Name: Angela Burge

Student Number: 11540173

Subject Code: ITC205

Assessment Number: 4

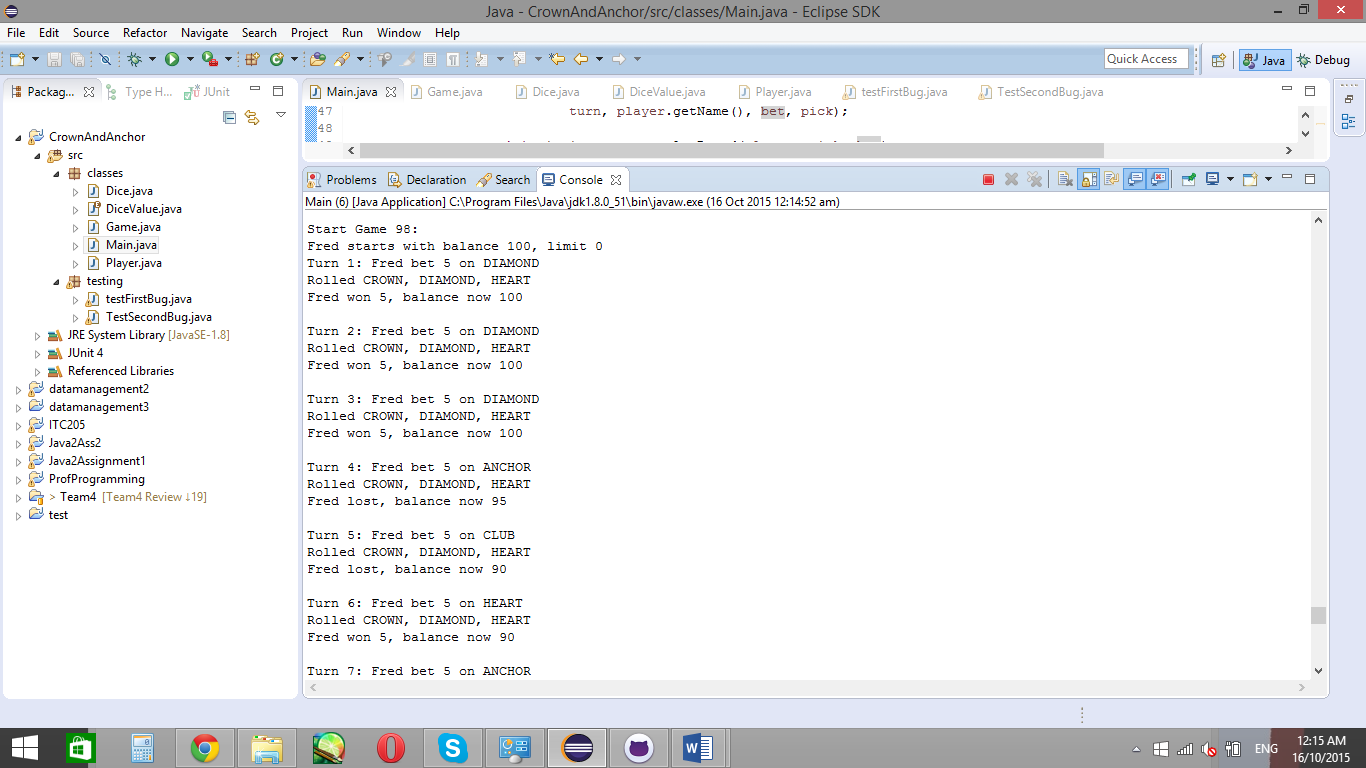
Assessment Question: Log Book

Debugging Log Book

# Bug 1

Bug 1 has the issue of the players balance appearing to not update after their chosen symbol appears on the top face of one die. The error in this bug makes the bug seem to go against the rule of when the player wins with their symbol on one die, they get 1 to 1 paid back. That is, if the player bets 5, they should be paid 5.

However, by looking at the code further, there appears to be no error at all, and the way the output is displayed to the user is the only mistake in the code.



As seen above, Fred wins 5 on roll 2, yet his balance is the same as the previous roll (roll 1). By looking at this, the player would feel as though they’re not being paid like they should, which is not the intention of the program. However, on closer inspection, if the user took 5 from the previous balance (100), they would have 95 (100 – 5 = 95). Then, as their symbol appears once, they get their bet returned (95 + 5 = 100).

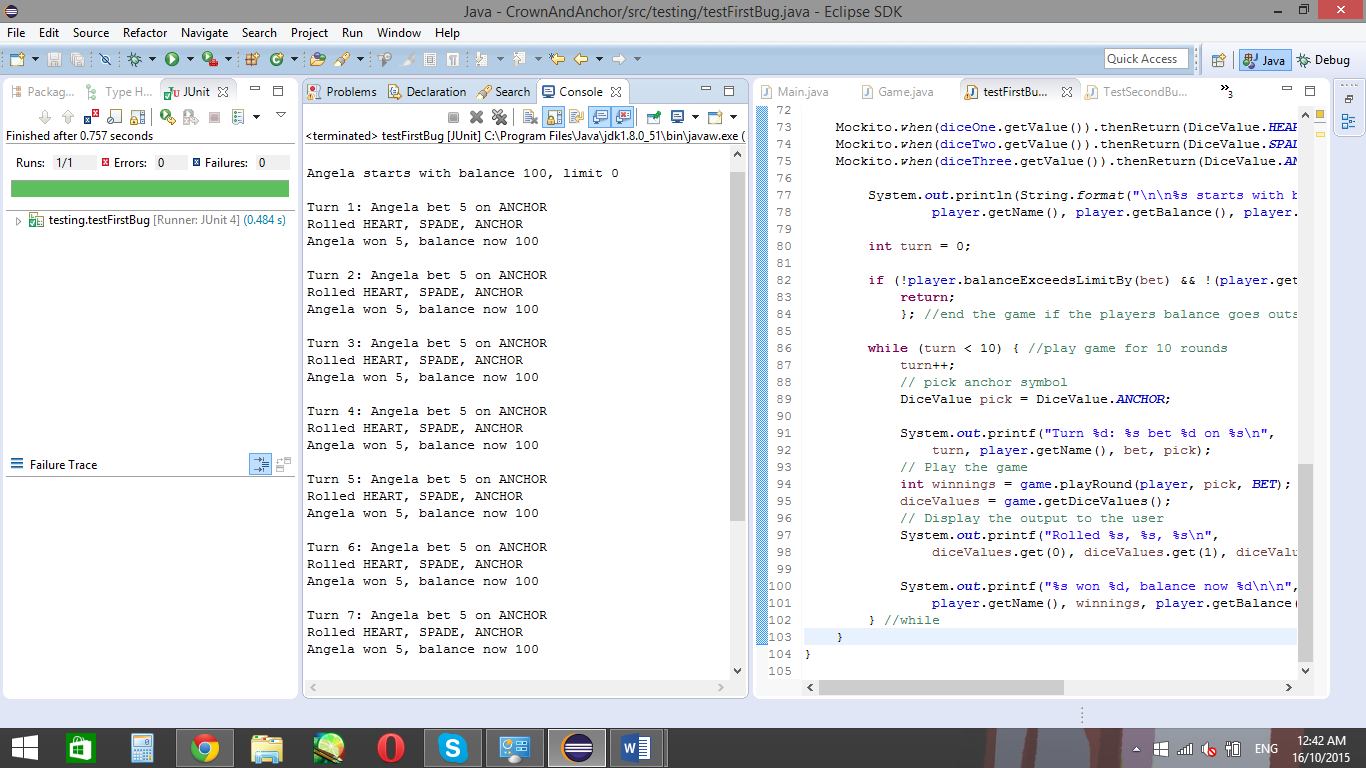
From this, it can be seen there is no error, though the code will be tested, just to ensure this hypothesis is true first before assuming it.

## **Simplification**

The simplified bug is in class testFirstBug, and runs code similar to the code in the main class to display the error. Should the code be successfully replicated, the outcome of the test class is the same as the main class.

As well as this, the code needs to be simplified, yet still have the same outcome. Simplifying the code is so the test class uses less dependencies, thus having less variables in the code which can cause different outcomes.

The simplified test gives the following result:



From this simplified code, it focuses primarily on the bug, instead of the entire output. By focusing on one bug, it breaks down the problem, making it easier to see and understand. In this case, the test focuses on the bug of the balance not updating when the player wins 5 after betting 5.

The simplified code gets rid of any variables, such as the random dice face which appears and the face picked, as shown below:

Mockito.*when*(diceOne.getValue()).thenReturn(DiceValue.***HEART***);

Mockito.*when*(diceTwo.getValue()).thenReturn(DiceValue.***SPADE***);

Mockito.*when*(diceThree.getValue()).thenReturn(DiceValue.***ANCHOR***);

And:

DiceValue pick = DiceValue.***ANCHOR***;

By looking at the test, the test still displays the buggy outcome, where it appears as though the player doesn’t get their winnings back after each play. Further hypothesis and testing would need to be done to determine what causes the error.

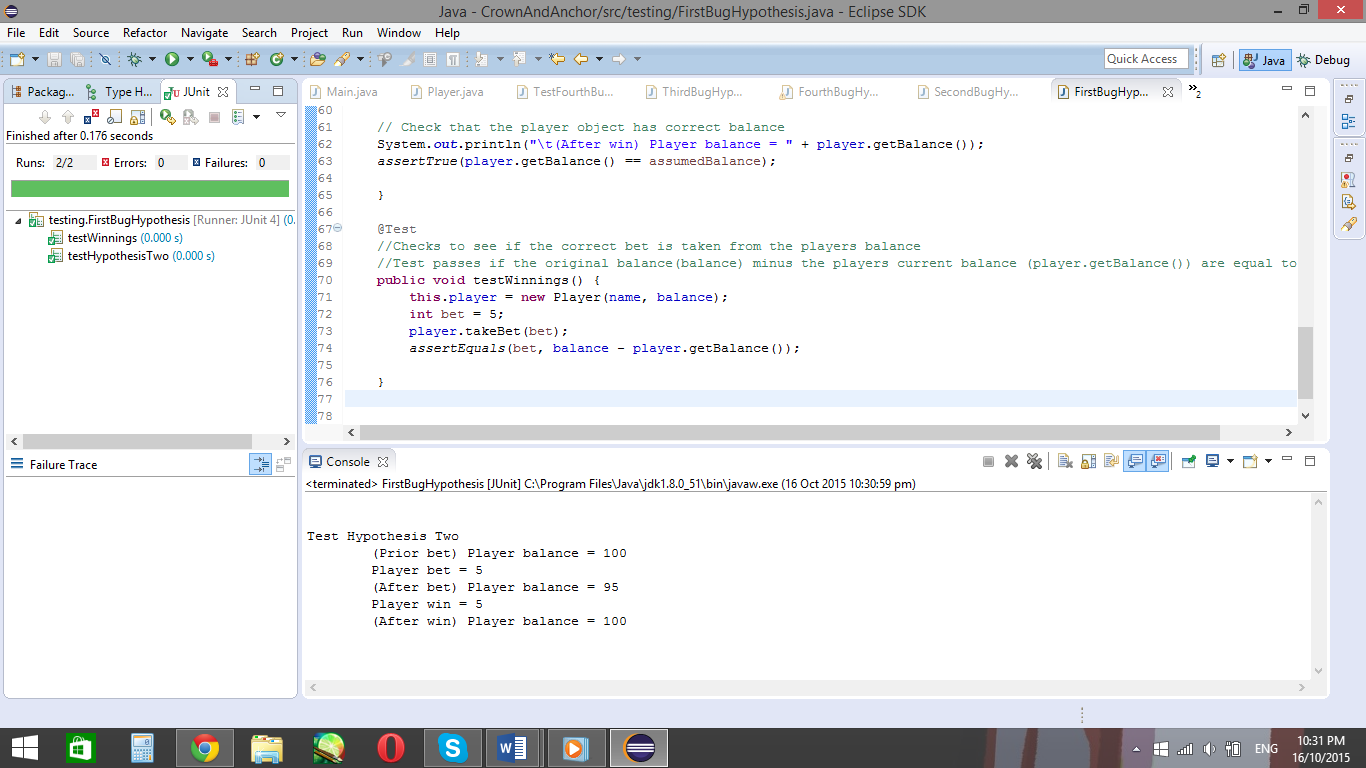
## **Hypothesis**

### Hypothesis 1

The right bet isn’t being taken from the players balance

One of the reasons of the error may be that the player isn’t receiving the right amount of winnings, which is added on to the users current amount after each play. For instance, if a player wins $5 on a play, their account may be getting paid $0 instead of 5, and therefore their balance isn’t updating as it should.

