a random dice face and returns it to the value variable in the Dice class. Which is retrieved by getValue method in the Dice class called by the if statement we were examining in Hypothesis 2. It is then compared to the pick variable which determined in the Main class by using the DiceValue getRandom() method also. We can conclude that no error lies in the DiceValue getRandom() method, Dice getValue() method and ultimately the if statement being observed in Hypothesis 2.

Hypothesis 4 - Game’s playRound() method bet variable is infected and incorrect

Prediction 4 - bet is incorrect

Test 4 – Trace statement and observation

* By implementing a simple trace statement:

System.***out***.println(bet);

In the console we can see that the bet is always 5:

Turn 28: Fred bet 5 on HEART

5

Rolled HEART, CROWN, ANCHOR

Fred won 5, balance now 50

Turn 29: Fred bet 5 on HEART

5

Rolled HEART, CROWN, ANCHOR

Fred won 5, balance now 50

Turn 30: Fred bet 5 on ANCHOR

5

Rolled HEART, CROWN, ANCHOR

Fred won 5, balance now 50

* We also observe that the bet variable is sent as a parameter to the playRound() method
* Tracing backward to the original statement examined in Hypothesis 1 (when the winnings variable is initialized in the Main class) we see that the bet variable is never manipulated. It is simply initialized to the value of 5 and never changes:

**int** bet = 5;

Therefore we conclude that the bet variable is sane.

Hypothesis 5 – Player’s receiveWinnings() method causes the defect.

Prediction 5 – The winnings variable is incorrectly added to the Player’s balance variable.

Test 5 – Observation

* In observing the recieveWinnings() method we see that an if statement will be true if the winnings variable is greater than zero:

**if** (winnings < 0) **throw** **new** IllegalArgumentException("Winnings cannot be negative.");

balance = balance + winnings;

* If true, the balance variable will increase by the winnings.
* Until now the balance variable was sane, tracing back we see that the takeBet() method is called from the playRound() method:

player.takeBet(bet);

The takeBet() method simply subtracts the bet(always 5, explained in Hypothesis 4) from the balance.

The balance variable is initially created in the Main class and set to 100:

**int** balance = 100;

It is then sent as a parameter to the Player constructer:

player = **new** Player(name, balance);

* However, it is infected by the statement below (the same as in the first bullet point):

balance = balance + winnings;

Here, the additional winnings (determined to be sane by Hypothesis 1) is added to the balance, but this is not a typical return when winning a bet. To explain; if a bet of 5 is played on 1 to 1 odds and is won, the winner will expect a return of their initial bet + winnings (5 + 1 \* 5 = 10), same goes for 2 to 1 odds (5 + 2 \* 5 = 15), and so on. However, the above statement only returns the additional winnings, using the same example on 1 to 1 odds the winner is returned the additional winnings only (1 \* 5 = 5), and the same for 2 to 1 odds (2 \* 5 = 10), and so on. In all cases of the current statement, the player’s balance variable will only ever increase when a bet of 2 to 1 odds and 3 to 1 odds is one, and will remain the same for 1 to 1 odds.

Ultimately, the above statement is the origin of the defect and Hypothesis 5 is proven.



Figure 1 - Origin of Bug 1

# Task 4 – Resolution

## **Automated Test Outputs**

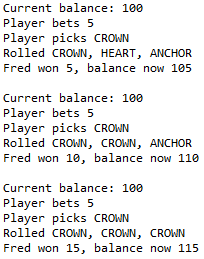


Figure 2 - Same as the Simplification Tests after the bug is fixed

## **Test Case Outputs**

Same logic and inputs were used for simplification tests so results the same.

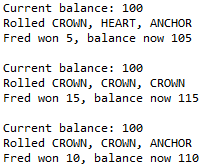


Figure 3 - Bug 1 Test Case Output - Test 1

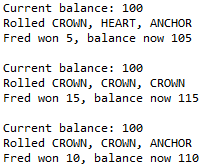


Figure 4 - Bug 1 Test Case Output - Test 2

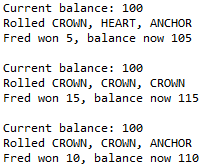


Figure 5 - Bug 1 Test Case Output - Test 3

Bug 2: Player cannot reach betting limit:

Limit set to 0, but game ends with player still with 5 (dollars) remaining.

