ITC205 Assignment 4

# Github Repository URL

<https://github.com/Piemaster2911/ITC205_Assignment4>

# Project Structure

* Original Project
  + The unchanged, bugged project to be used as comparison to the debugged, fixed project
* Debugged Project
  + The debugged project, with fixes implemented from bug tests
* bugTesting
  + Contains subfolders for each test (where applicable), which contains the test unit classes that performs tests.
* The ITC205 Assignment 4 Document

# Bug Report 1

## Description

The bug report describes that the game does not pay out at correct level. “When player wins on 1 match, balance does not increase.”

## Pre-Test Run

Running the Main.java class shows on the very first turn where the player “Fred” plays his turn.

Start Game 0:

Fred starts with balance 100, limit 0

Turn 1: Fred bet 5 on ANCHOR

Rolled ANCHOR, ANCHOR, CLUB

Fred won 10, balance now 105

Here, Fred got two symbols matching, and because it matches the symbol he betted on, he should have won 5, on top of the additional 5 returned after betting 5 into the bet. Here, the problem isn’t clear at first – he won 10, but the balance is incremented by only 5. As it turns out, the balance must first pay the amount paid as bet, which first removes 5 from balance, and if the player won with two matching symbols like in this case, 5 is awarded back as a win for having one matching symbol, then another 5 is awarded for having a second matching symbol.

Repeating the pre-test run of the main class, every win by Fred appears to have not award the correct amount, with a win in one symbol matching appearing to state that money has been awarded, when it merely kept the balance at a still as it only instead refunded the amount betted after he matched only one symbol.

Turn 3: Fred bet 5 on CLUB

Rolled CLUB, HEART, CLUB

Fred won 10, balance now 105

Turn 4: Fred bet 5 on HEART

Rolled CLUB, HEART, CLUB

Fred won 5, balance now 105

Winning with only one matching symbol awards none when 5 should be awarded. Similarly, only 5 is awarded for two matching symbols of the symbol betted.

Not winning did function normally, however,

Turn 6: Fred bet 5 on CROWN

Rolled CLUB, HEART, CLUB

Fred lost, balance now 100

Here, not winning will decrement winnings by the bet amount.

## Pre-Analysis Hypothesis

Few possible hypothesis can be made from this bug. It could be interpreted that the bet amount does remove the amount betted, but that the value betted is not added to the printed output to reflect the change.

## Analysis

Based on the above, a further analysis is done. Looking at the Main.java class, the call to the function that does the winnings process is found as highlighted in yellow below.

**while** (player.balanceExceedsLimitBy(bet) && player.getBalance() < 200)

{

turn++;

DiceValue pick = DiceValue.*getRandom*();

System.***out***.printf("Turn %d: %s bet %d on %s\n",

turn, player.getName(), bet, pick);

**int** winnings = game.playRound(player, pick, bet);

cdv = game.getDiceValues();

System.***out***.printf("Rolled %s, %s, %s\n",

cdv.get(0), cdv.get(1), cdv.get(2));

**if** (winnings > 0) {

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

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

winCount++;

}

**else** {

…

}

} //while

An attempt at creating a test of the procedure narrows down the processes involved in the random selection of dies that are used in checking whether anything is won, where we discover the variable responsible for the printing out of winnings prize size.

**public** **int** playRound(Player player, DiceValue pick, **int** bet ) {

// Exceptions goes here

player.takeBet(bet);

**int** matches = 0;

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

d.roll();

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

matches += 1;

}

}

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

**if** (matches > 0) {

player.receiveWinnings(winnings);

}

**return** winnings;

}

Here, we can see that the amount of winnings made is returned to be outputted in such a way that it generates the console output “Fred won 10, balance now 105” when it should be “Fred won 5, balance now 105”.

## Post-Analysis Hypothesis

From the code, we can determine that the winnings returned to be outputted did not reflect the contribution that paying the bet would have on the player’s balance, and as such is giving out incorrect winnings output.

## Test

A test is created to replicate the function playRound(player, pick, bet) in the Game class, and the above hypothesis is tested.

After running the test, with the appropriate console output given, the output starts to look clear. The bet paid is not added as a factor into determining the value of the winnings variable, thus causing the confusion that more is paid than it looked to be.