Testing this hypothesis gives the following code:



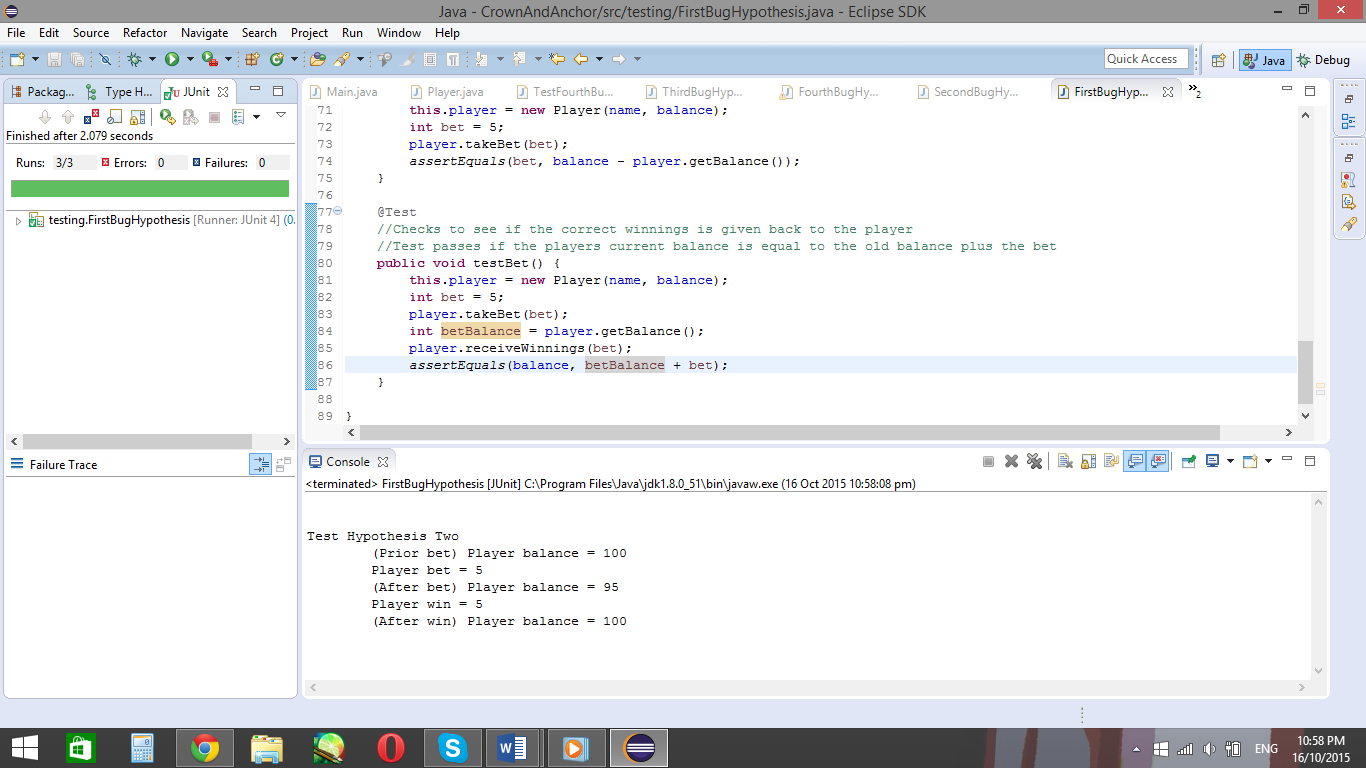
This code tests the bet taken from the players balance and test to see if the bet is equal to the balance (100) minus the players current balance (which should be 95). If this test passes, as it does in the picture above, then the previous hypothesis is proven false.

Since the previous hypothesis is false, then another hypothesis needs to be made.

### Hypothesis 2

The wrong bet is being given to the player

Since the player isn’t getting the wrong bet taken from them, then maybe they’re getting the wrong bet given back to them. This is done in the test testBet.

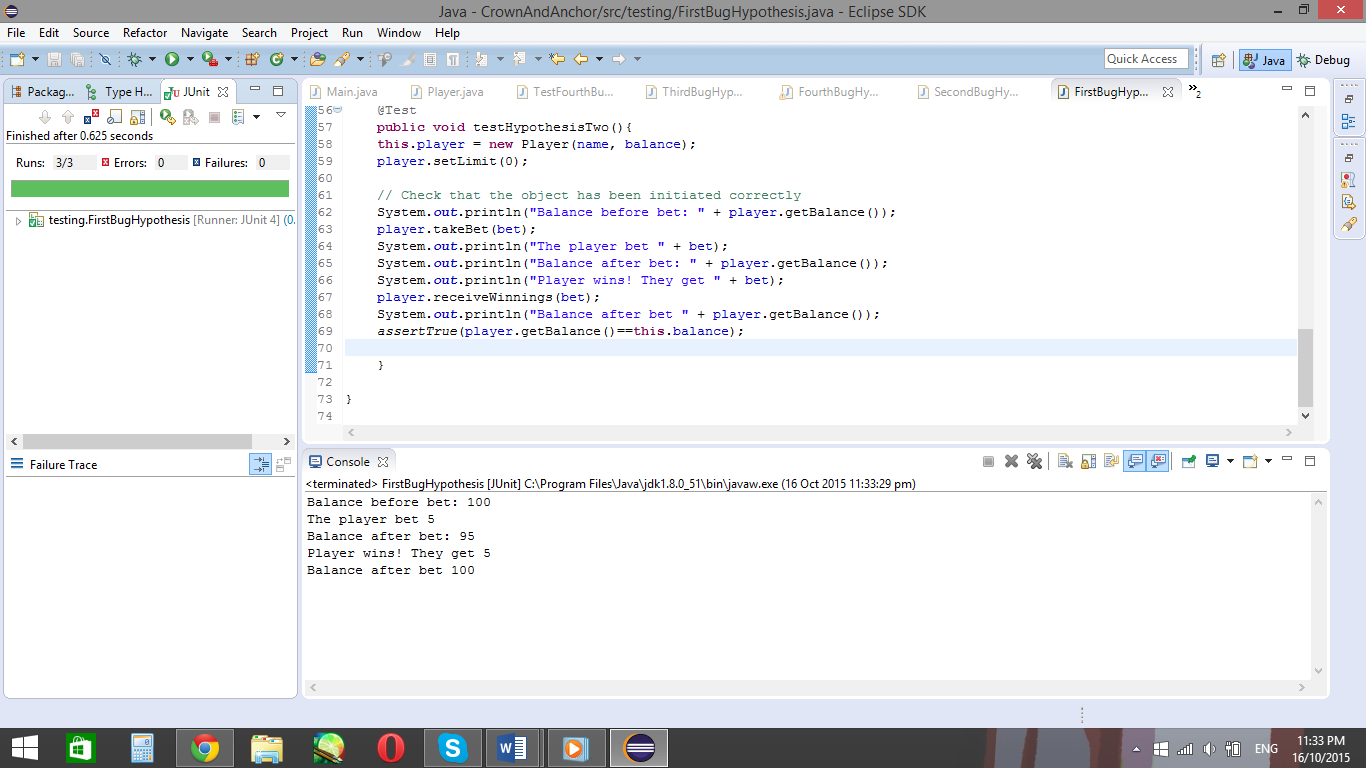


This test also passes, so no bugs there either.

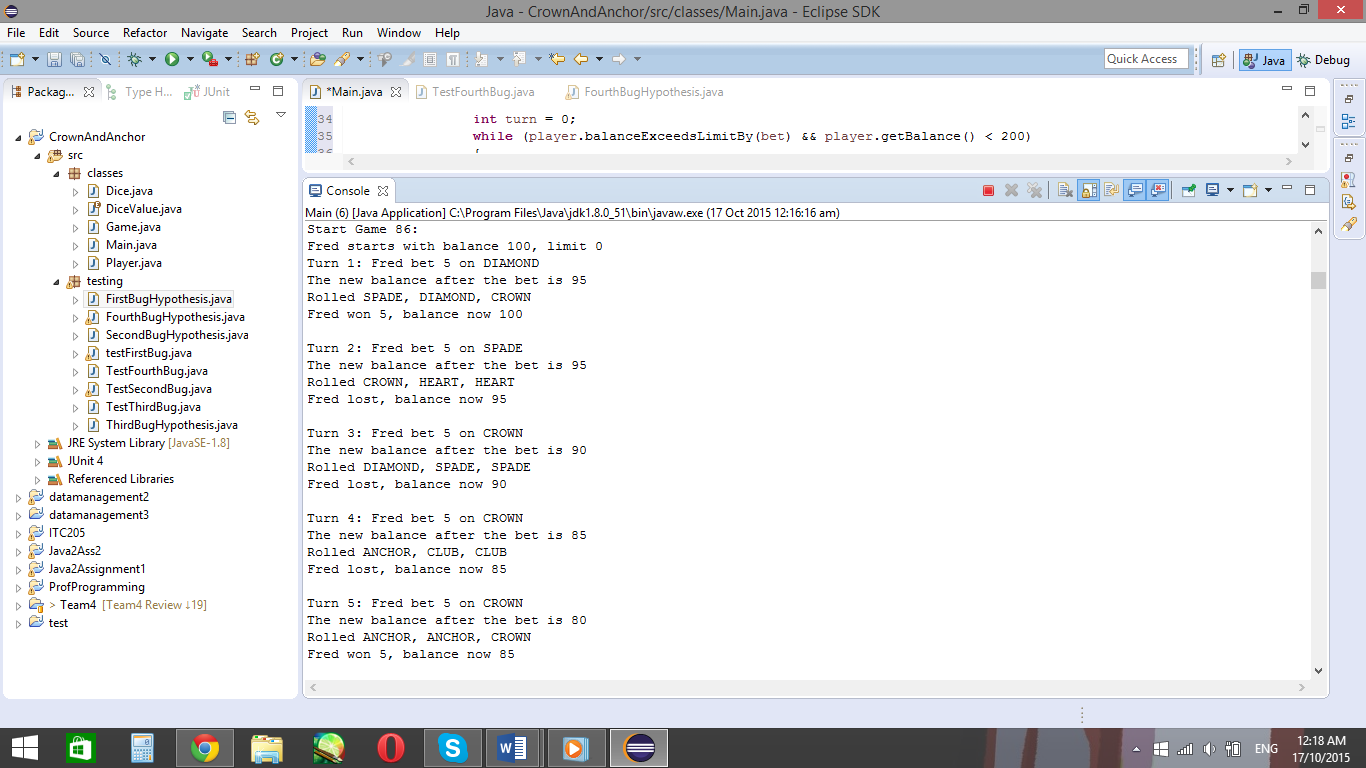
### Hypothesis 3

The payout is correct, the output is just confusing to the user

The third hypothesis is that there’s no error at all, and the way the information is printed out to the user is confusing. One way to fix this is by printing out the players balance after the player takes a bet. This hypothesis is tested in showBetBalance in FirstBugHypothesis.



By looking at the outcome, it shows the balance does update. The reason it looks like the balance doesn’t update in the Main class is because it doesn’t give an update to the player of their balance after they take a bet.



Implementing the method into the main class gives the updated balance to the user.

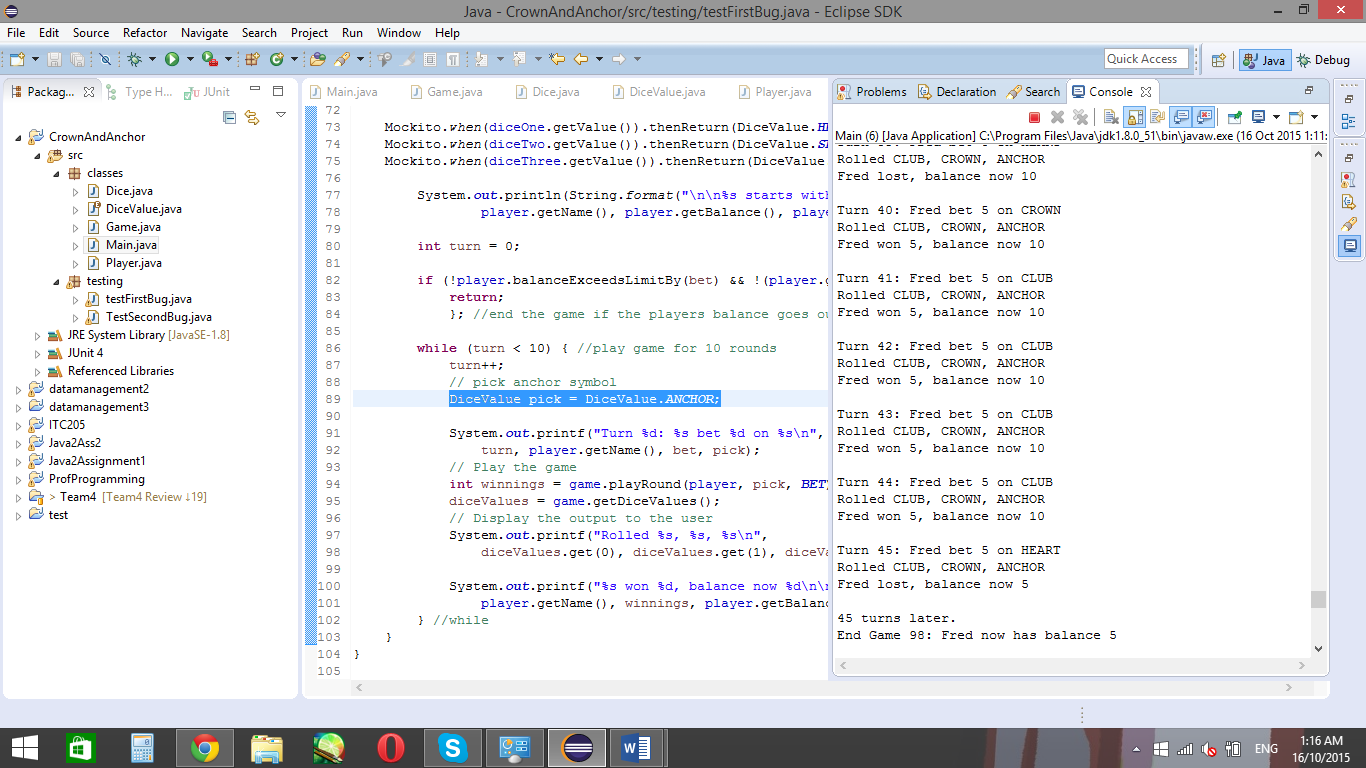
Code used:



From this code, the balance of the player is given to them after the bet.

# Bug 2

The second bug in the program occurs when the game choses to end. In the set rules, the game should end once the user reaches 200 in funds or they try to place a bet with zero funds. Therefore, the player should have 0 or 200 funds in their balance when the game finishes.