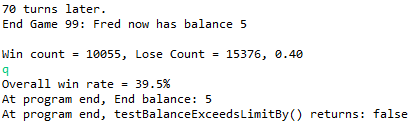
# Task 1 – Replication

| **Test Name** | Betting limit reached Test | | | |
| --- | --- | --- | --- | --- |
| **Test Description:** | Tests the ending of the game when a Player’s balance reaches the 0 limit. | | | |
| **Pre-conditions** | * Main object exists * Dice object exists * Player object exists * Game object exists * DiceValue list exists * One game has completed | | | |
| **Post-conditions** | * One game completes when the Player’s balance variable reaches the Player’s limit variable (0) * Main’s i variable (controlling the amount of games) increases by 1 * All elapsed turn details displayed in the console * End game details displayed with the i variable and limit variable (0) | | | |
| **Notes** | This test demonstrates the known bug 2, it is expected to fail. | | | |
| **Result (Pass/Fail/Warning/Incomplete)** | Fail, bug exists. | | | |
| **Expected bug output** | Program ends when the players balance has 5 remaining. | | | |
| **TEST STEP** | | **EXPECTED TEST RESULTS** | P | F |
| Test 1 – Game completes when balance reaches limit | | | | |
| 1. Run program | | * One game complete (one iteration of the Main for statement controlled by the i variable * Player’s balance variable reaches the limit variable (0) * Main’s i variable (controlling the amount of games) increases by 1 * All elapsed turn details displayed in the console * End game details displayed with the i variable and limit variable (0) |  | X |

# Task 2 – Simplification

Junit bug simplification tests in project.

## **Simplification Test Outputs Demonstrating the Bug**



# Task 3 – Tracing

## **Debugging Log**

*Hypothesis 1* – One or both of the game controlling while statement control methods in the Main class causes the defect

*Prediction 1* – Player’s balanceExceedsLimitBy() or getBalance() method has an infected variable or defect statement

*Note:* We can rule out the bet parameter because from evaluating Bug 1 we determined that the bet variable in the Main class is sane and never changes. The Player object that calls the balanceExceedsLimitBy() method could be infected, if examining the balanceExceedsLimitBy() method shows no results we will examine the Player object next, there may be issues with the constructor or other methods that were called for the Player object previously in the Main class.

*Test 1* – Observation

* We know it isn’t the getBalance() method also in the same while statement in the Main class because we narrowed it down to the balanceExceedsLimitBy() method in the simplification tests.

*Hypothesis 2* – Player’s balanceExceedsLimitBy() method causes the defect

*Prediction 2* – Player’s balanceExceedsLimitBy method has an infected variable or defect statement

*Test 2* – Observation and Junit test

* The method has one statement, a return:

**return** (balance - amount > limit);

We don’t require a new hypothesis to determine if balance is sane because we determined in Hypothesis 5 of Bug 1 that the Player’s balance variable was sane upon fixing a defect in the addition of winnings. However, we must investigate the limit variable.

* Hypothesis 3 determines that the limit variable is sane, therefore the defect must be in the statement above. Upon further examination of the statement we can see that it would return true when the balance minus the amount(bet) is greater than the limit. To provide an example of this logic, I have provided a Junit test class titled Bug2Hypothesis2 inside the TracingAidTest source folder in the project. Following this test class, an example displaying this flawed logic is provided. If balance = 5 and limit = 0, and we apply the balanceExceedsLimitBy() method logic; 5 – 5 > 0 will return false. This will leave the player with a balance of 5 since the method will return false Main’s while loop will break prematurely. Instead the logic should be; balance > limit.

Ultimately, the above statement is the origin of the defect and Hypothesis 2 is proven.

*Hypothesis 3* – Player’s limit variable is infected or incorrect

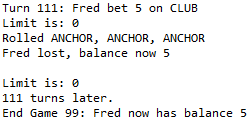
*Prediction 3* – Player’s limit variable is not equal to zero as initialized in the Main class

*Test 3* – Observation and trace statements

* To test this we can use the trace statement:

System.***out***.println("Limit is: " + limit);

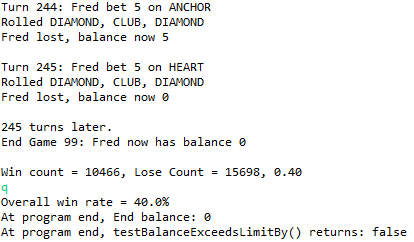
This will print out in the console along with normal outputs the limit every new turn. Here is the output:



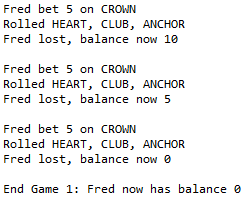
We can see that the limit variable remains at zero between turns, therefore we can rule limit as sane and return to Hypothesis 2.

# Task 4 – Resolution

## **Automated Test Outputs**



## **Test Case Outputs**



Bug 3: Odds in the game do not appear to be correct.

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.