Player preparing to place bet! Balance: 100

Player places bet! Balance: 95

Player picks DIAMOND, dice result values: DIAMOND, HEART, DIAMOND

Winnings: 10, balance: 105

Player preparing to place bet! Balance: 105

Player places bet! Balance: 100

Player picks ANCHOR, dice result values: CLUB, DIAMOND, CROWN

Winnings: 0, balance: 100

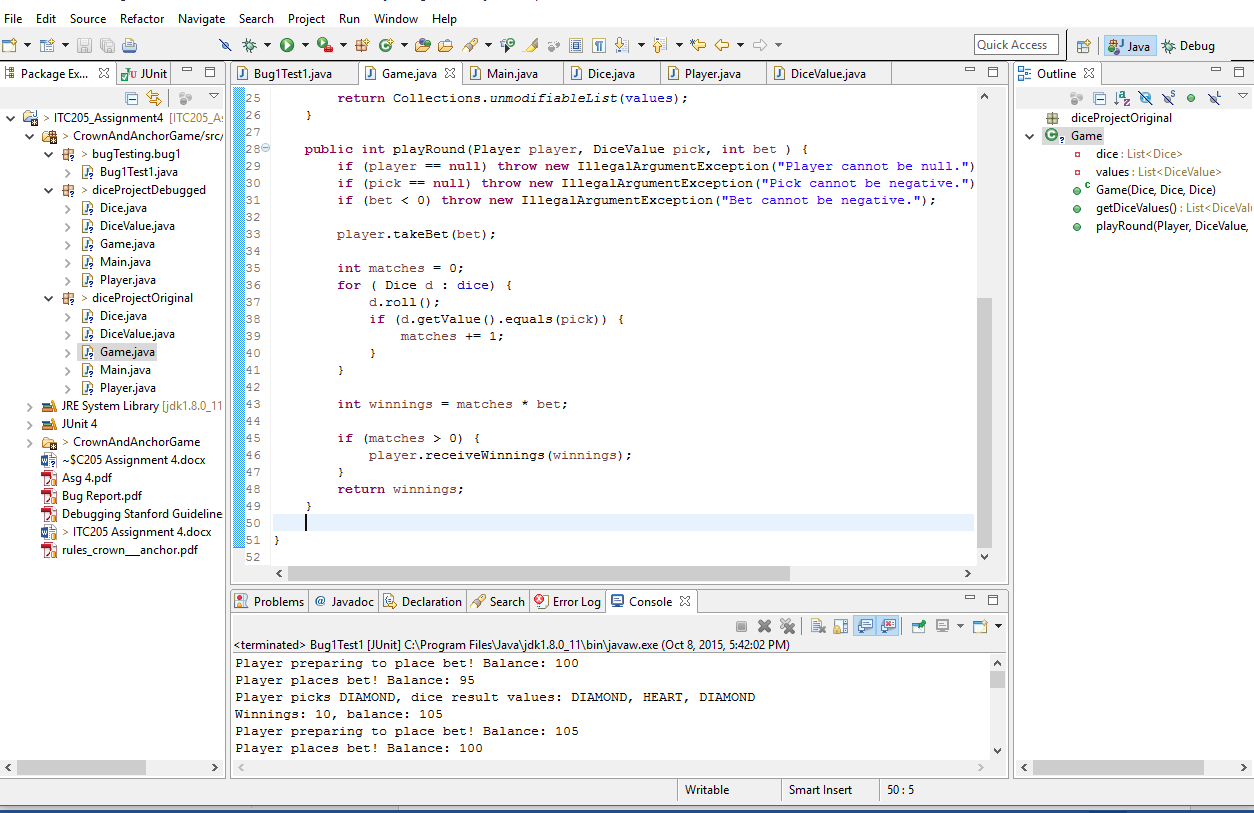
The above excerpt from this instance of the test run supports the hypothesis.

## Solution Attempt

This has one very simple solution – the winnings value needs to be modified by the bet value that is paid from the balance in order to run the betting process. This involves a rather simple one-line addition in the Game class, under the playRound function.

The solution is first tested as a solution test, to see if it does function

Before (the original project’s Game class is shown here instead of the test):



A line of code is added to the test:

winnings -= bet;

The test output is as given

Player preparing to place bet! Balance: 90

Player places bet! Balance: 85

Player picks CLUB, dice result values: CROWN, HEART, DIAMOND

Winnings: 0, balance: 85

Player preparing to place bet! Balance: 85

Player places bet! Balance: 80

Player picks CLUB, dice result values: CLUB, DIAMOND, CLUB

Winnings: 10, balance: 90

This line helps make a less confusing result reading for the user.

The changes made to the Game class under the modified or debugged branch of the project results in output like this.

Turn 6: Fred bet 5 on CLUB

Rolled CROWN, ANCHOR, CROWN

Fred lost, balance now 95

Turn 7: Fred bet 5 on CROWN

Rolled CROWN, ANCHOR, CROWN

Fred won 5, balance now 100

The winnings are now reflected properly.

# Bug Report 2

## Description

The bug report describes that the player does not appear to be able to reach the betting limit of 0, where instead, the game would instead end with the player still having 5 dollars remaining.

## Pre-Test Run

The output supports that this bug has occurred.

Turn 51: Fred bet 5 on CLUB

Rolled CROWN, ANCHOR, CROWN

Fred lost, balance now 10

Turn 52: Fred bet 5 on CLUB

Rolled CROWN, ANCHOR, CROWN

Fred lost, balance now 5

52 turns later.

End Game 0: Fred now has balance 5

Here, the game ends when the player only has 5 remaining, which is not typical.

## Analysis

Analysis indicate that the limit’s value is indeed set to 0, and that the limit is set in the following highlighted line from the Main class.

**for** (**int** i = 0; i < 100; i++)

{

String name = "Fred";

**int** balance = 100;

**int** limit = 0;

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

player.setLimit(limit);

**int** bet = 5;

…

**while** (player.balanceExceedsLimitBy(bet) && player.getBalance() < 200)

{ …

This leads to the Player class, where it appears the function does not alter the limit value.

**public** **void** setLimit(**int** limit) {

**if** (limit < 0) **throw** **new** IllegalArgumentException("Limit cannot be negative.");

**if** (limit > balance) **throw** **new** IllegalArgumentException("Limit cannot be greater than balance.");

**this**.limit = limit;

}

Another portion of the code in Main class is investigated for any possible clue as to why the test end prematurely with the player having some cash remaining above the betting limit given.

**while** (player.balanceExceedsLimitBy(bet) && player.getBalance() < 200)

This one while loop condition lead to another function under the Player class.