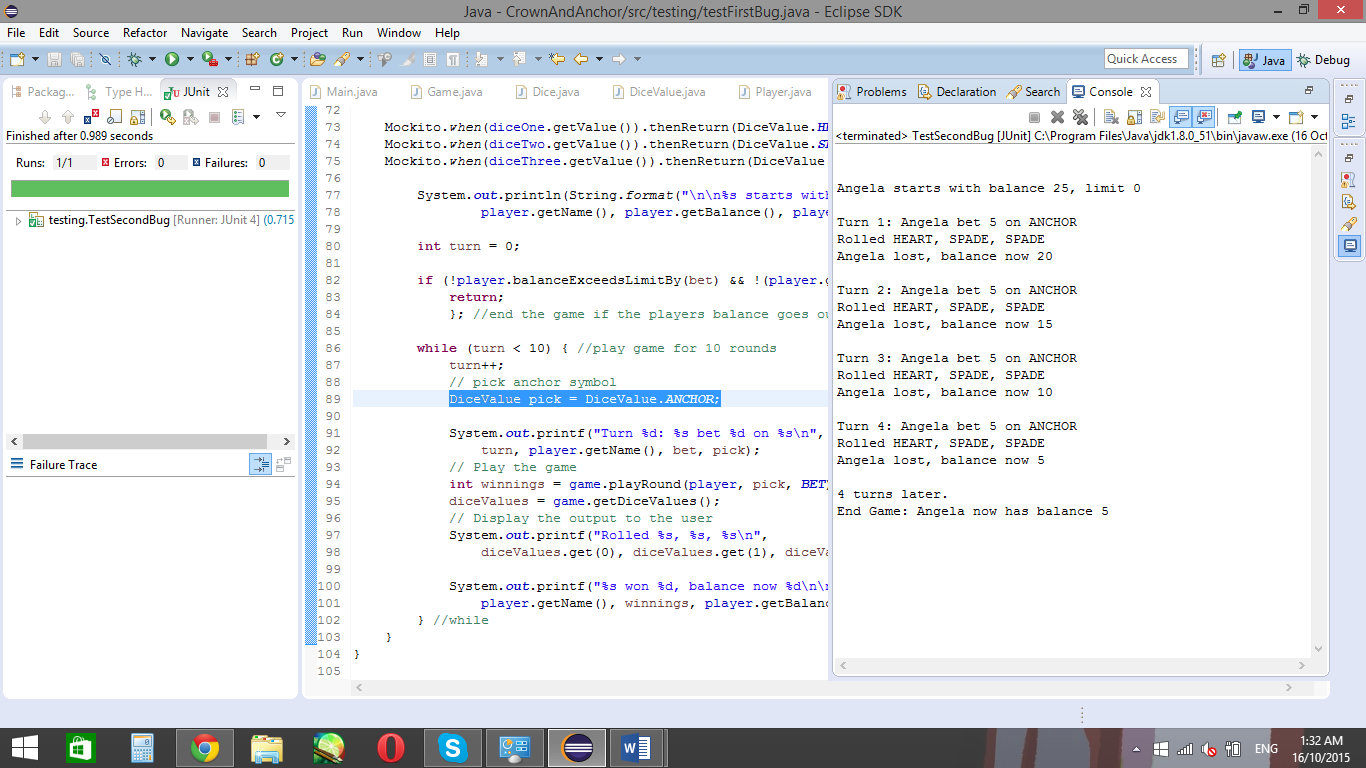


However, as seen in the code, the player still has $5 of funds still left in their balance when the game finishes. Therefore, there is a supposed bug in the code which makes the game stop when it’s not supposed to.

## **Simplification**

The bug is recreated in the Junit test class testSecondBug. In this class, it uses mocks to get rid of any dependencies, and uses injections only when it needs to. As well as this, it also gets rid of any randomisations, such as the die face, and focuses on the end balance.

The simplified test runs code similar to that of the main class. This is because the main class is what prints the result to the user, and therefore the code which is run from the main class is the end of the code before the user sees it, and where to start debugging the code when debugging it backwards.



The shown output of the tests displays the bug still, where the game ends, even though the player still has enough in their balance to make another bet.

From this output, the test focuses on the end balance, where it doesn’t give the player the choice of winning anything. The test has been rigged to always take $5 from the user after each turn, to simplify it down and make the testing process faster.

In the testBugTwo class, the code for the dice is as follows…

Mockito.*when*(diceOne.getValue()).thenReturn(DiceValue.***HEART***);

Mockito.*when*(diceTwo.getValue()).thenReturn(DiceValue.***SPADE***);

Mockito.*when*(diceThree.getValue()).thenReturn(DiceValue.***SPADE***);

DiceValue face = DiceValue.***ANCHOR***;

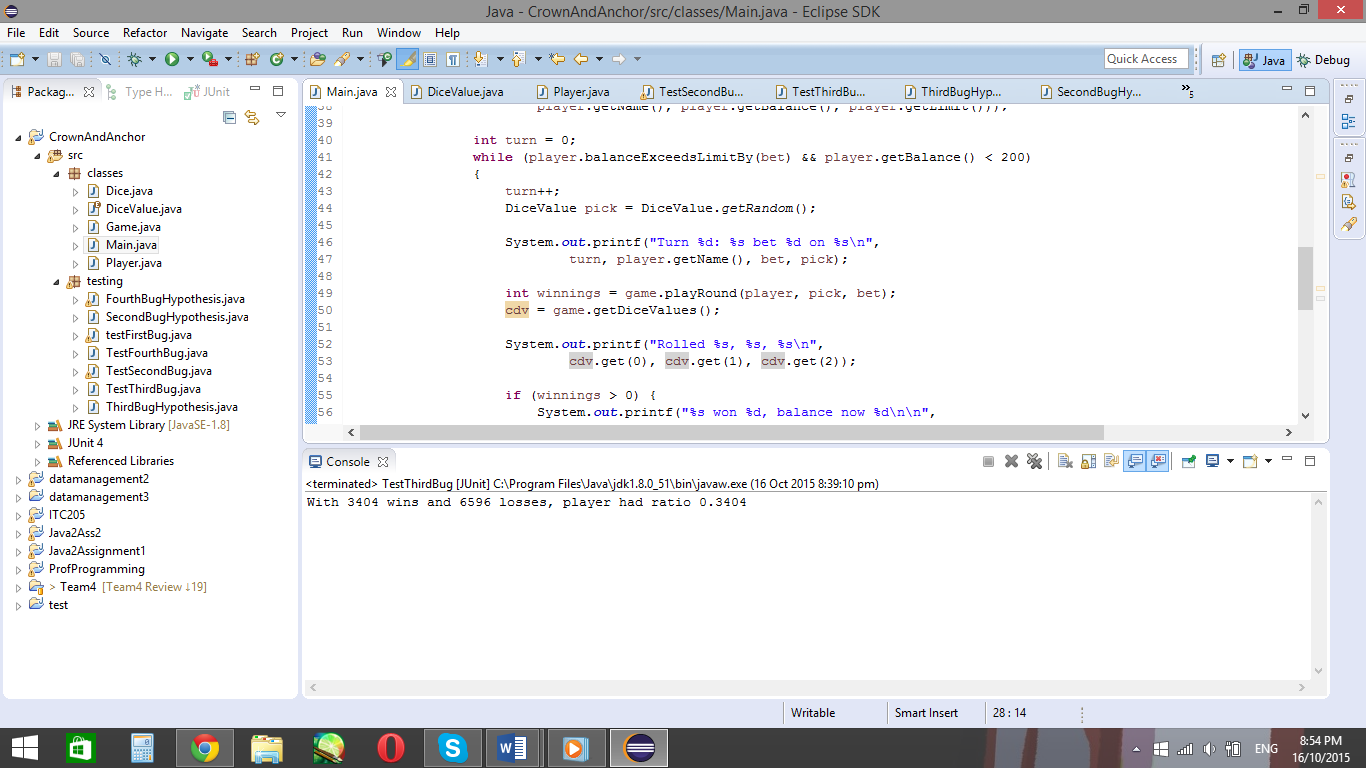
…Where diceOne, diceTwo and diceThree are mocks of the Dice class.

## **Hypothesis**

### Hypothesis 1

The Boolean expression which allows the player to make a bet is incorrect.

Before a player can make a bet, the players bet and balance are checked in a while statement:

where the players balance and bet are checked to see if the bet won’t go below the players balance. In this case, if the players balance is below $200 and the bet won’t exceed the leftover balance, the player can take another bet.

In this case, the test is only concerned about when the players balance reaches zero, so the next step is to trace back to where the balanceExceedsLimitBy() method is previously seen, which is in the player class.

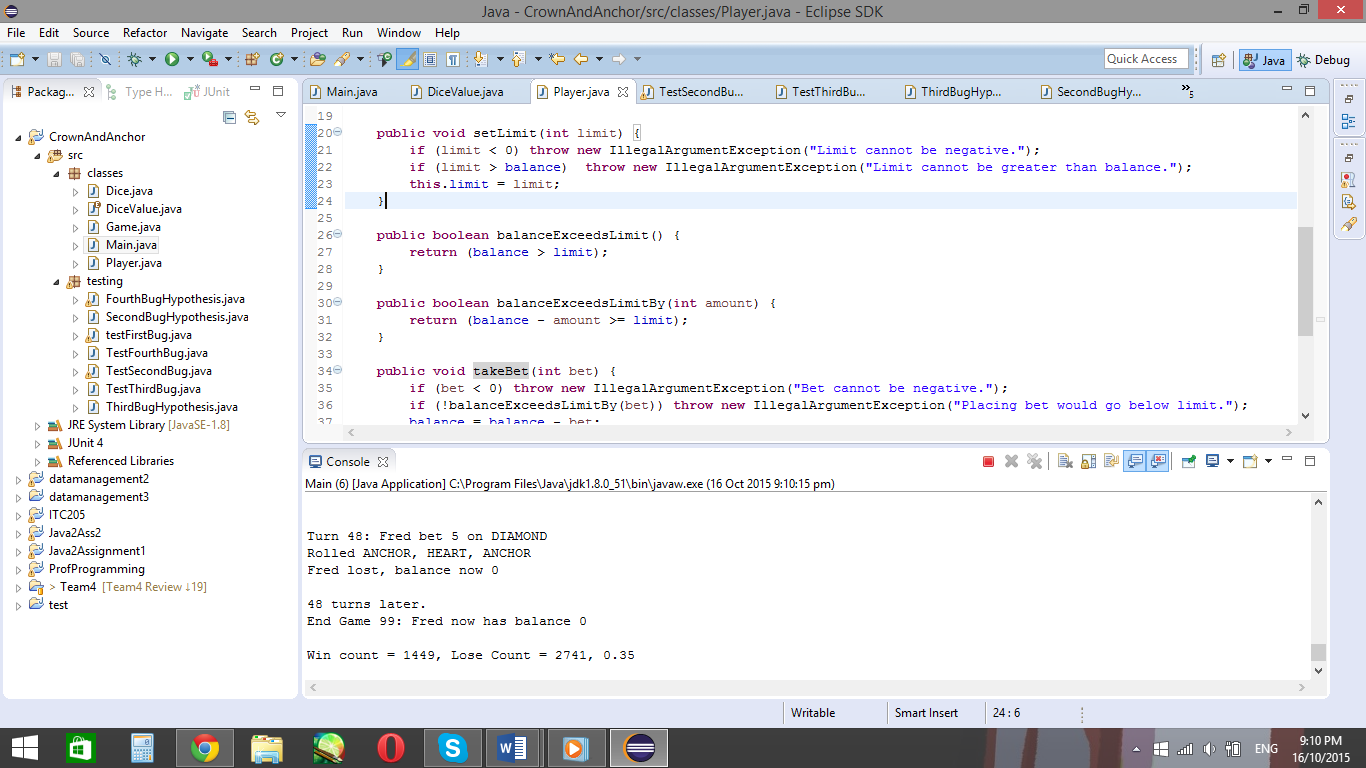
By looking at the balanceExceedsLimitBy() method in player, it’s shown as the following:



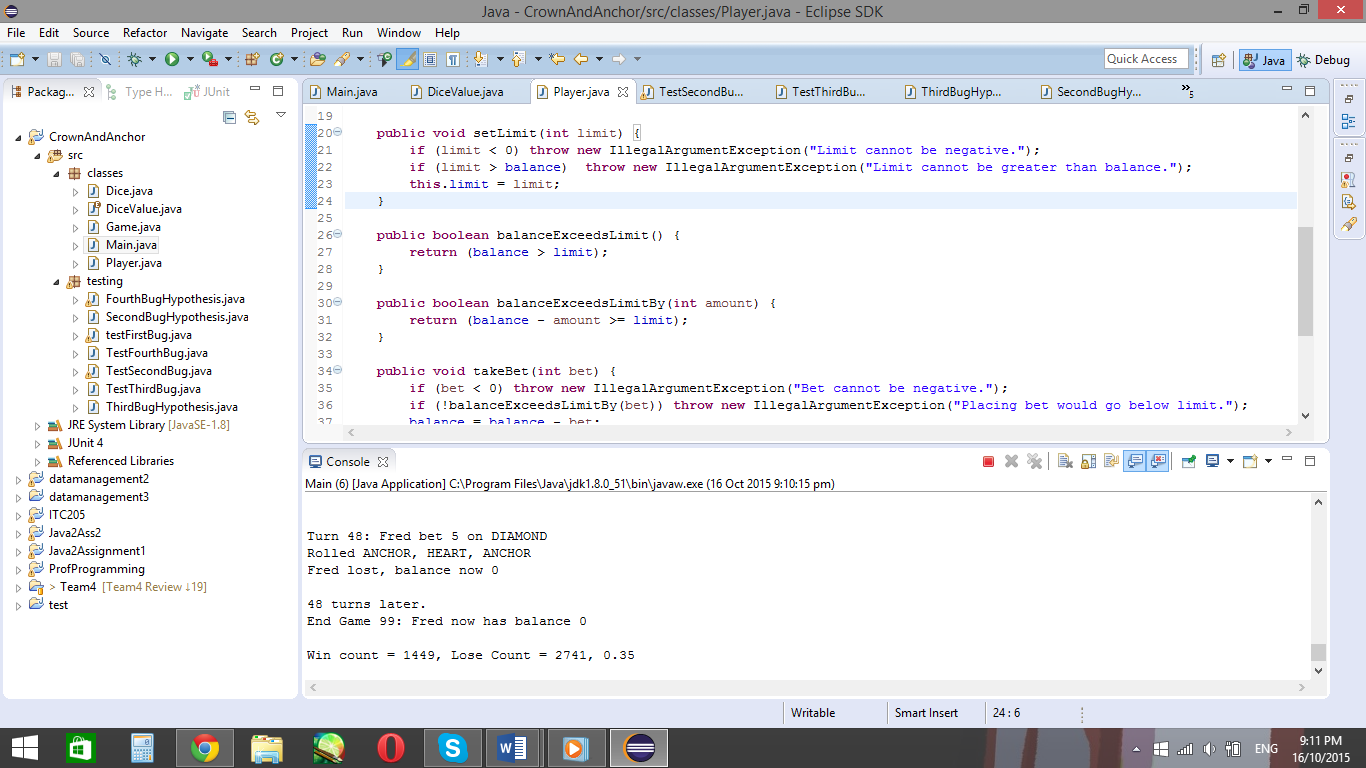
By reading the class further, it says that if the players current balance, plus the amount they wish to bet is more than the limit, than the Boolean is true. So if the player had $5 left, and they want to bet $5, the players balance would be 0, and not the limit, making the Boolean false. Once the boolean’s set to false, the player cannot place another bet. So this could be why the player always finishes with $5 left.

