CIS573 –Software Engineering

2rd Homework

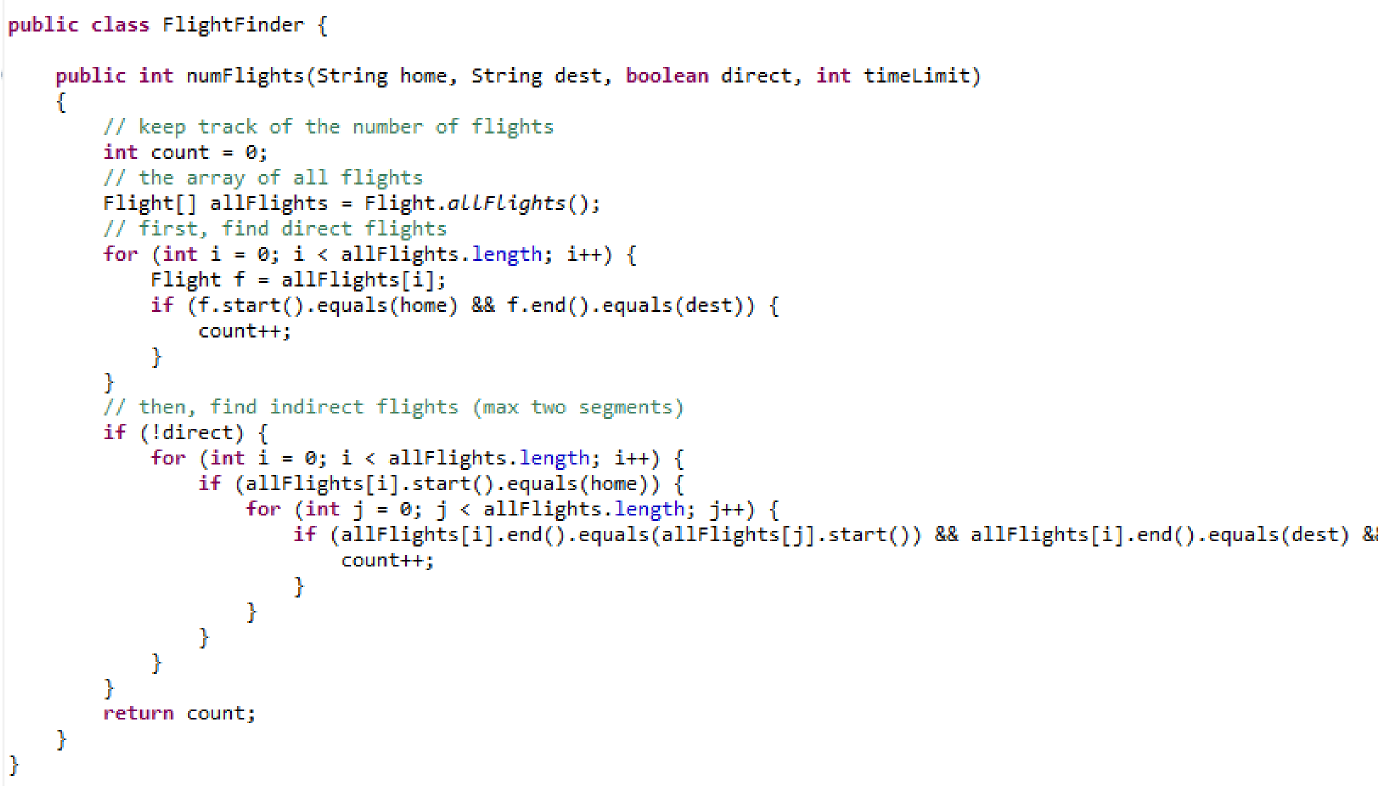
Yayang Tian 24963298

Ning Xu 20139217

Flight Finder

1. Create a Static Slice

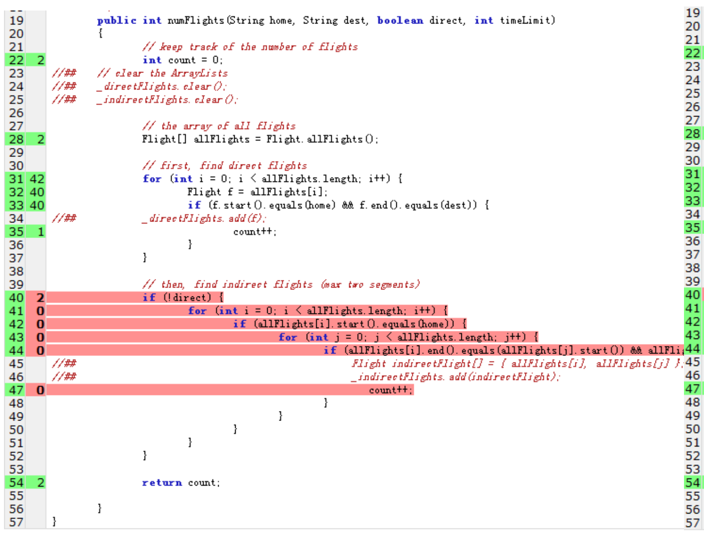
First of all, we generate a static slice code. Line 1, 6, 7, 9, 10, 23, 24, 25, 34, 45, 46 can be removed because they do not affect the return value. That is, we only care about the number of the flights, any sentences relating to available flighting details can be removed. The slice code is shown in Figure 1.



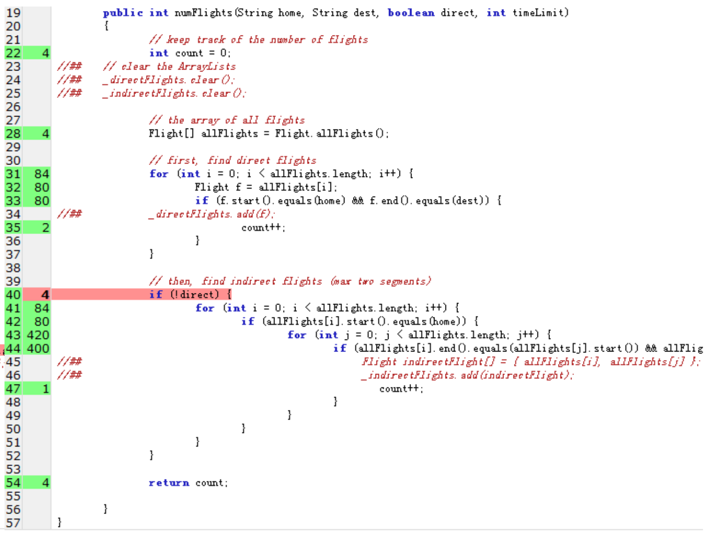
```java
public class FlightFinder {

    public int numFlights(String home, String dest, boolean direct, int timeLimit)
    {
        // keep track of the number of flights
        int count = 0;
        // the array of all flights
        Flight[] allFlights = Flight.allFlights();
        // first, find direct flights
        for (int i = 0; i < allFlights.length; i++) {
            Flight f = allFlights[i];
            if (f.start().equals(home) && f.end().equals(dest)) {
                count++;
            }
        }
        // then, find indirect flights (max two segments)
        if (!direct) {
            for (int i = 0; i < allFlights.length; i++) {
                if (allFlights[i].start().equals(home)) {
                    for (int j = 0; j < allFlights.length; j++) {
                        if (allFlights[i].end().equals(allFlights[j].start()) && allFlights[i].end().equals(dest) &
                            count++;
                        }
                    }
                }
            }
        }
        return count;
    }
}
```