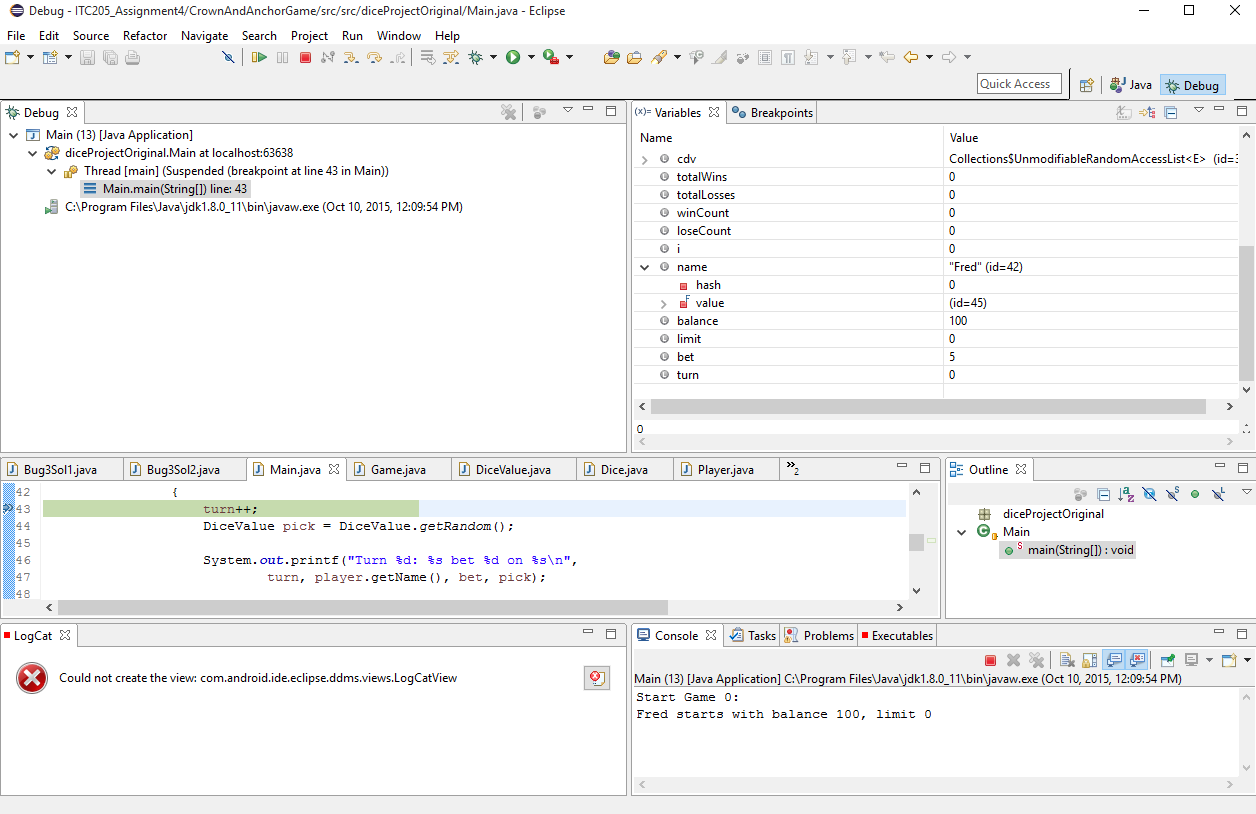
**public** **boolean** balanceExceedsLimitBy(**int** amount) {

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

}

According to this function, the amount value inputted to the function is the bet value, which is 5. This is negated from the balance amount of the player, and is checked if the value is greater than the limit provided. Because of this, when the player has only 5 or less remaining, the game will terminate when the player has only 5 or less remaining, instead of at 0.

This is supported by the debug done on the program code, as shown in screenshots below.



Here, balance is set at 100, bet at 5, and limit at 0.

Although not easy to capture in debug mode, the function balanceExceedsLimitBy does play a role, as its input must have been a cause of the premature game over.

## Test

The first test is produced to check and validate the function balanceExceedsLimitBy by checking the value produced from the (balance – amount) portion against limit value to check if the sum of the left hand portion is greater than the limit provided the amount value provided in a hypothetical case where the player has only 5 remaining.

Balance: 5, amount: 5, limit: 0

Now checking boolean state if balance is above limit

Balance is above limit

Now performing balance - amount...

Balance: 0

Now checking boolean state if balance is above limit

Balance is not above limit

Test ended...

When the amount is set to 0 however, this will result.

Balance: 5, amount: 0, limit: 0

Now checking boolean state if balance is above limit

Balance is above limit

Now performing balance - amount...

Balance: 5

Now checking boolean state if balance is above limit

Balance is above limit

Test ended...

It seems that the cause of the problem might have been found – the amount value is responsible for the bug.

## Solution Attempt

A solution is produced based on the test. The solution made is by omitting the amount, as it makes little sense to take amount as input if it is causing this erroneous bug that confuses the player or user.

The following output is produced after the change is made.

Turn 75: Fred bet 5 on DIAMOND

Rolled DIAMOND, HEART, HEART

Fred lost, balance now 5

Turn 76: Fred bet 5 on CLUB

Rolled DIAMOND, HEART, HEART

Fred lost, balance now 0

76 turns later.

End Game 99: Fred now has balance 0

The output generated clearly shows the program is now working as intended. The bug can be considered fixed by this point.

# Bug Report 3

## Description

There is a severe bug with the program in that the dice roll does not reroll for each and every game, instead keeping the same roll throughout the entire game.

## Pre-Test Run

As noticed from all previous test runs, the die rolls remained persistent. This is obvious from the text output, as indicated by the extract below.

Start Game 99:

Fred starts with balance 100, limit 0

Turn 1: Fred bet 5 on HEART

Rolled DIAMOND, DIAMOND, CLUB

Fred lost, balance now 95

Turn 2: Fred bet 5 on ANCHOR

Rolled DIAMOND, DIAMOND, CLUB

Fred lost, balance now 90

…

Turn 63: Fred bet 5 on HEART

Rolled DIAMOND, DIAMOND, CLUB

Fred lost, balance now 5

It even persists between games.