Therefore, changing the Boolean to balance – amount >= limit might get rid of the bug.

Applying this idea in the Player.java class:



And running the Main class again:



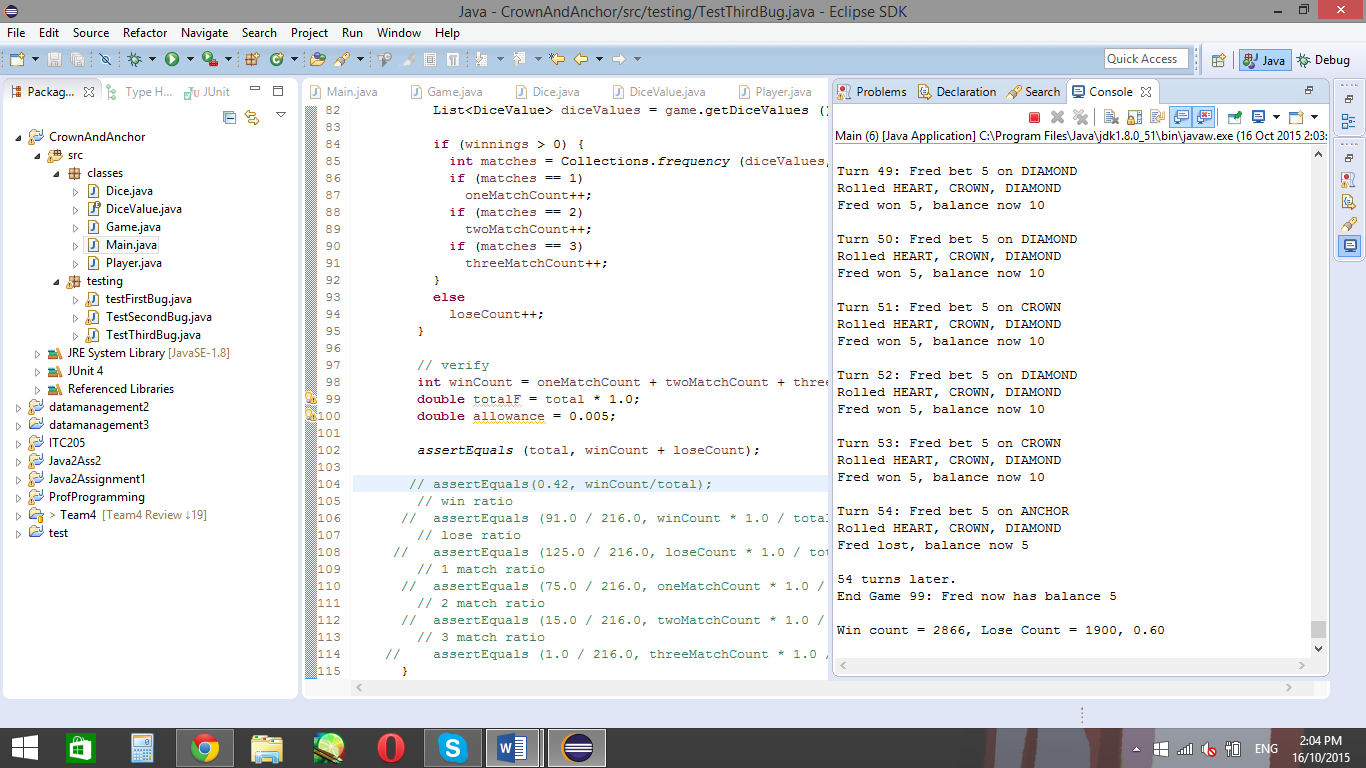
The player’s game ends when their balance reaches $0 instead of $5

Therefore, this output shows the game only finishes once the players balance reaches 0, where it previously ended when their balance reached 5. Since the balance no longer stops at 5, the output shows the bug has been fixed.

# Bug 3

The third bug in the program has incorrect statistics for the win to lose ratio. In the game rules, it states the player should win 42% of the time. That means the players chosen die face should appear on at least one die for 42 out of 100 turns for every 100 turns.

However, running the code gives the following output:

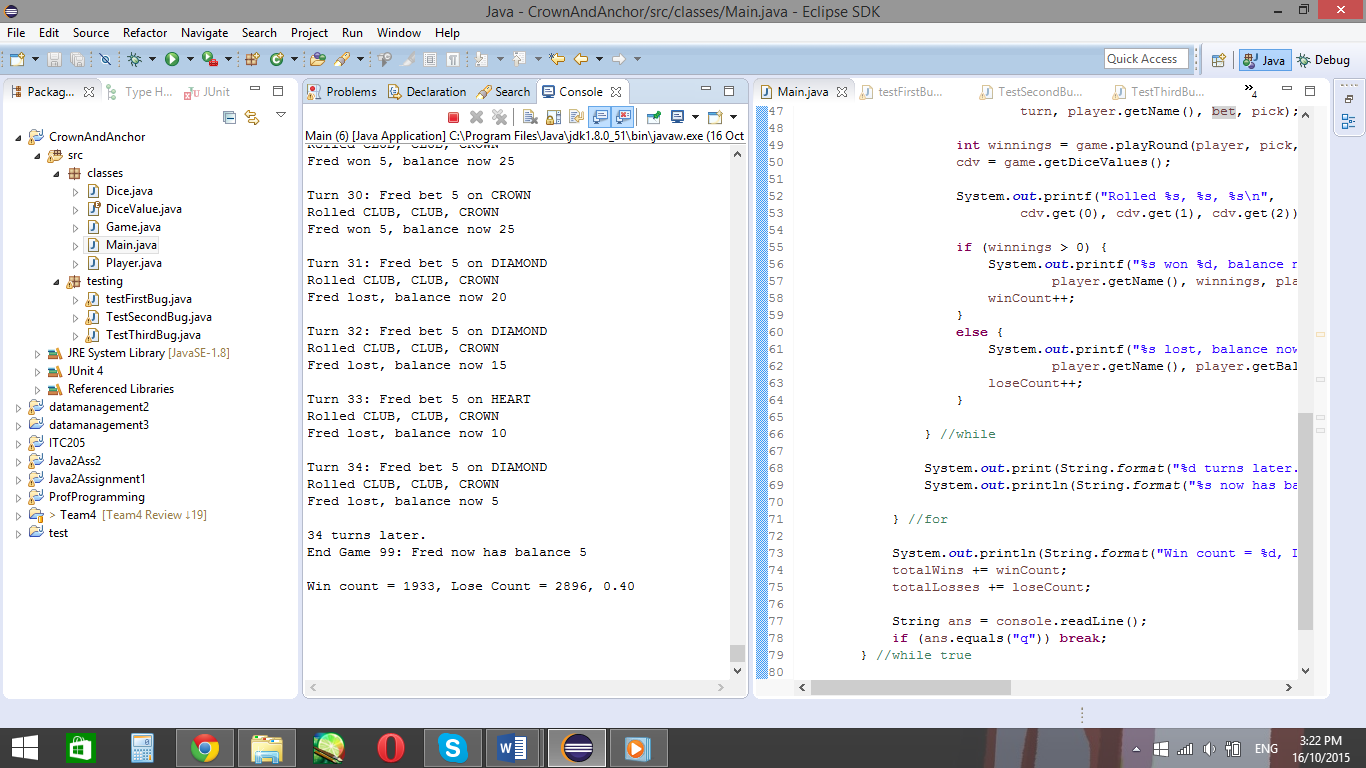


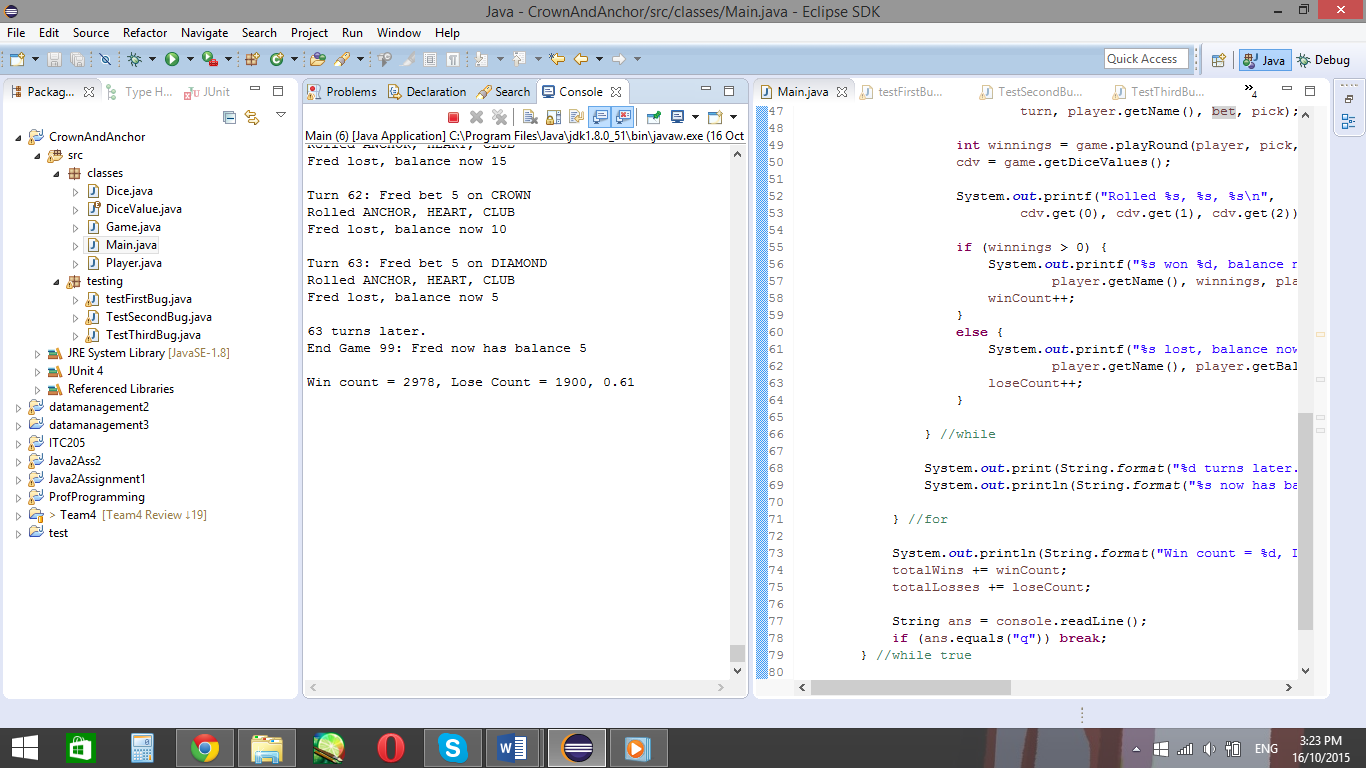
By looking at this output, the player wins more than the expected outcome. In fact, the player wins over at least half the time, where they should be losing 58% of the time. There is a supposed bug in the code which causes the outcomes to appear in the players favour and not be correctly supported by the win/lose ratio.

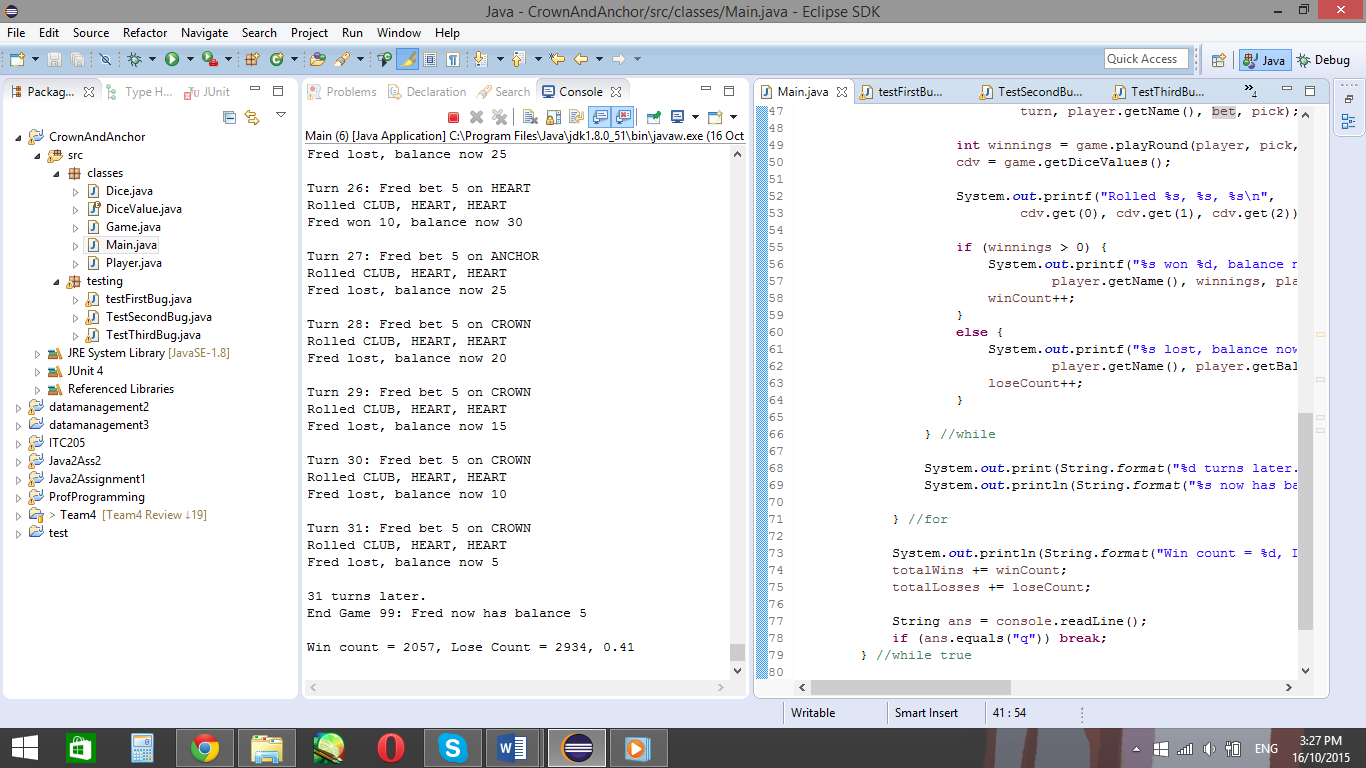
## **Simplification**

This test was difficult to simplify due to the amount of variables. For instance, the win/lose count seems to keep changing between games, so the test which simplifies it will randomly fail.