Figure 1

2. Compare Dynamic Slices



```
19         public int numFlights(String home, String dest, boolean direct, int timeLimit)
20         {
21                 // keep track of the number of flights
22  2              int count = 0;
23     //##  // clear the ArrayLists
24     //##  _directFlights.clear();
25     //##  _indirectFlights.clear();
26
27                 // the array of all flights
28  2              Flight[] allFlights = Flight.allFlights();
29
30                 // first, find direct flights
31 42              for (int i = 0; i < allFlights.length; i++) {
32 40                      Flight f = allFlights[i];
33 40                      if (f.start().equals(home) && f.end().equals(dest)) {
34     //##          _directFlights.add(f);
35  1                              count++;
36                         }
37                 }
38
39                 // then, find indirect flights (max two segments)
40  2              if (!direct) {
41  0                      for (int i = 0; i < allFlights.length; i++) {
42  0                              if (allFlights[i].start().equals(home)) {
43  0                                      for (int j = 0; j < allFlights.length; j++) {
44  0                                              if (allFlights[i].end().equals(allFlights[j].start()) && allFli
45     //##                                             Flight indirectFlight[] = { allFlights[i], allFlights[j] };
46     //##                                             _indirectFlights.add(indirectFlight);
47  0                                                      count++;
48                                                 }
49                                         }
50                                 }
51                         }
52                 }
53
54  2              return count;
55
56         }
57 }
```



```
19          public int numFlights(String home, String dest, boolean direct, int timeLimit)
20          {
21                  // keep track of the number of flights
22   4              int count = 0;
23      //##  // clear the ArrayLists
24      //##  _directFlights.clear();
25      //##  _indirectFlights.clear();
26
27                  // the array of all flights
28   4              Flight[] allFlights = Flight.allFlights();
29
30                  // first, find direct flights
31  84              for (int i = 0; i < allFlights.length; i++) {
32  80                      Flight f = allFlights[i];
33  80                      if (f.start().equals(home) && f.end().equals(dest)) {
34      //##          _directFlights.add(f);
35   2                              count++;
36                          }
37                  }
38
39                  // then, find indirect flights (max two segments)
40   4              if (!direct) {
41  84                      for (int i = 0; i < allFlights.length; i++) {
42  80                              if (allFlights[i].start().equals(home)) {
43 420                                      for (int j = 0; j < allFlights.length; j++) {
44 400                                              if (allFlights[i].end().equals(allFlights[j].start()) && allFlig
45      //##                                             Flight indirectFlight[] = { allFlights[i], allFlights[j] };
46      //##                                             _indirectFlights.add(indirectFlight);
47   1                                                      count++;
48                                                  }
49                                          }
50                                  }
51                          }
52                  }
53
54   4              return count;
55
56          }
57  }
```

Figure2

Then, we compare the coverage of dynamic slices for 2 passing test cases and 4 failing cases using Cobertura. As in shown in Figure2, we can clearly see the passing test cases(left) and failing test cases(right) side by side in the same Cobertura report.

From Figure2, for the passing test cases on the left, the branch containing lines 41-47 has never been executed; while for the failing test cases on the right, lines 41-47 has been executed for many times.

So we believe that if there is only one fault, it was in lines 41-47, because only failing test cases will run these lines, while no passing test cases would go through them.

3. Identify Likely Causes

Then, in order to know the reasons for failure, we used Eclipsed debugger to examine the variables step by step. We added back `private ArrayList<Flight[]> _indirectFlights = new ArrayList<Flight[]>(); public ArrayList<Flight[]> indirectFlights() { return _indirectFlights; }, Flight indirectFlight[] = { allFlights[i], allFlights[j] };` and `_indirectFlights.add(indirectFlight);` and breakpoints on line 41-44, assuming that there was only one fault. We observed that for failing test cases, the elements in `Flight indirectFlight[]` is not continuous, which cannot be an indirect flight. We speculated that the index of judgement sentence was faulty.

4. Fix the Bug

To fix the bug, we speculated that in line 44, index of allFlight[i].end().equals(dest) j, which is the conjugation of two elements of indirect flight should be the same as the last one, namely, j. So we changed it to j.

After running the test suit again, all tests passed, indicating that the bug had been fixed.

In retrospect, we fixed the bug in line 44, which greatly matched to our fault localization result between lines 41-47. After having gone through Step 3 and then fixing the bugs in Step 4, it seemed to be very accurate about the ranking of the likelihood of being faulty in Step 2.