Start Game 0:

Fred starts with balance 100, limit 0

Turn 1: Fred bet 5 on CLUB

Rolled DIAMOND, DIAMOND, CLUB

Fred lost, balance now 100

…

Start Game 41:

Fred starts with balance 100, limit 0

Turn 1: Fred bet 5 on HEART

Rolled DIAMOND, DIAMOND, CLUB

Fred lost, balance now 95

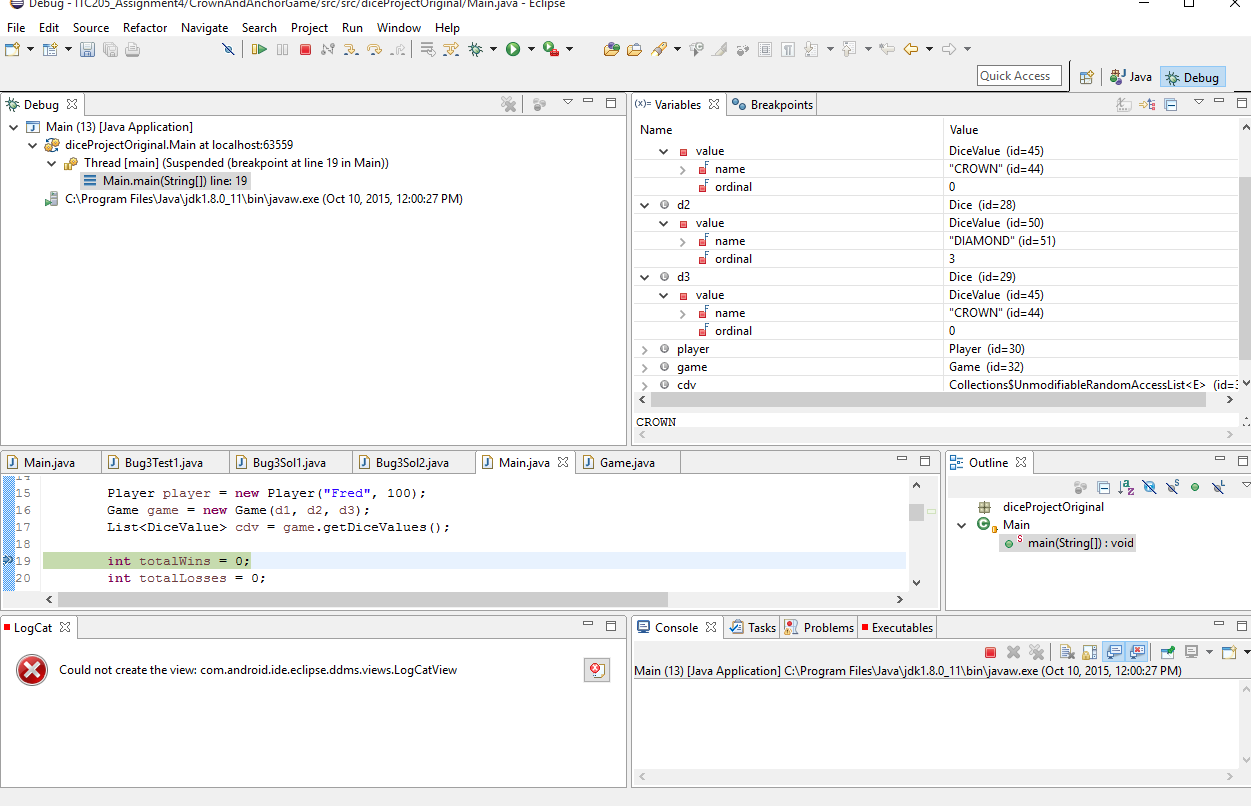
…

This is repeatable throughout every instance of the test.

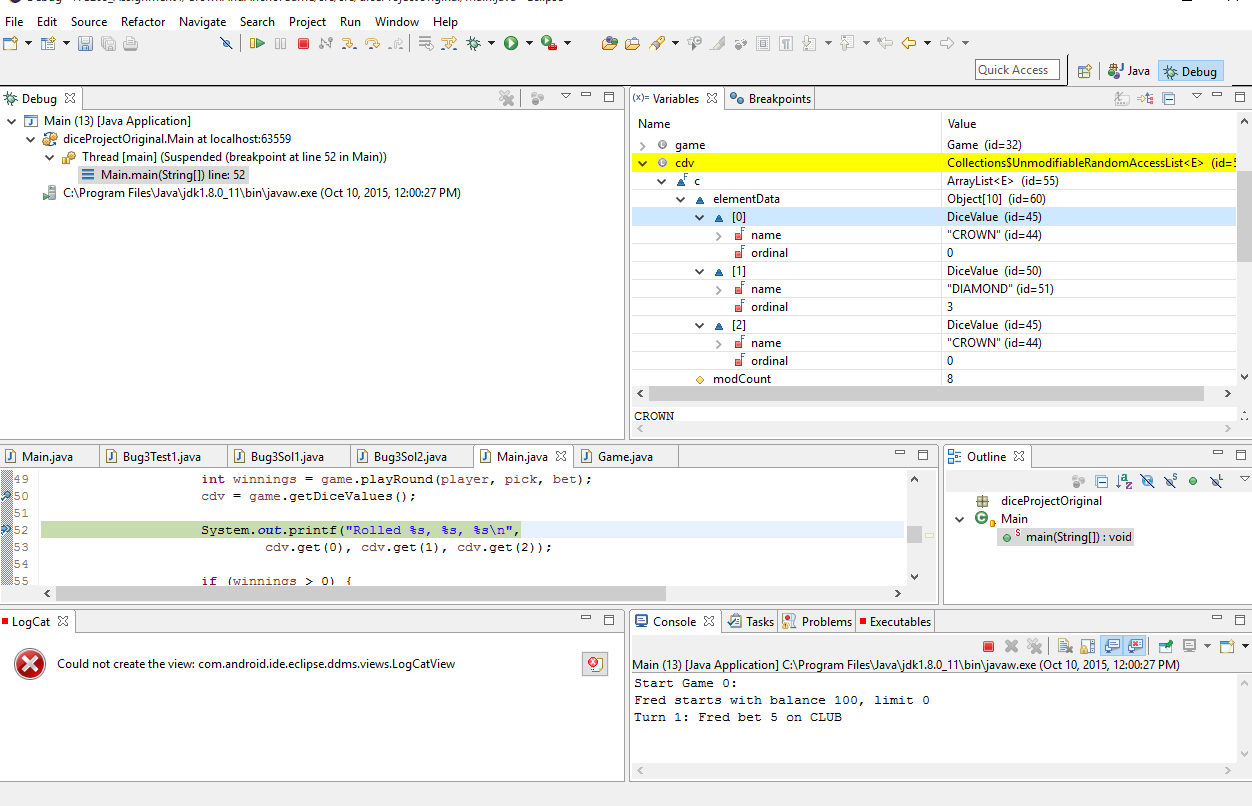
The source of this might be due to the way the dice objects are created and rolled. From the Main class, it does not appear that a call to randomize or roll each die object has been called at all.