The changing win/lose from the program:



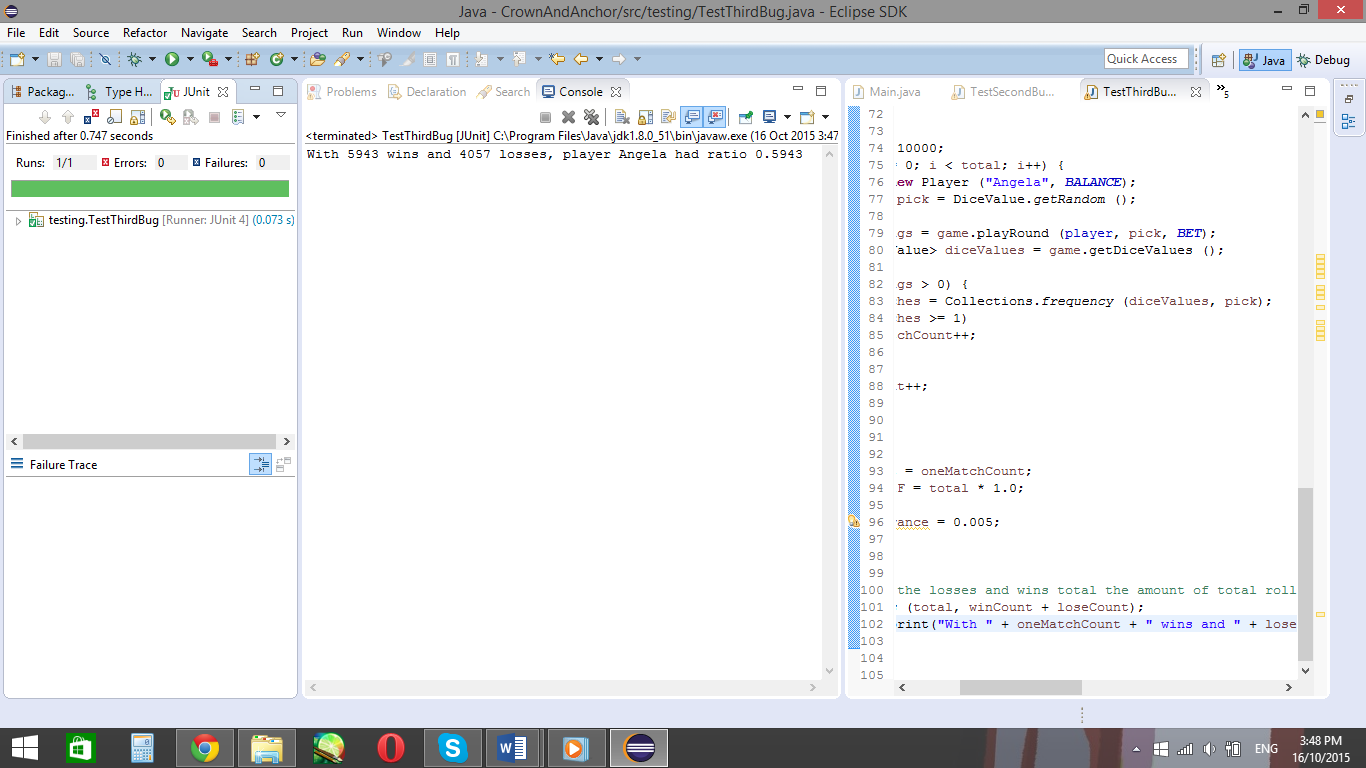


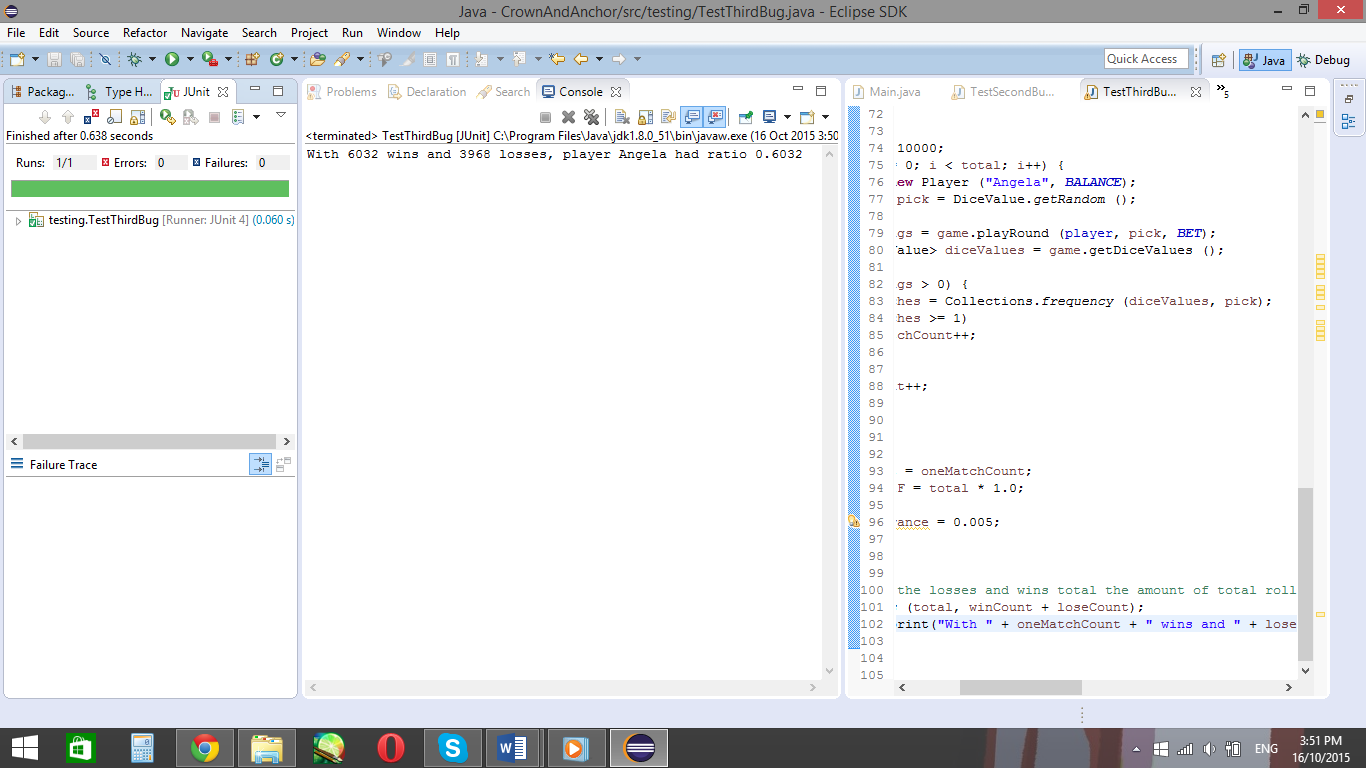


By looking at these three examples, the win/lose ratio keeps changing. So not only does the program have the bug of the incorrect win/lose, it also has the changing win/lose ratio.

For the simplification, I have copied the method from the main class which runs 10,000 runs of the game and displays the win/lose on the console. To simplify, I have gotten rid of the game ending when the player runs out of bets and the payout scenarios. Instead, the test only focuses on the win/lose ratio, as shown below.







Like with the actual program, the test displays that the win/loss ratio fluctuates, which isn’t supposed to happen. Therefore, to successfully fix this bug, the win/loss ratio should stay around it expected 42% win/loss ratio.