5. Regression testing

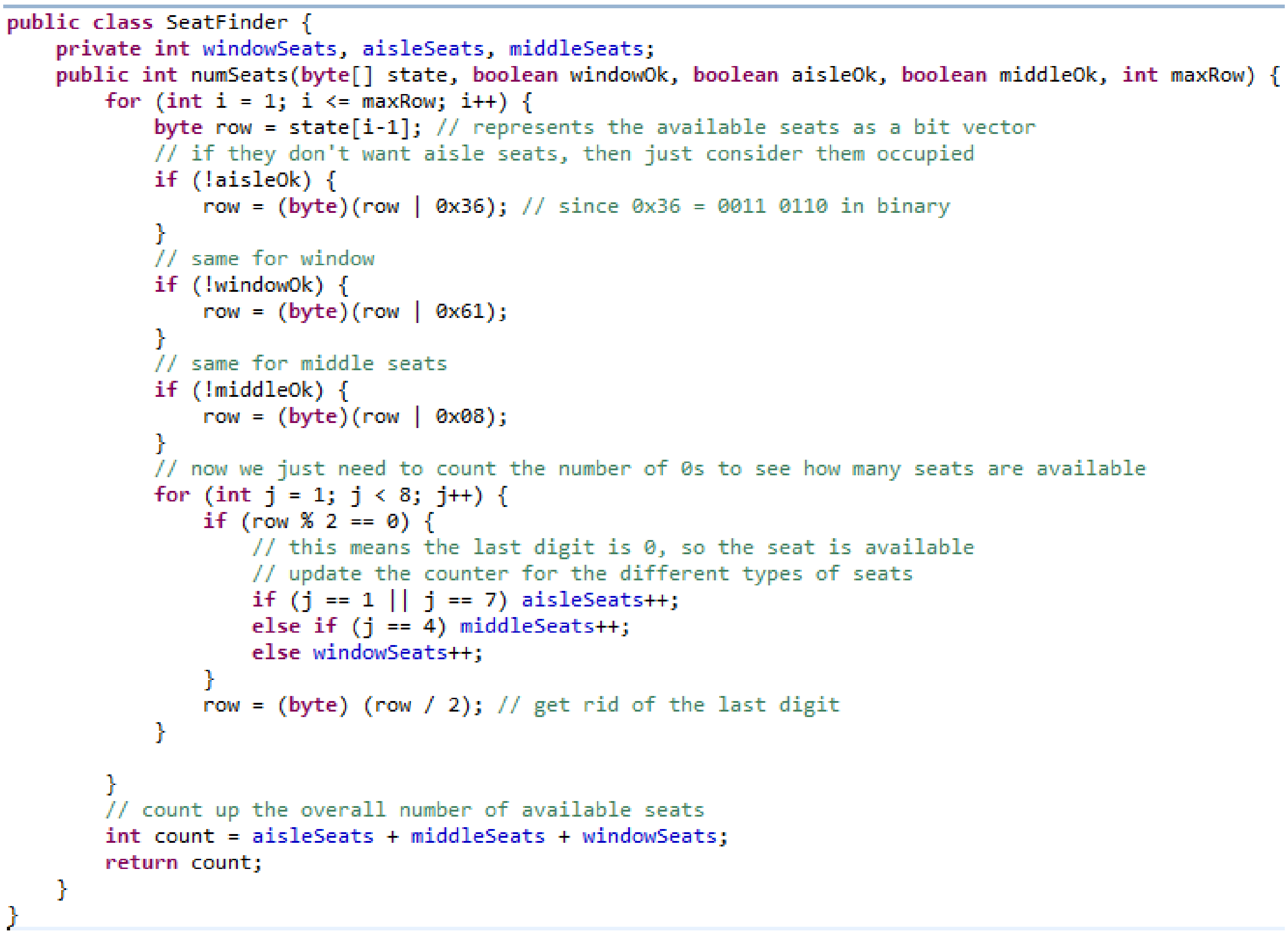
Finally, we conducted regression testing to make sure that the change to the airline software system did not introduce new errors in the unchanged part of the program. That is, after we had made sure our sliced code was passing all the tests, we modified the original code with the same fix.

We ran all the unit tests again, and they all passed. So we successfully fixed the bug.