## Tracing

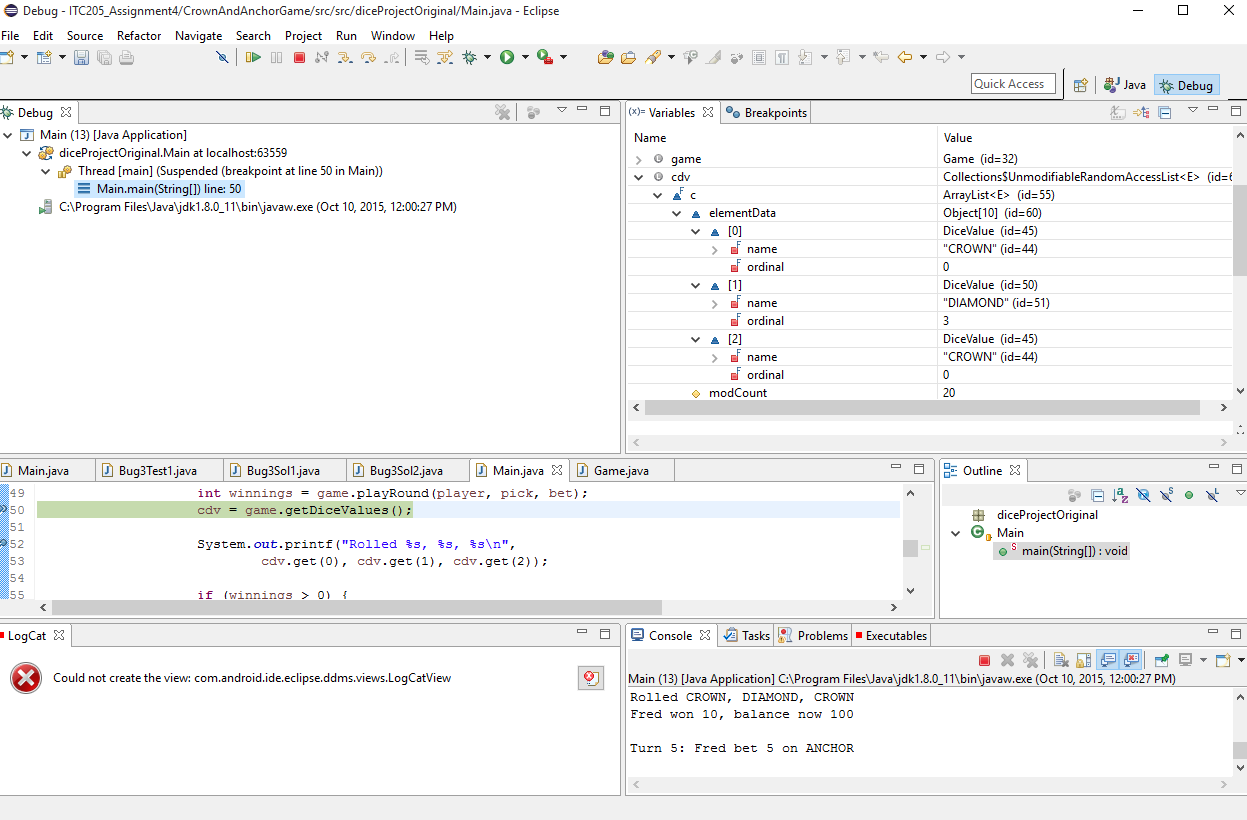
From the above, it appears that the dice rolls are not rerolled after it has been first initialized.



On this screenshot, the variables are initialized, specifically the dice rolls that are then inserted into the game.



When the game starts, it can be seen that the dice rolls CROWN, DIAMOND, CROWN is kept.



However, it gets erroneous when it starts to become clear that the dice rolls are not being rerolled.

## Test

We now test by first replicating the roll, and the output in a loop to see if it does indeed randomize the dice rolls.

Running Bug3Test1, using the two lines of code that does indeed print out and receive what values each die has, the results are as given.

Test Starting...

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Rolled HEART, DIAMOND, DIAMOND

Test Ending...

From this, it seems that the dice are not rolled inside the loop, but before it.

## Solution Attempt

First, a solution test is created before we do anything to the debugged branch of the project. We try to find a way to regenerate each roll inside the loop instead of outside it.

In this solution unit test, we reroll the dice values by re-initializing the game object. This generates the following results from the Bug3Sol1 test class.

Test Starting...

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Rolled CROWN, ANCHOR, CROWN

Test Ending...

It appears that re-initializing the game object alone is not enough to reroll the dice within the loop, so we also re-initialize each and every dice. This generates the following results as extracted from the Bug3Sol2 test class.

Test Starting...

Rolled ANCHOR, CROWN, DIAMOND

Rolled DIAMOND, CLUB, CROWN

Rolled HEART, DIAMOND, HEART

Rolled ANCHOR, DIAMOND, DIAMOND

Rolled ANCHOR, ANCHOR, HEART

Rolled DIAMOND, HEART, HEART

Rolled CROWN, CLUB, CLUB

Rolled HEART, CROWN, CROWN

Rolled DIAMOND, HEART, CLUB

Rolled HEART, DIAMOND, CLUB

Test Ending...