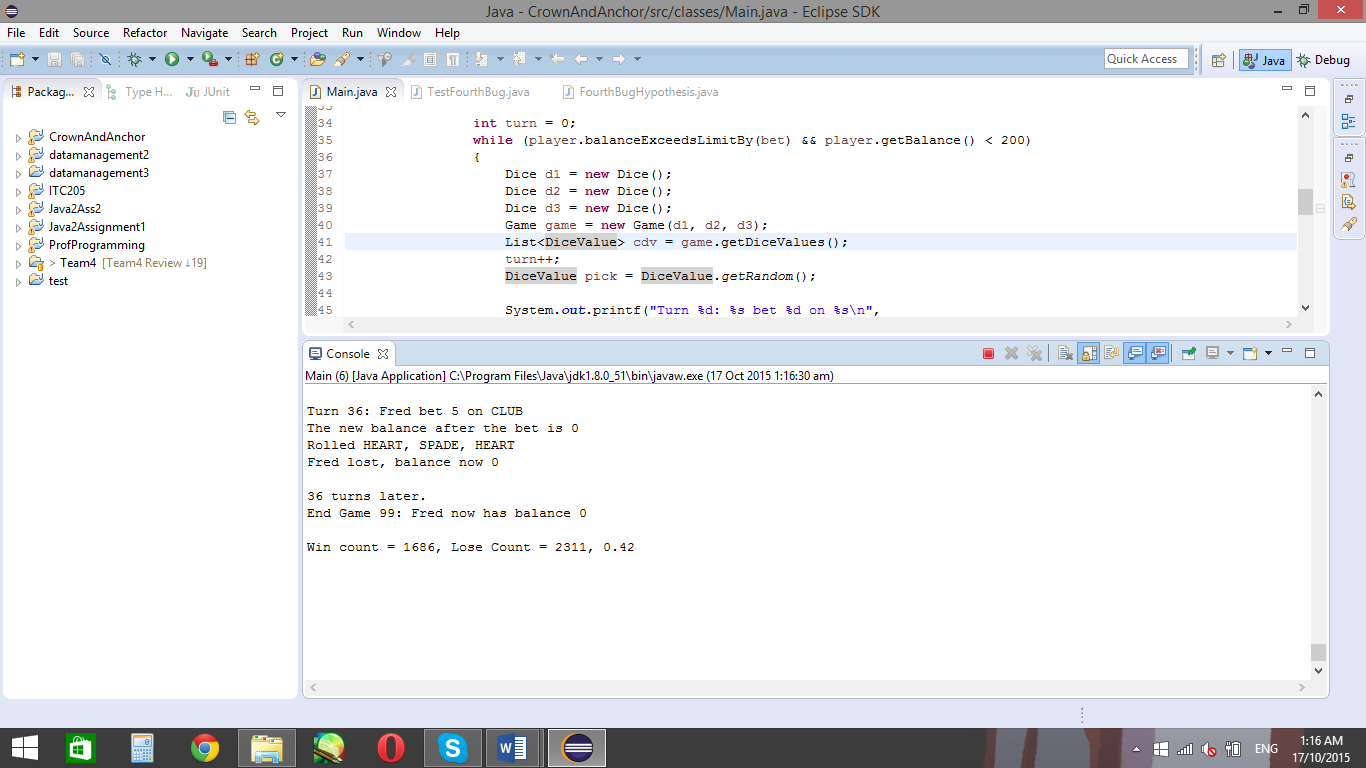
## **Hypothesis**

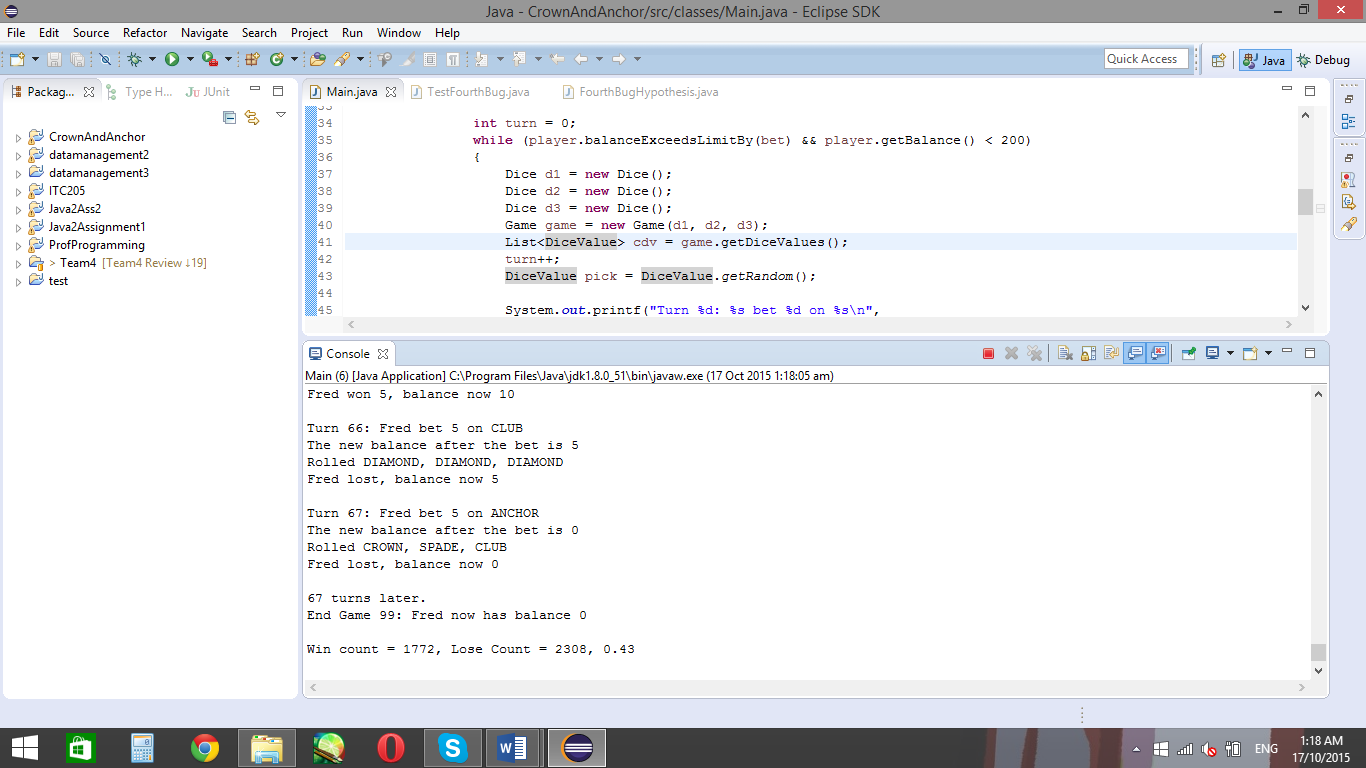
### Hypothesis 1

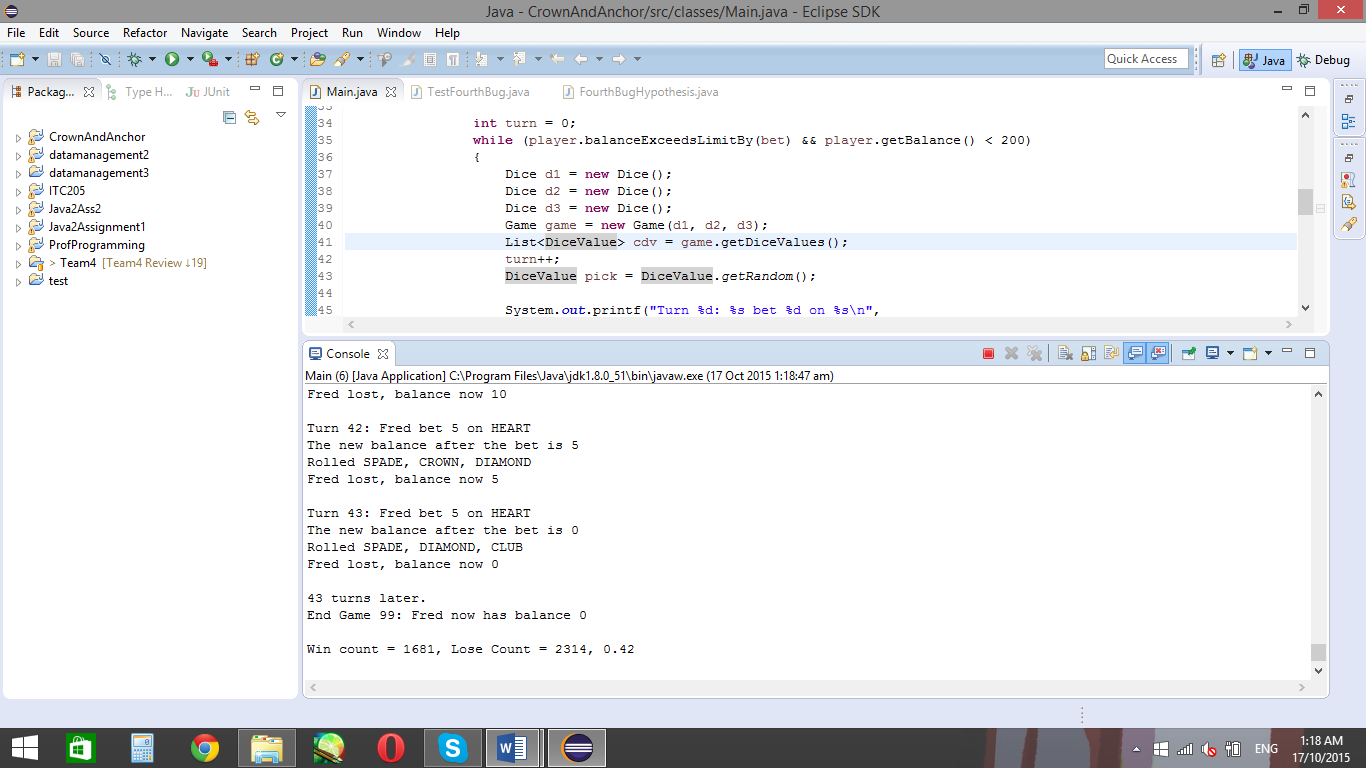
The win/lose ratio is wrong because one of the die faces isn’t appearing

This hypothesis is related to the fourth bug found in the code. Whenever the game plays, one of its faces, Spades, never appears. Therefore, Spades will need to be fixed to see if this fixes the problem.

Running a few games in the Main class after fixing bug 4 shows the following:





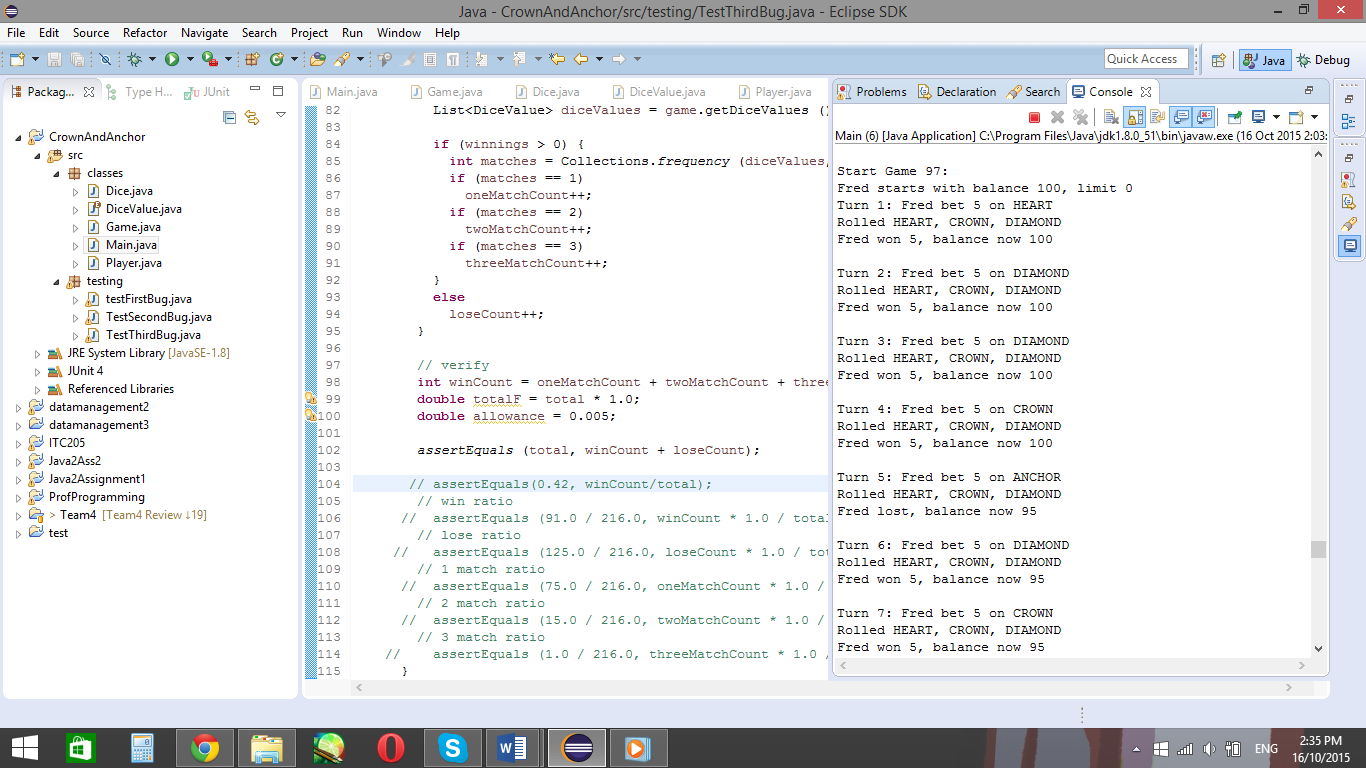


For each game which is run after fixing the fourth bug, the win/lose ratio always seems to be around the 42% mark. This is the aimed mark, as the user should only win 42% of the time. With this new result, and the win/lose ratio not being around 60 anymore, the bug is now fixed.

# Bug 4

The program uses three die which each have a heart, diamond, club, spade, anchor and crown on their faces, each shown at random. Therefore, it’s expected each face ben displayed equal chances as the rest. As there’s only three dice thrown on each turn, it’s then expected the faces change between turns.

However, looking at the output, this does not seem like the case:

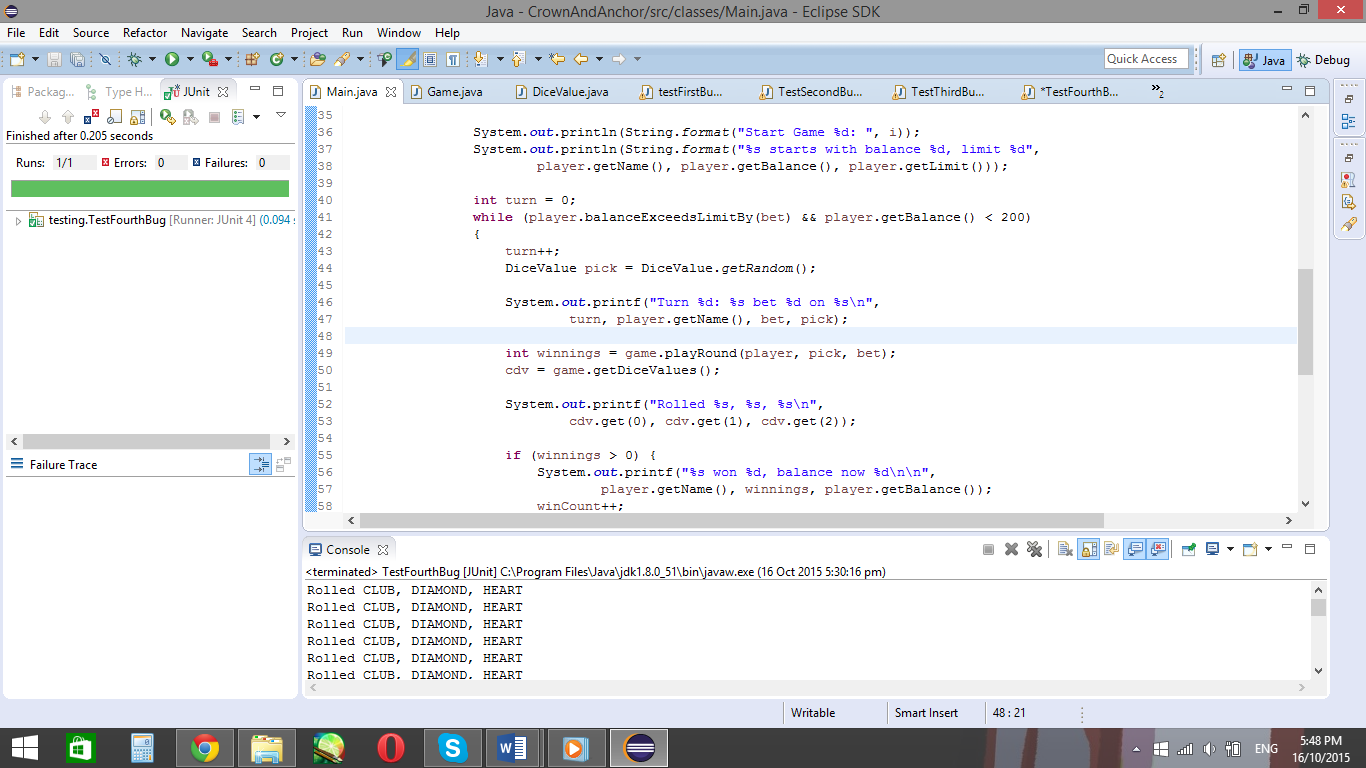


Looking at the code output above, each turn shows the same three faces, Heart, Crown and Diamond appear. Since each face should have equal chances to appear, this causes a problem, as only three of the die faces appear. It’s also displaying the same three face and not changing their order, so the player only ever receives $5 from their bets, and will never have the chance to get two or three faces in the same roll.

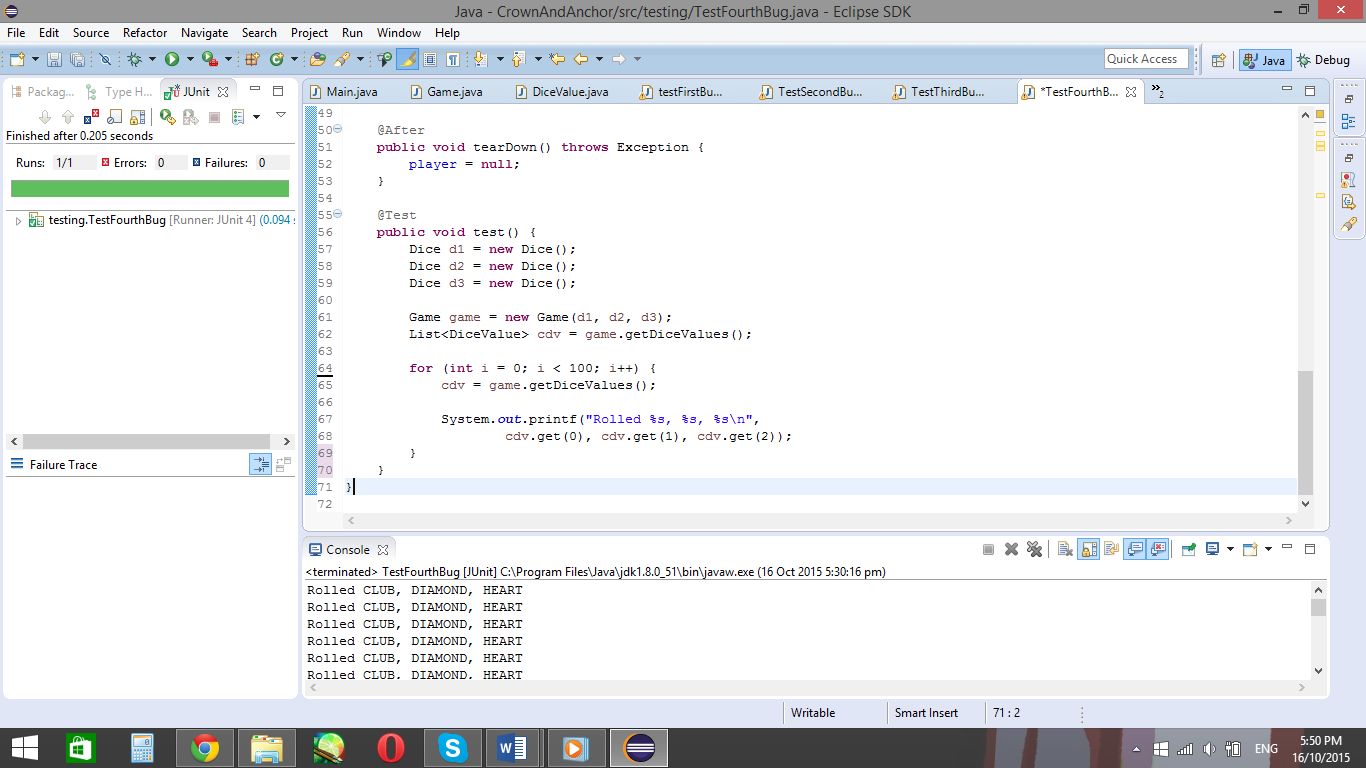
The bug in this case is that the die faces never change between turns. This bug is present until the die faces change between rolls and each die face appears as often as the other five die faces.