Seat Finder

1. Create a Static Slice

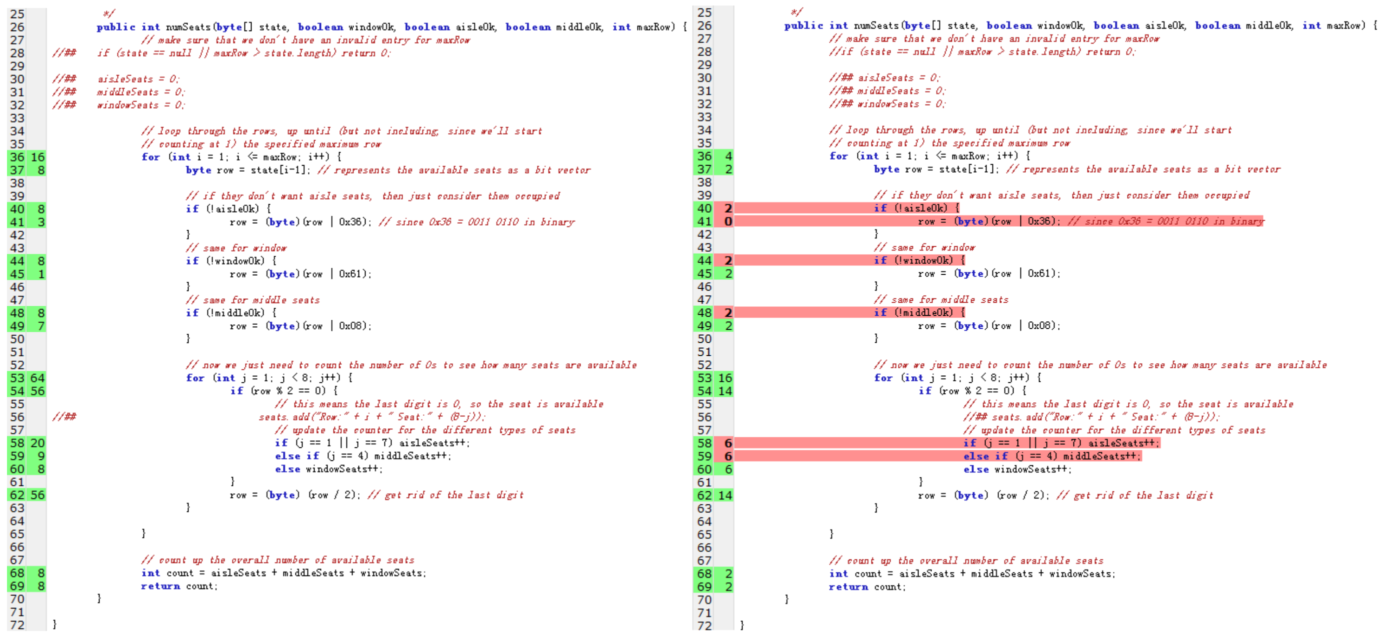
First of all, we generate a static slice code. Line 7,28,30,31,32,56 can be removed because they do not affect the return value. That is, we only care about the number of the available seats, any sentences relating to available seats details can be removed. Besides, the initialization process can be removed, because the default value is 0. The slice code is shown in Figure 3.



```java
public class SeatFinder {
    private int windowSeats, aisleSeats, middleSeats;
    public int numSeats(byte[] state, boolean windowOk, boolean aisleOk, boolean middleOk, int maxRow) {
        for (int i = 1; i <= maxRow; i++) {
            byte row = state[i-1]; // represents the available seats as a bit vector
            // if they don't want aisle seats, then just consider them occupied
            if (!aisleOk) {
                row = (byte)(row | 0x36); // since 0x36 = 0011 0110 in binary
            }
            // same for window
            if (!windowOk) {
                row = (byte)(row | 0x61);
            }
            // same for middle seats
            if (!middleOk) {
                row = (byte)(row | 0x08);
            }
            // now we just need to count the number of 0s to see how many seats are available
            for (int j = 1; j < 8; j++) {
                if (row % 2 == 0) {
                    // this means the last digit is 0, so the seat is available
                    // update the counter for the different types of seats
                    if (j == 1 || j == 7) aisleSeats++;
                    else if (j == 4) middleSeats++;
                    else windowSeats++;
                }
                row = (byte) (row / 2); // get rid of the last digit
            }

        }
        // count up the overall number of available seats
        int count = aisleSeats + middleSeats + windowSeats;
        return count;
    }
}
```