From the above, it does appear that the dice rolls are performed properly within the class.

This is then implemented into the debugged project (with other bug fixed implemented in it prior to this bug report), and the results are as follows from the output console.

Start Game 0:

Fred starts with balance 100, limit 0

Turn 1: Fred bet 5 on ANCHOR

Rolled DIAMOND, CLUB, CLUB

Fred lost, balance now 95

Turn 2: Fred bet 5 on CROWN

Rolled CROWN, CROWN, ANCHOR

Fred won 5, balance now 100

Turn 3: Fred bet 5 on ANCHOR

Rolled ANCHOR, DIAMOND, CLUB

Fred lost, balance now 100

…

Turn 55: Fred bet 5 on ANCHOR

Rolled CLUB, CLUB, CLUB

Fred lost, balance now 5

Turn 56: Fred bet 5 on HEART

Rolled CLUB, DIAMOND, CLUB

Fred lost, balance now 0

56 turns later.

End Game 0: Fred now has balance 0

The dice rolls are now performing properly and randomizes at each and every turn. This also had the side effect of reducing the win odds ratio average from 0.2 to 0.1. The bug can be considered fixed as it is incredibly rare for the dice rolls to produce the same roll twice or more in a roll, instead of the roll always consistently showing the same dice roll value as before, or persisting after to the end of the program’s 100 games.

# Bug Report 4

## Description

The bug describes that the odds given 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 (be) approximately equal 0.42. This does not appear to be the case.”

## Pre-Test Run

From the above, it appears that it might be possible that the functions or maths involved in calculating the winning odds may not be entirely correct. But first of all, a run of the program must be made to check what result it produce for that problem, so as to verify that the bug exist.

65 turns later.

End Game 99: Fred now has balance 0

Win count = 548, Lose Count = 4518, 0.11

However, it does look like the maths is right. The code below suggests as such.

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

totalWins += winCount;

totalLosses += loseCount;

String ans = console.readLine();

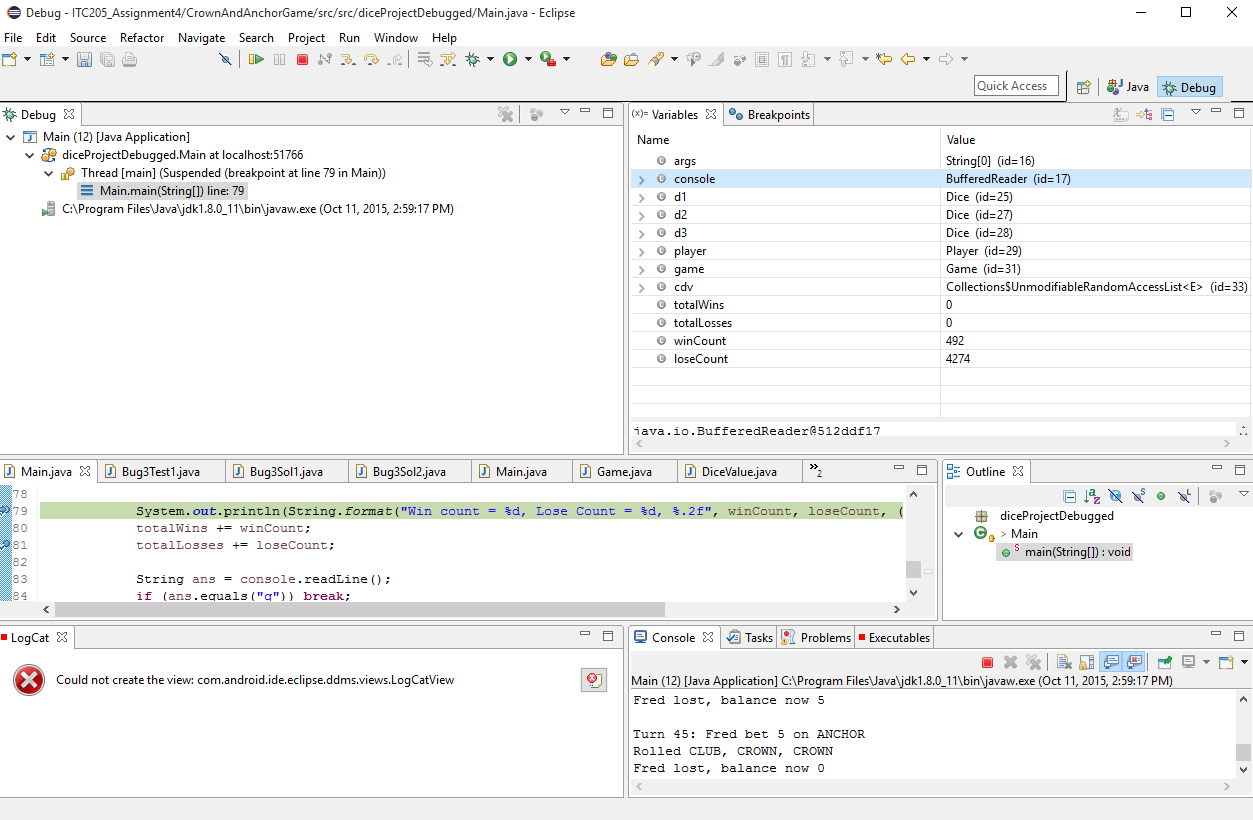
**if** (ans.equals("q")) **break**;