## **Simplification**

Using simplification, I have shortened down the bug in TestFourthBug. This Junit test only tests the output of the three dice being rolled, using the same technique used in the main class. Simplifying down the method got rid of the player and the players bets, though kept the dice output. However, the main class has loops which stopped once the user had no balance left. In this case, I have made the three dice be thrown 100 times in a for loop.

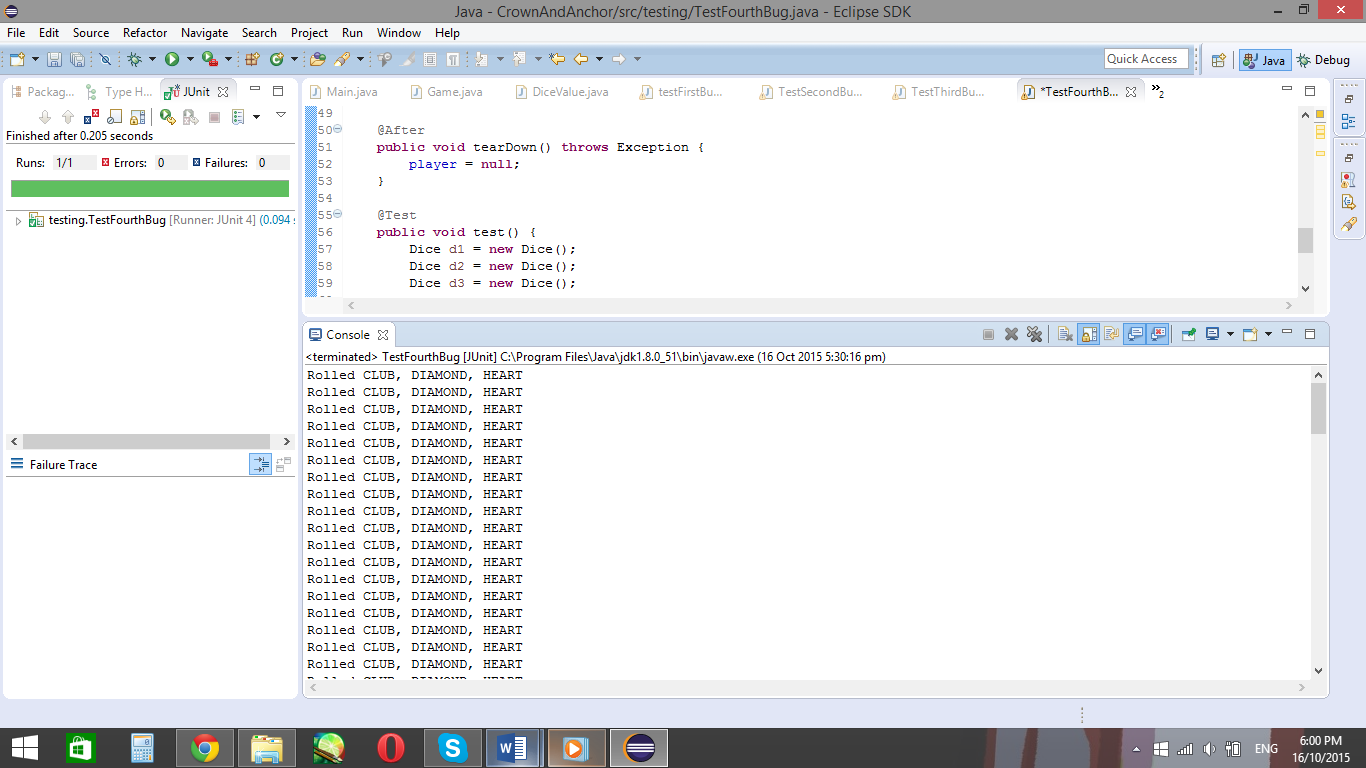


The original code in the Main class

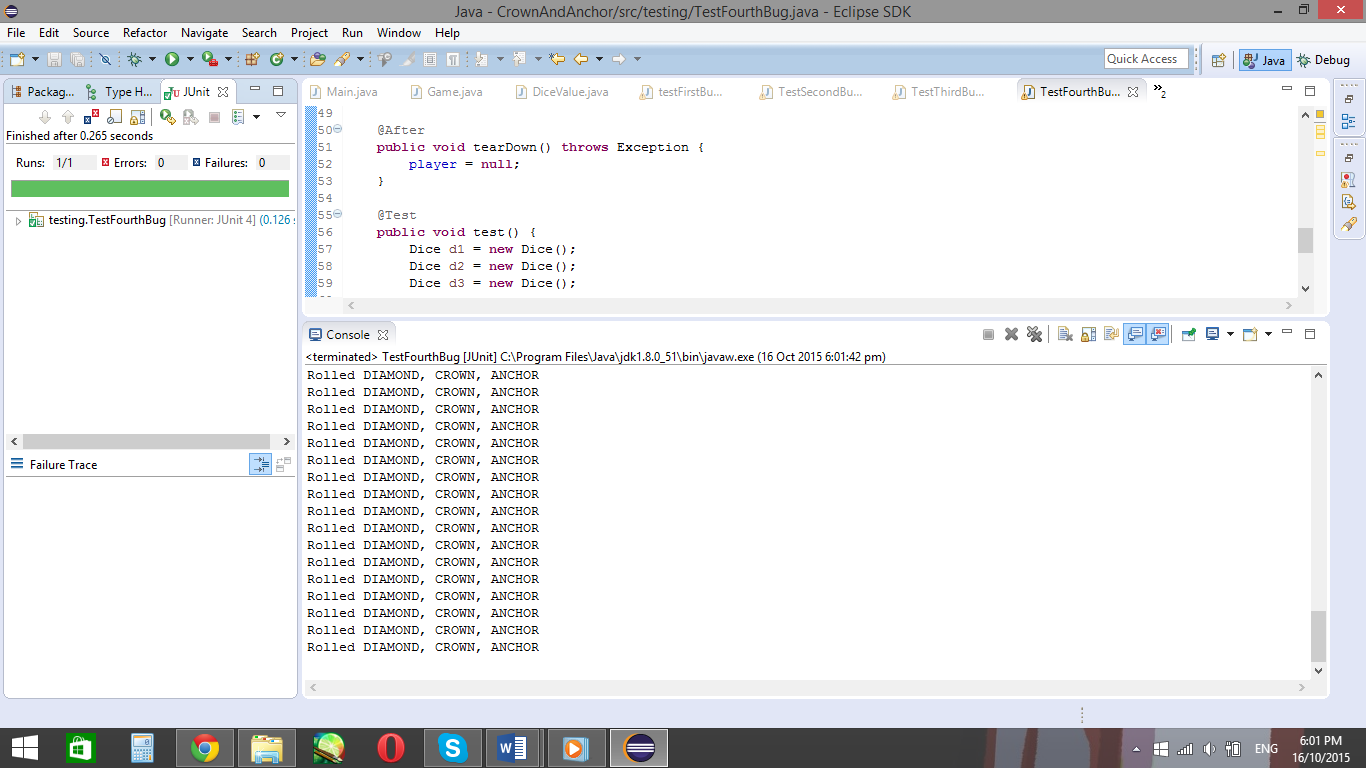


The code used in the test class

After running the test class, it prints out the following:



Running the test program shows the test prints the same output as the Main



Running the test again has different die faces, meaning the faces change after each run of the test

Looking at these outputs, it shows the dice faces change after each run of the test. This is also what happens in the Main class. Since the simplified test works, the bug then needs to be solved and fixed.

## **Hypothesis**

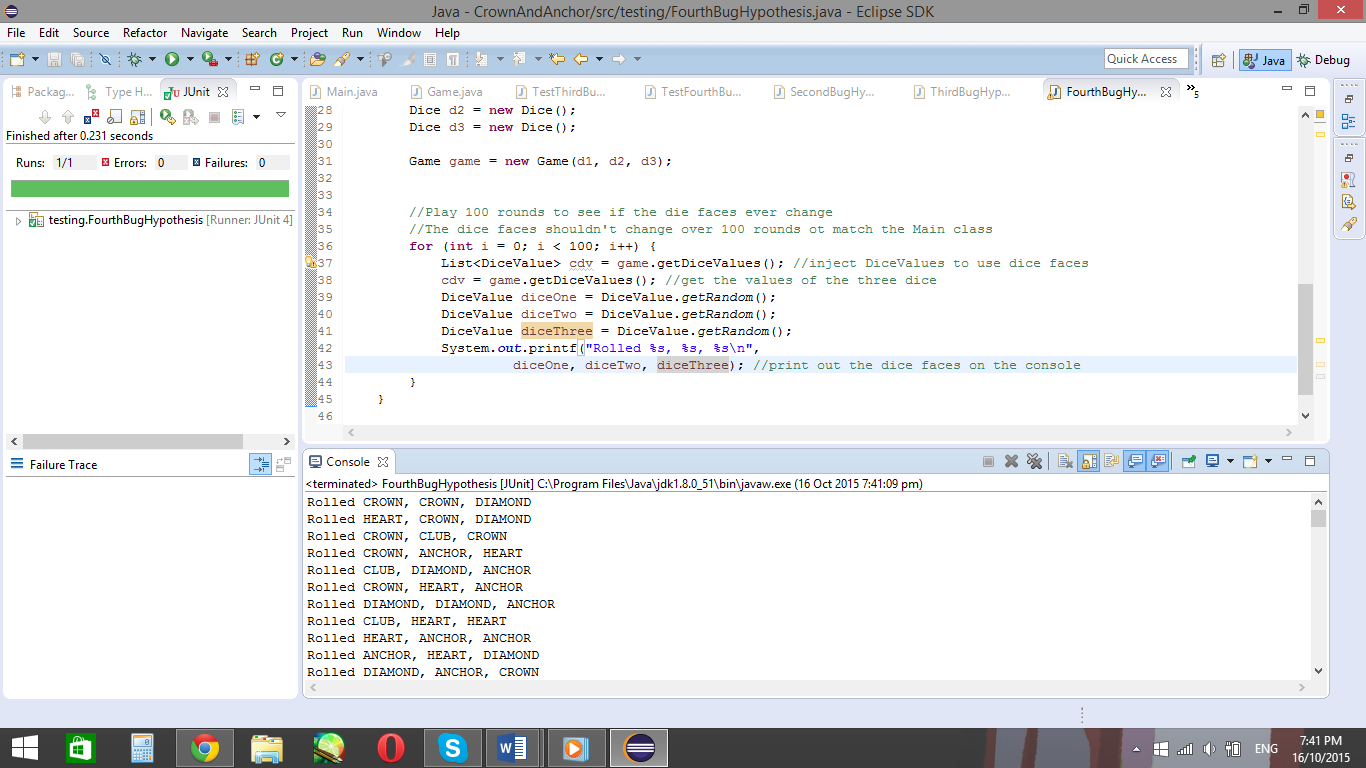
### Hypothesis 1

Die faces don’t change because the dice faces need to be done inside the loop

In the code, the player chooses a random face assigned to the variable pick using the method DiceValue pick = DiceValue.getRandom(); This generates a random die face for the users pick each turn, but what about the die faces? The die faces are assigned outside the loop, using the constructor game.getDiceValues. So from that, it means the same three chosen faces will be repeated over and over for the assigned three dice, and the three dice will always have the same three symbols.

The reason the dice faces change when reloading the program is because the game.getDiceValue method is called again. So repeating thegame.getDiceValues inside the loop should re-run the code and chose different die faces for each turn.