# Task 1 – Replication

| **Test Name** | Win ratio calculation Test | | | |
| --- | --- | --- | --- | --- |
| **Test Description:** | Tests the expectation that the 8% house bias should produce an approximately 0.42 win ratio. | | | |
| **Pre-conditions** | * Main object exists * Dice object exists * Player object exists * Game object exists * DiceValue list exists * 100 games have completed | | | |
| **Post-conditions** | * Win ratio calculated by multiplying the totalWins (turn wins) by 100 and dividing the result by the totalWins + totalLosses * Win ratio displayed in the console | | | |
| **Notes** | This test demonstrates the known bug 3, it is expected to fail. | | | |
| **Result (Pass/Fail/Warning/Incomplete)** | Fail, bug exists. | | | |
| **Expected bug output** | Win ratio has large outliers, far away from approximately 0.42. | | | |
| **TEST STEP** | | **EXPECTED TEST RESULTS** | P | F |
| Test 1 – 100 games completed and approximately 0.42 win ratio calculated | | | | |
| 1. Run program | | * Win ratio of approximately 0.42 calculated by multiplying the totalWins (turn wins) by 100 and dividing the result by the totalWins + totalLosses * Win ratio displayed in the console |  | X |

# Task 2 – Simplification

Junit bug simplification tests in project.

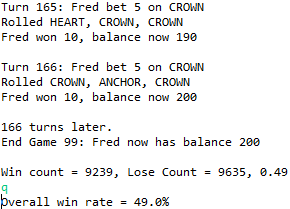


Figure 6 – Tests pass, although we want them to fail, as demonstrated in the following Tracing section the Bug appears to be a lack of bias implementation rather than win rate calculation

# Task 3 – Tracing

## **Debugging Log**

*Hypothesis 1* – totalWins variable is infected or incorrect

*Prediction 1* – totalsWins is incorrectly calculated

*Note:* We must also rule totalLosses as sane

*Test 1* – Observation

* Tracing backwards from the statement which displays the win rate:

System.***out***.println(String.*format*("Overall win rate = %.1f%%", (**float**)(totalWins \* 100) / (totalWins + totalLosses)));

We encounter totalsWins declaration statement:

totalWins += winCount;

Further observation reveals that the totalWins variable is not manipulated at all since its initialization at the top of Main’s main() method:

**int** totalWins = 0;

However, we must investigate the winCount variable to determine that totalWins is sane.

*Hypothesis 2* – winCount variable is infected or incorrect

*Prediction 2* – winCount is incorrectly calculated

*Test 2* – Observation

* We can start from the same statement observed in Hypothesis 1:

totalWins += winCount;

* Tracing back we encounter a statement with the winCount variable:

System.***out***.println(String.*format*("Win count = %d, Lose Count = %d, %.2f", winCount, loseCount, (**float**) winCount/(winCount+loseCount)));

The variable is not changed in this statement so we can rule it out as the origin of the defect

* Next we see winCount in this statement:

**if** (winnings > 0) {

System.***out***.printf("%s won %d, balance now %d\n\n",

player.getName(), winnings, player.getBalance());

winCount++;

}

The if statement, controlled by the winnings variable couldn’t cause since from Bug 1 Hypothesis 1 we know that the winnings variable is created by the playRound() method and was deemed sane.

* No other statement uses the winCount variable until its initialization:

**int** winCount = 0;

*Hypothesis 3* – totalLosses variable is infected or incorrect

*Prediction 3* – totalLosses is incorrectly calculated

*Test 3* – Observation

* totalLosses follows almost exactly the same cycle as totalWins, here it is declared by winCount opposite loseCount:

totalLosses += loseCount;

* And it is not manipulated since its initialization:

**int** totalLosses = 0;

Again, we must investigate the declaring variable (loseCount) to determine that totalLosses is sane.

*Hypothesis 4* – totalLosses variable is infected or incorrect

*Prediction 4* – totalLosses is incorrectly calculated

*Test 4* – Observation

* We can start from the same statement observed in Hypothesis 3:

totalLosses += loseCount;

* Tracing backwards we encounter:

**if** (winnings > 0) {

System.***out***.printf("%s won %d, balance now %d\n\n",

player.getName(), winnings, player.getBalance());

winCount++;

}

**else** {

System.***out***.printf("%s lost, balance now %d\n\n",

player.getName(), player.getBalance());

loseCount++;

}

But again, as Hypothesis 2 states, the winnings variable controlling the if else statement cannot be the cause of the defect since it is sane.

Note: To sum up Hypothesis’ 1-4, the totalLosses, loseCount, totalWins and winCount variables are all sane. There it must be determined that there has not been a bias placed upon the game and that it is totally random, to resolve this error bias must be implemented.