} //while true

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

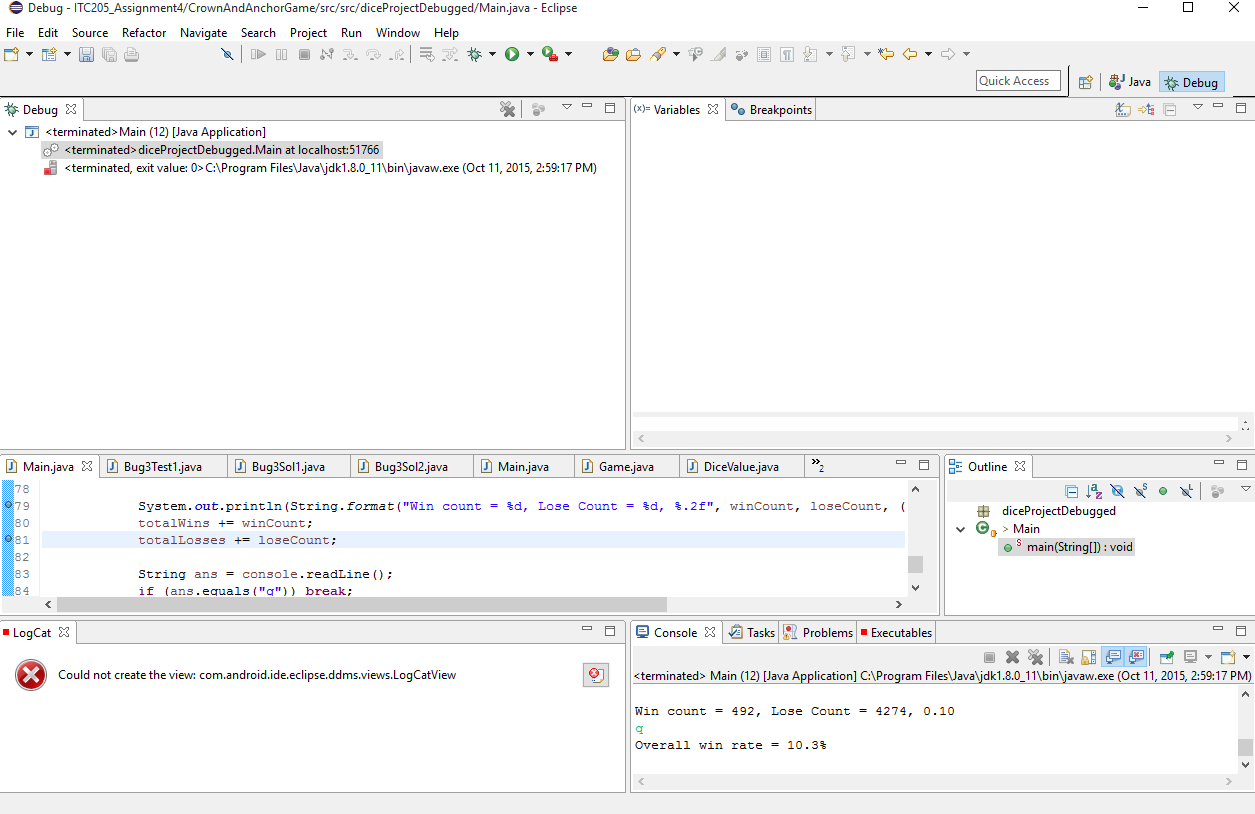
}

Now we debug this portion to see if there might be some hidden issue in the code. The screenshot below shows the variables.



Mathematically, 492 divided by 4274 will equal to the ratio, which here equals 0.103, or 10.3%.

The next screenshot shows the result.



This shows that the code of line producing the ratio output is not incorrect.

## Test

Although the mathematics behind the ratio percentage calculation is correct, there might be other causes for this ratio result (assuming that the ratio is supposed to equal to 0.47 or around there).

The way the winnings are calculated is one suspect. From the code below, it is clear that winnings are only counted if the winnings total is above 0.

**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++;

}

However, getting even from a single dice match to the picked symbol is considered a win, as the bet amount is returned for matching one of three dice rolls. This meant that winnings should be greater or equal to, instead of greater than.

To test this, we duplicated the entirety of Main as a test class, and checked its result.

By changing the condition for winnings to be counted, we drastically altered the ratio, as shown in the result below.

Turn 37: Fred bet 5 on ANCHOR

Rolled CLUB, CLUB, CROWN

Fred lost, balance now 0

37 turns later.

End Game 99: Fred now has balance 0

Win count = 2513, Lose Count = 2554, 0.50

q

Overall win rate = 49.6%

By making this one minor change, we are able to correct the odds probability by making sure it accurately count wins, instead of ignoring one-dice symbol matches winnings.

## Solution Attempt

We now implement this one condition change to the debugged project, and the following result is as follows from the extract from the console output of the project program.

79 turns later.

End Game 99: Fred now has balance 0

Win count = 2336, Lose Count = 2568, 0.48

The odds are more accurately described now that even winnings are counted as wins, as opposed to losses.

# Bug Report 5

## Description

There appears to be several odd code line issues – the problem in particular is the duplicate initialization of the Player in the Main class.

Player player = **new** Player("Fred", 100);

Game game = **new** Game(d1, d2, d3);

List<DiceValue> cdv = game.getDiceValues();

…

String name = "Fred";

**int** balance = 100;

**int** limit = 0;

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

player.setLimit(limit);

**int** bet = 5;

It appears that this duplication might be unintended. However, we must figure a solution to this, as this will invariably cause problems if a player other than “Fred” were to play the game.

As the player is reinitialized each game, rather than persist for every game, it has to be initialized and reinitialized inside the loop, as opposed to outside the loop. This is because otherwise it will end the other games prematurely at the very beginning of each game turn after the first game, giving a result that may be deceiving.

## Test

We tested by first performing a test where it is initialized outside the loop, which is done in the Bug5Test1 class. The result is as follows:

Start Game 0:

Fred starts with balance 100, limit 0

Turn 1: Fred bet 5 on ANCHOR

Rolled HEART, CLUB, ANCHOR

Fred won 0, balance now 100

…

Turn 66: Fred bet 5 on HEART

Rolled ANCHOR, CROWN, CLUB

Fred lost, balance now 0

66 turns later.