So testing this hypothesis…



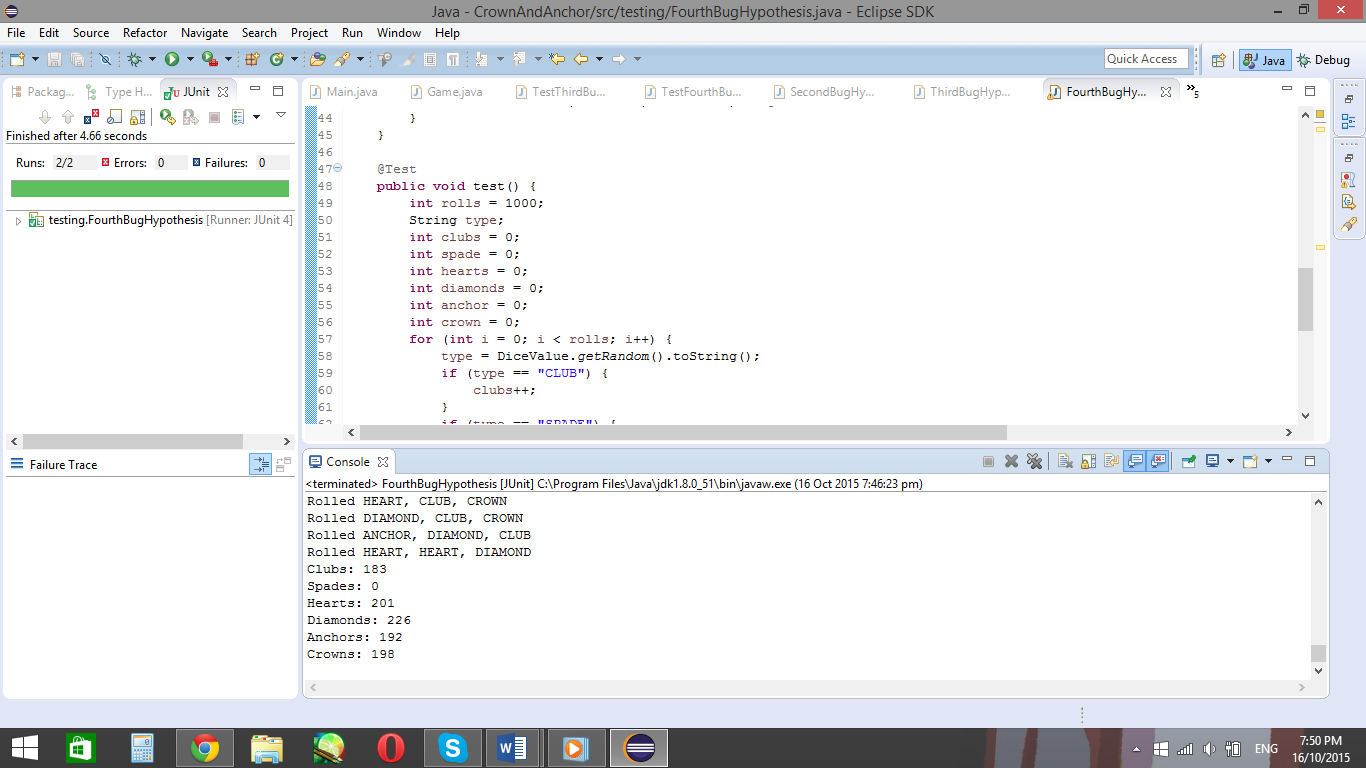
The values now change after each roll. However, it seems to only show Crown, Diamond, Heart, Club and Anchor, but no Spades. It would be wrong to assume that there’s no spades, so another test will have to be written to ensure if spades appears or not.

### Hypothesis 2

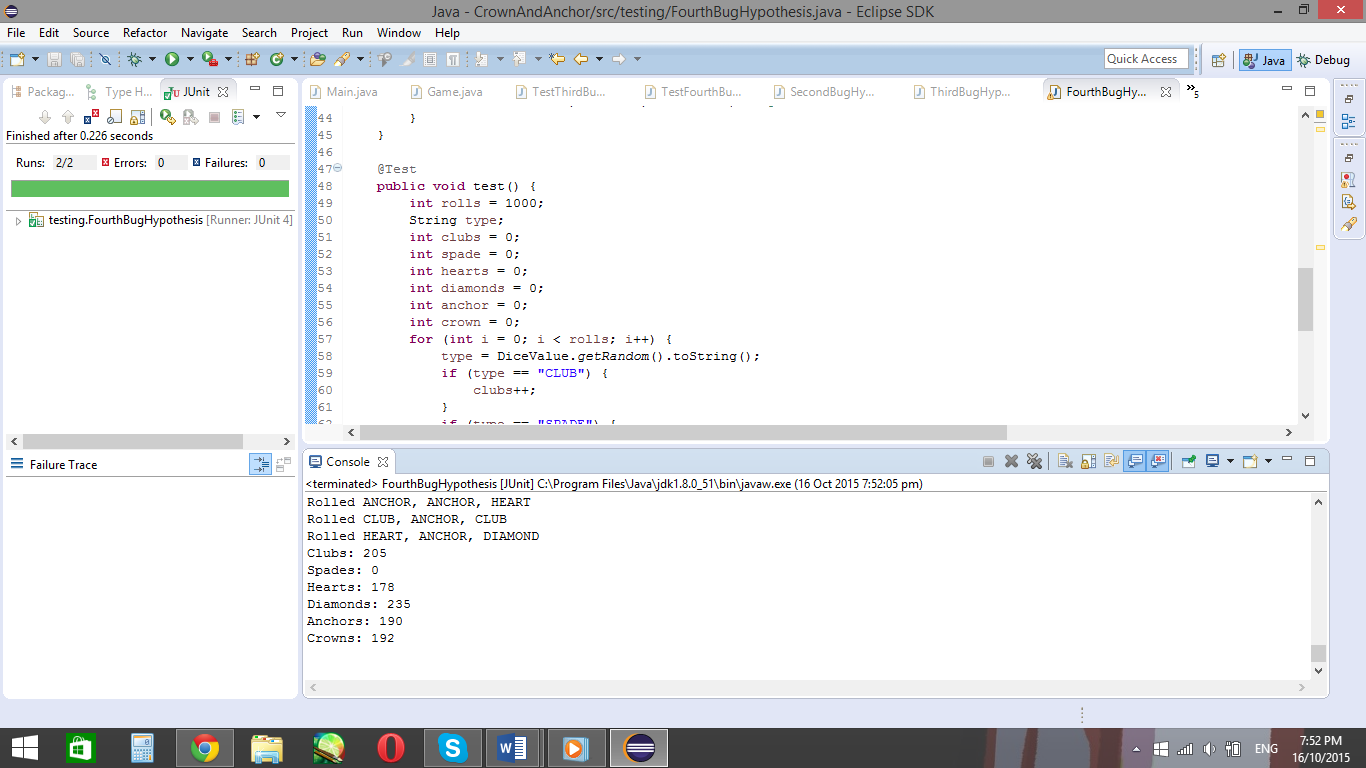
Spades don’t appear in the results

If this hypothesis were correct, it could be the reason why bug 3 exists (where user wins more than they should). This would be because the user has a 1 in 5 chance of winning, instead of 1 in 6. To test for this bug, the dice.getRandom() method is used in a for loop to roll dice 1000 times. For every face on the die, an integer is created. Then, as each face appears, the face integer will increment and be displayed in the console once the loop has finished.

Testing this hypothesis gives the following results:



Counted face values from a first run of the loop



Testing from a second run for clarity. Again, no spades appear

So from this test, it shows that no Spades appear after 100 random throws. The other faces values, however, change. So the getRandom() method in Dice is incorrect.

Since getRandom() is the first instance of the error, I traced back to the previous instance of the getRandom() method, which is the method it’s created in. One of the main things which stands out is the method used to get the random faces of the die, which is the ordinal method.

The ordinal method in Java is the method which randomises up to a certain item in a set, but not the item itself. In this case, the ordinal method counts up to the SPADE, which is the last item in the set. This means the five faces before the spade are randomised, but not spades itself, meaning spades will not appear in the getRandom() method.

An easy way to solve this is to place another value after spades. This value is the new end of the set to be randomised, so adding a number at the end, such as ‘1’, will make the ‘1’ the new stop.

Before the edit:

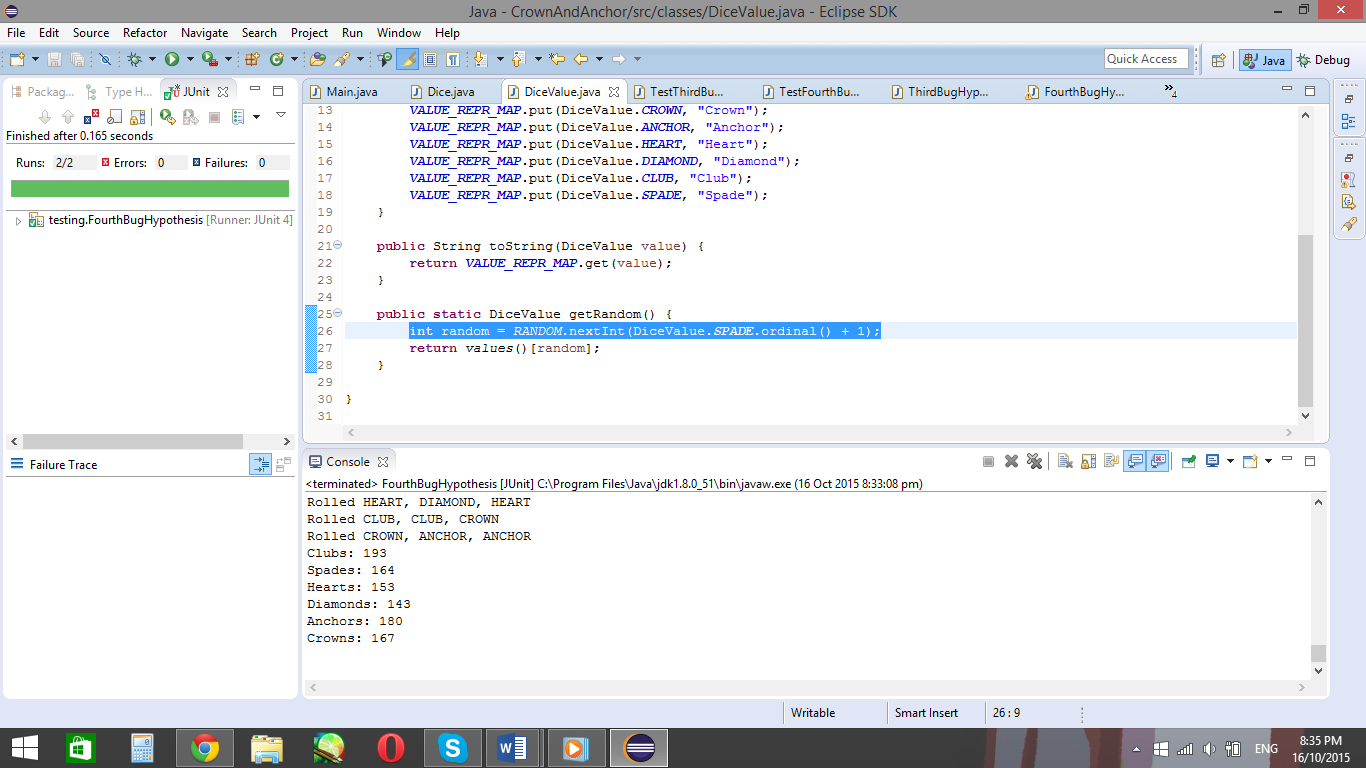
**int** random = *RANDOM*.nextInt(DiceValue.***SPADE***.ordinal());

After the edit:

**int** random = *RANDOM*.nextInt(DiceValue.***SPADE***.ordinal() + 1);

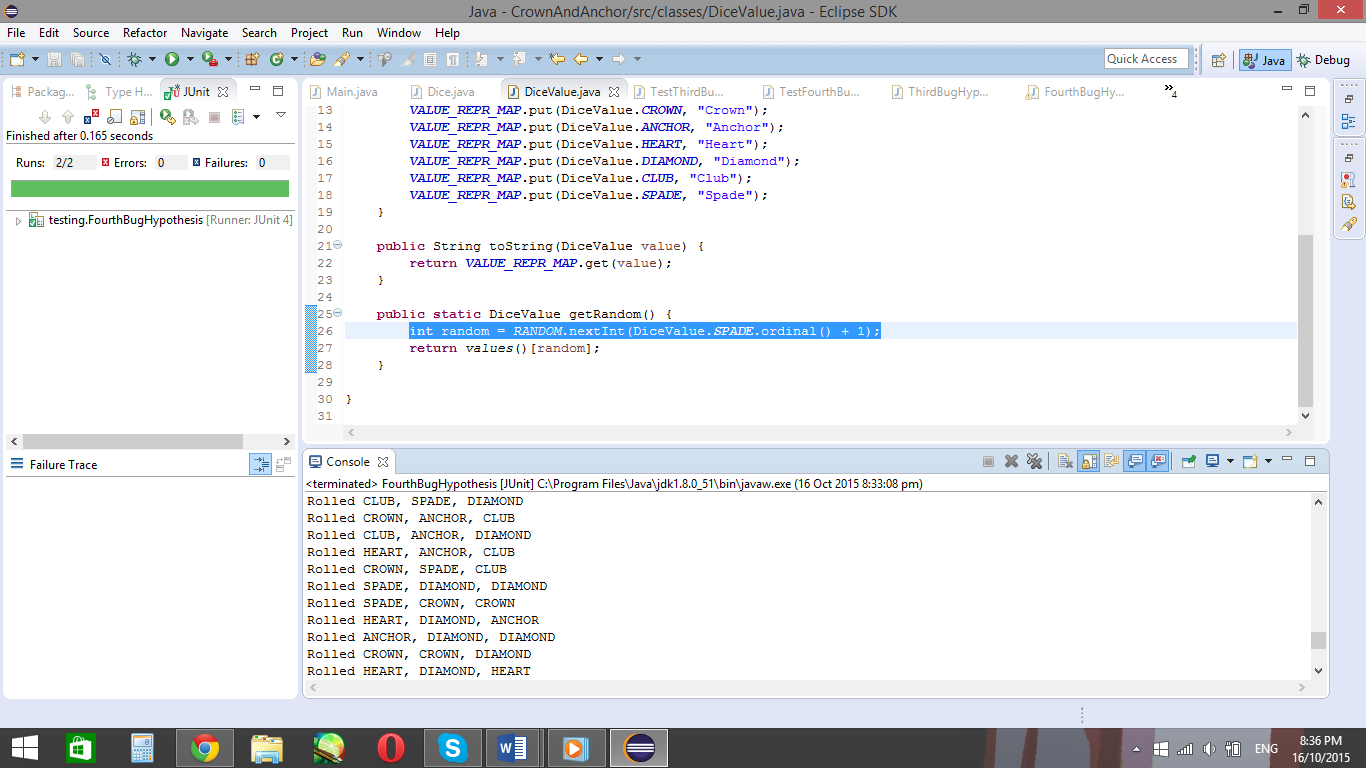
Then, to ensure this worked, rerun the testFourthBug Junit test again to see the new output.

The rerun test:



Shows that the SPADES now appears in the list.

Another version of output:



Displays how the spades appear on the top of the dice.

So now that the spades are now included when the dice are rolled, and the die faces change after each roll, the output shows the bug has been fixed.