# Task 4 – Resolution

## **Automated Test Outputs**

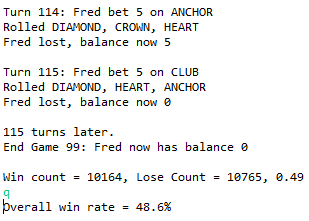


Figure 7 - Tests also assertTrue for winCount or lossCount increasing by 1 for both cases where 1 game is won and 1 game is lost

## **Test Case Outputs**

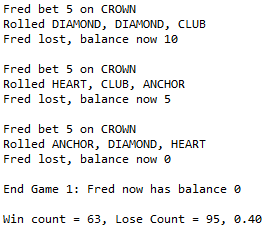


Figure 8 - Test also assertTrue that the win rate is not higher than 0.50 or lower than 0.35

Bug 4: The dice rolls never change throughout the 100 games and all turns

Each turn a new set of three dice should be generated, otherwise in a real life scenario if the same dice were thrown each turn a player could just make the same pick and infinitely win.

# Task 1 – Replication

| **Test Name** | Dice roll Test | | | |
| --- | --- | --- | --- | --- |
| **Test Description:** | Tests that a new set of three dice are generated each turn. | | | |
| **Pre-conditions** | * Main object exists * Dice object exists * Player object exists * Game object exists * DiceValue list exists * 2 games have completed. | | | |
| **Post-conditions** | * DiceValue cdv List elements are equal to a truly random new set of three dice * cdv DiceValues are displayed in the console | | | |
| **Notes** | This test demonstrates the known bug 4, it is expected to fail. | | | |
| **Result (Pass/Fail/Warning/Incomplete)** | Fail, bug exists. | | | |
| **Expected bug output** | All dice throughout the 2 games and all turns are the same. | | | |
| **TEST STEP** | | **EXPECTED TEST RESULTS** | P | F |
| Test 1 – 2 games completed and dice rolls truly random | | | | |
| 1. Run program | | * DiceValue cdv List elements are equal to a truly random new set of three dice * cdv DiceValues are displayed in the console |  | X |

# Task 2 – Simplification

Junit bug simplification tests in project.

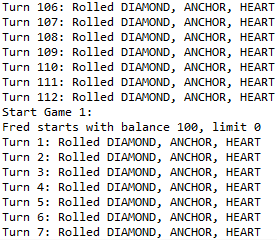


Figure 9 - Samples of outputs of 2 games, see that all the dice rolls are the same for both games and all sample turns

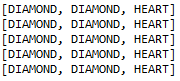


Figure 10 - Result of calling the playRound() method 5 times with the same game and printing the dice values

# Task 3 – Tracing

## **Debugging Log**

*Hypothesis 1* – The playRound() method is infected

*Prediction 1* – Dice values do not change when they are rolled

*Note:* In the simplification test testPlayRoundMethod() it is determined that the following statement is infected:

game.playRound(player, pick, bet);

*Test 1* – Observation and trace statements

* Tracing backwards through the playRound() method we don’t encounter any Dice object until the following for loop:

**for** ( Dice d : dice) {

d.roll();

**if** (d.getValue().equals(pick)) {

matches += 1;

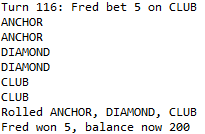
}

}

By placing the following trace statement we can see that the Dice ArrayList (dice) contains the correct Dice:

System.***out***.println(d.getValue());

Ouput:



* The next statement is:

d.roll();

Currently, this statement follows the current prediction and appears to be infected, to Hypothesis 2.

*Hypothesis 2* – Dice class’ roll() method is infected

*Prediction 2* – Dice values do not change when they are rolled

*Test 2* – Observation

* The roll() method does nothing but return the call return of the DiceValue getRandom() method:

**public** DiceValue roll() {

**return** DiceValue.*getRandom*();

}

Here is the origin of the bug, when the getRandom() method is called, it simply returns a random dice face. However, the value variable in the Dice class does not change to the same as the return, which it should to ensure that when a Dice object calls the roll() method, the value actually changes.

To fix the bug we add a new statement that stores the value variable with the return of the getRandom() method and change the return of the roll() method. It now looks like this:

**public** DiceValue roll() {

value = DiceValue.*getRandom*();

**return** value;

}

# Task 4 – Resolution

## **Automated Test Outputs**

## 

## **Test Case Outputs**

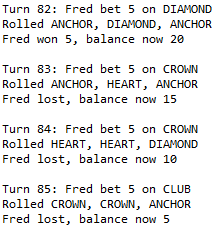


Figure 11 - This test simply runs the Main's main() method since the UAT only requires the running of the program, it is visible that there are truly random dice rolls between turns