End Game 0: Fred now has balance 0

Start Game 1:

Fred starts with balance 0, limit 0

0 turns later.

End Game 1: Fred now has balance 0

…

Start Game 97:

Fred starts with balance 0, limit 0

0 turns later.

End Game 97: Fred now has balance 0

Start Game 98:

Fred starts with balance 0, limit 0

0 turns later.

End Game 98: Fred now has balance 0

Start Game 99:

Fred starts with balance 0, limit 0

0 turns later.

End Game 99: Fred now has balance 0

Win count = 37, Lose Count = 29, 0.56

q

Overall win rate = 56.1%

This indicates that initializing the player class outside the loop will cause erroneous problems such as every game after game 1 starting with the player with no balance to bet, and as such, skips the game, over and over until the program finishes. This also produces less reliable win ratios.

The next attempt is done by placing the player initialization inside of the loop. The following result is as follows, from the Bug5Test2 class.

Turn 43: Fred bet 5 on ANCHOR

Rolled DIAMOND, CLUB, CLUB

Fred lost, balance now 0

43 turns later.

End Game 99: Fred now has balance 0

Win count = 2537, Lose Count = 2603, 0.49

q

Overall win rate = 49.4%

Here, it is safe to say that initializing the player inside the loop will produce a result that is as intended.

## Solution Attempt

One solution that is attempted is by removing the initialization from the start player outside the loop, and retaining the initialization of the player inside the loop instead.

Player player;

Game game = **new** Game(d1, d2, d3);

List<DiceValue> cdv = game.getDiceValues();

…

String name = "Fred";

**int** balance = 100;

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

**int** limit = 0;

player.setLimit(limit);

**int** bet = 5;

This appear to address the issue regarding the duplicate initialization. The console output is also preserved with no erroneous results.

46 turns later.

End Game 99: Fred now has balance 0

Win count = 2358, Lose Count = 2529, 0.48

q

Overall win rate = 48.3%

# Bug Report 6

## Description

There are two boolean functions in Player, called balanceExceedsLimit() and balanceExceedsLimitBy() respectively, which both contained identical returns.

## Pre-Test Run

The program works as intended even with the duplicate functions, but it is not wise to have duplicates of the same function, not to mention the Main class using the less correctly worded function balanceExceedsLimitBy().

## Analysis

So far, the main class has references to the duplicate functions.

**int** turn = 0;

**while** (player.balanceExceedsLimitBy() && player.getBalance() < 200)

{

Another reference exists within the Player class, which also contains the two duplicate functions.

**public** **void** takeBet(**int** bet) {

**if** (bet < 0) **throw** **new** IllegalArgumentException("Bet cannot be negative.");

**if** (!balanceExceedsLimitBy(bet)) **throw** **new** IllegalArgumentException("Placing bet would go below limit.");

balance = balance - bet;

}

It is not possible to find any other reference until we perform the removal of the duplicate.

## Solution Attempt

The solution attempt involves removing the balanceExceedsLimitBy function, as it doesn’t make much sense as it returns a boolean, not an integer or float.

The result gives no anomalous results.

Turn 59: Fred bet 5 on HEART

Rolled DIAMOND, CLUB, CROWN

Fred lost, balance now 0

59 turns later.

End Game 99: Fred now has balance 0

Win count = 2433, Lose Count = 2520, 0.49

q

Overall win rate = 49.1%

This is an indication that the removal of the duplicate function did not severely alter the program’s functions, and runs as intended.

# Bug Report 7

## Description

Test did not terminate after returning win ratio, and the win rate is never displayed, because it requires input of q to console.

## Pre-Test Run

In the Main class, there is a line near the end that is as follows.

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

totalWins += winCount;

totalLosses += loseCount;

String ans = console.readLine();

**if** (ans.equals("q")) **break**;

} //while true

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

From this, it appears that it requests a new line before quitting. Although it might be intentional, it does not allow the program to run automated without any user input involved in running the program.

The system output generated from this is as follows:

39 turns later.

End Game 99: Fred now has balance 0

Win count = 2475, Lose Count = 2550, 0.49

Until we type q and press enter, we will not see the program terminate with the final line. Instead, the program continues to run at this point indefinitely if it were not for user intervention.

This is the output given after q is entered and the enter key is pressed.

Win count = 2475, Lose Count = 2550, 0.49

q

Overall win rate = 49.3%

The program now terminates, and displays the overall win rate in percentage.

## Test

We produce a unit test to demonstrate this, as shown in class Bug7Test1.

The string output result is similar to the pre-test run above, and demonstrates that unless a key input is entered, and entered into the program, it will not stop running the program and will remain idle. The below console output demonstrates a successful termination of the program.

Win count = 4, Lose Count = 6, 0.40

q

Overall win rate = 40.0%

Program successfully shut down!

## Analysis

It is clear that we need to remove this prompt so that it can terminate properly without user input. There is a solution to this – removing the console input variable and any reference to it. This way, the program can automate its task.

## Solution Attempt

The solution attempt is as follows:

@Test

**public** **void** test() **throws** IOException {

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

totalWins += winCount;

totalLosses += loseCount;

**boolean** isValid = **false**;

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

isValid = **true**;

**if**(isValid)

System.***out***.println("Program successfully shut down!");

}

This generates the following output.

Win count = 4, Lose Count = 6, 0.40

Overall win rate = 40.0%

Program successfully shut down!

The program now shuts down without user intervention, fixing this odd choice of having input required to end the program where the program’s role is to basically just automate the process for 100 times.

The fix is adapted into the debugged project branch, with the removal of the console input being the main priority.

The results are then as follows from the console output.

45 turns later.

End Game 99: Fred now has balance 0

Win count = 2338, Lose Count = 2535, 0.48

Overall win rate = 48.0%

The program now terminates properly and all results displayed.