Figure 3

2. Compare Dynamic Slices



```
25            */
26            public int numSeats(byte[] state, boolean windowOk, boolean aisleOk, boolean middleOk, int maxRow) {
27                    // make sure that we don't have an invalid entry for maxRow
28      //##    if (state == null || maxRow > state.length) return 0;
29
30      //##    aisleSeats = 0;
31      //##    middleSeats = 0;
32      //##    windowSeats = 0;
33
34                    // loop through the rows, up until (but not including, since we'll start
35                    // counting at 1) the specified maximum row
36 16                 for (int i = 1; i <= maxRow; i++) {
37  8                         byte row = state[i-1]; // represents the available seats as a bit vector
38
39                            // if they don't want aisle seats, then just consider them occupied
40  8                         if (!aisleOk) {
41  3                                 row = (byte)(row | 0x36); // since 0x36 = 0011 0110 in binary
42                            }
43                            // same for window
44  8                         if (!windowOk) {
45  1                                 row = (byte)(row | 0x61);
46                            }
47                            // same for middle seats
48  8                         if (!middleOk) {
49  7                                 row = (byte)(row | 0x08);
50                            }
51
52                            // now we just need to count the number of 0s to see how many seats are available
53 64                         for (int j = 1; j < 8; j++) {
54 56                                 if (row % 2 == 0) {
55                                            // this means the last digit is 0, so the seat is available
56      //##                                seats.add("Row:" + i + " Seat:" + (8-j));
57                                            // update the counter for the different types of seats
58 20                                         if (j == 1 || j == 7) aisleSeats++;
59  9                                         else if (j == 4) middleSeats++;
60  8                                         else windowSeats++;
61                                    }
62 56                                 row = (byte) (row / 2); // get rid of the last digit
63                            }
64
65                    }
66
67                    // count up the overall number of available seats
68  8                 int count = aisleSeats + middleSeats + windowSeats;
69  8                 return count;
70            }
71
72      }
```

```
25            */
26            public int numSeats(byte[] state, boolean windowOk, boolean aisleOk, boolean middleOk, int maxRow) {
27                    // make sure that we don't have an invalid entry for maxRow
28                    //if (state == null || maxRow > state.length) return 0;
29
30                    //## aisleSeats = 0;
31                    //## middleSeats = 0;
32                    //## windowSeats = 0;
33
34                    // loop through the rows, up until (but not including, since we'll start
35                    // counting at 1) the specified maximum row
36  4                 for (int i = 1; i <= maxRow; i++) {
37  2                         byte row = state[i-1]; // represents the available seats as a bit vector
38
39                            // if they don't want aisle seats, then just consider them occupied
40  2                         if (!aisleOk) {
41  0                                 row = (byte)(row | 0x36); // since 0x36 = 0011 0110 in binary
42                            }
43                            // same for window
44  2                         if (!windowOk) {
45  2                                 row = (byte)(row | 0x61);
46                            }
47                            // same for middle seats
48  2                         if (!middleOk) {
49  2                                 row = (byte)(row | 0x08);
50                            }
51
52                            // now we just need to count the number of 0s to see how many seats are available
53 16                         for (int j = 1; j < 8; j++) {
54 14                                 if (row % 2 == 0) {
55                                            // this means the last digit is 0, so the seat is available
56                                            //## seats.add("Row:" + i + " Seat:" + (8-j));
57                                            // update the counter for the different types of seats
58  6                                         if (j == 1 || j == 7) aisleSeats++;
59  6                                         else if (j == 4) middleSeats++;
60  6                                         else windowSeats++;
61                                    }
62 14                                 row = (byte) (row / 2); // get rid of the last digit
63                            }
64
65                    }
66
67                    // count up the overall number of available seats
68  2                 int count = aisleSeats + middleSeats + windowSeats;
69  2                 return count;
70            }
71
72      }
```

Figure4

Then, we compare the coverage of dynamic slices for 8 passing test cases and 2 failing cases using Cobertura. As in shown in Figure4, we can clearly see the passing test cases(left) and failing test cases(right) side by side in the same Cobertura report.

From Figure4, line 45 is the only line for the failing test cases running more than the passing one, while for other lines the time run by failing test cases are far less passing ones. So it is most likely that there is a bug in line 45.

So we believe that if there is only one fault, it was in lines 45, because only failing test cases will run these lines more often, while less passing test cases would go through them.

3. Identify Likely Causes

Then, in order to know the reasons for failure, we used Eclipsed debugger to examine the variables step by step. We started on line 45. We observed that the value of row was not changed as we had expected after line 45. It not only assigned the second and the last bit one, but also assigned the third bit of row to be one.

4. Fix the Bug

To fix the bug, we speculated that in line 45, To fix the bug, we calculated 0100 0001 to be 0*41 in the form of HEX. Then in the code, we changed 0*61 to 0*41. In retrospect, we fixed the bug in line 45, which greatly matched to our fault localization result between lines 45. After having gone through Step 3 and then fixing the bugs in Step 4, it seemed to be very accurate about the ranking of the likelihood of being faulty in Step 2.

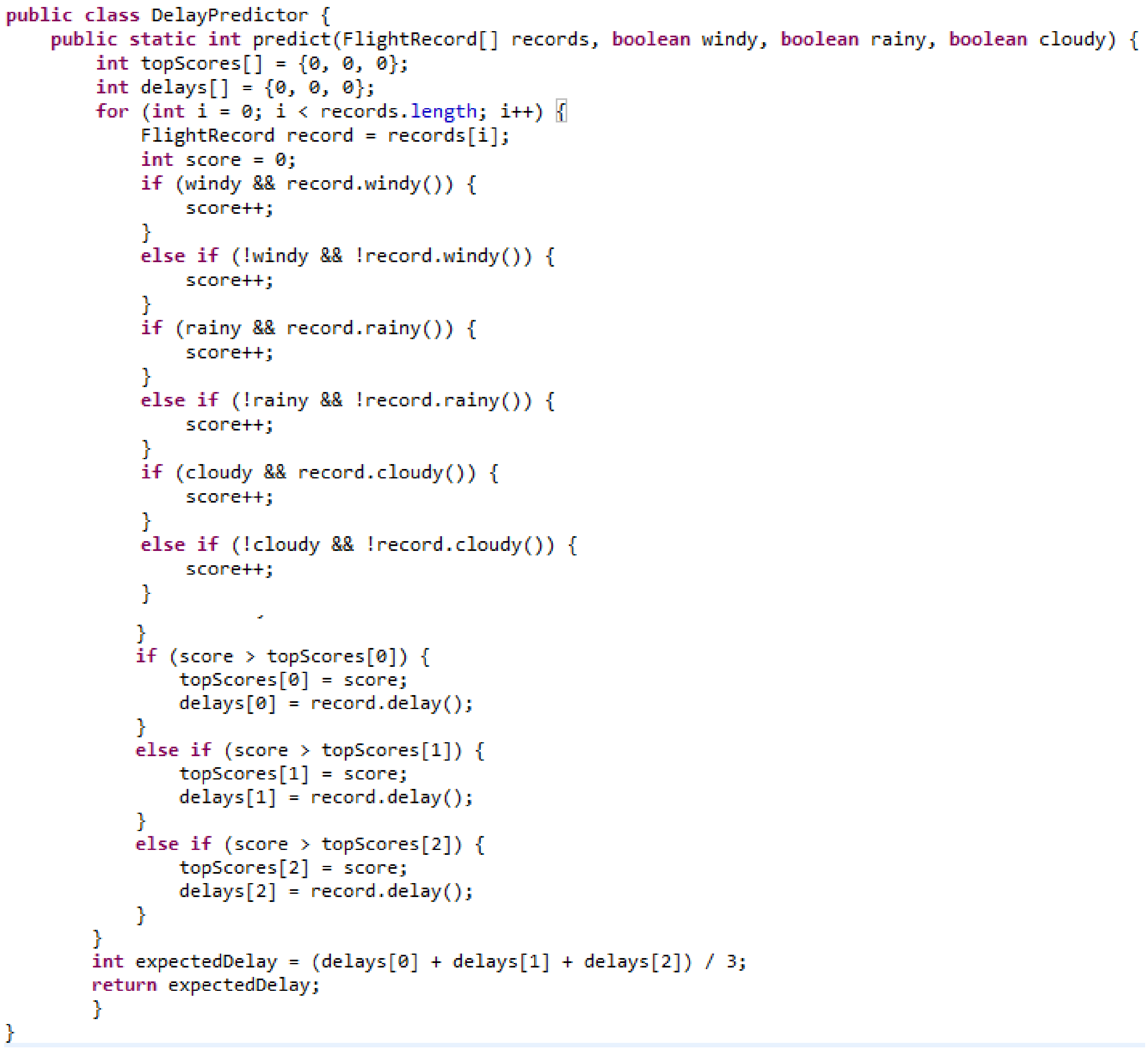
5. Regression testing

Finally, we conducted regression testing to make sure that the change to the airline software system did not introduce new errors in the unchanged part of the program. That is, after we had made sure our sliced code was passing all the tests, we modified the original code with the same fix.
We ran all the unit tests again, and they all passed. So we successfully fixed the bug.

DelayPredictor

1. Create a Static Slice

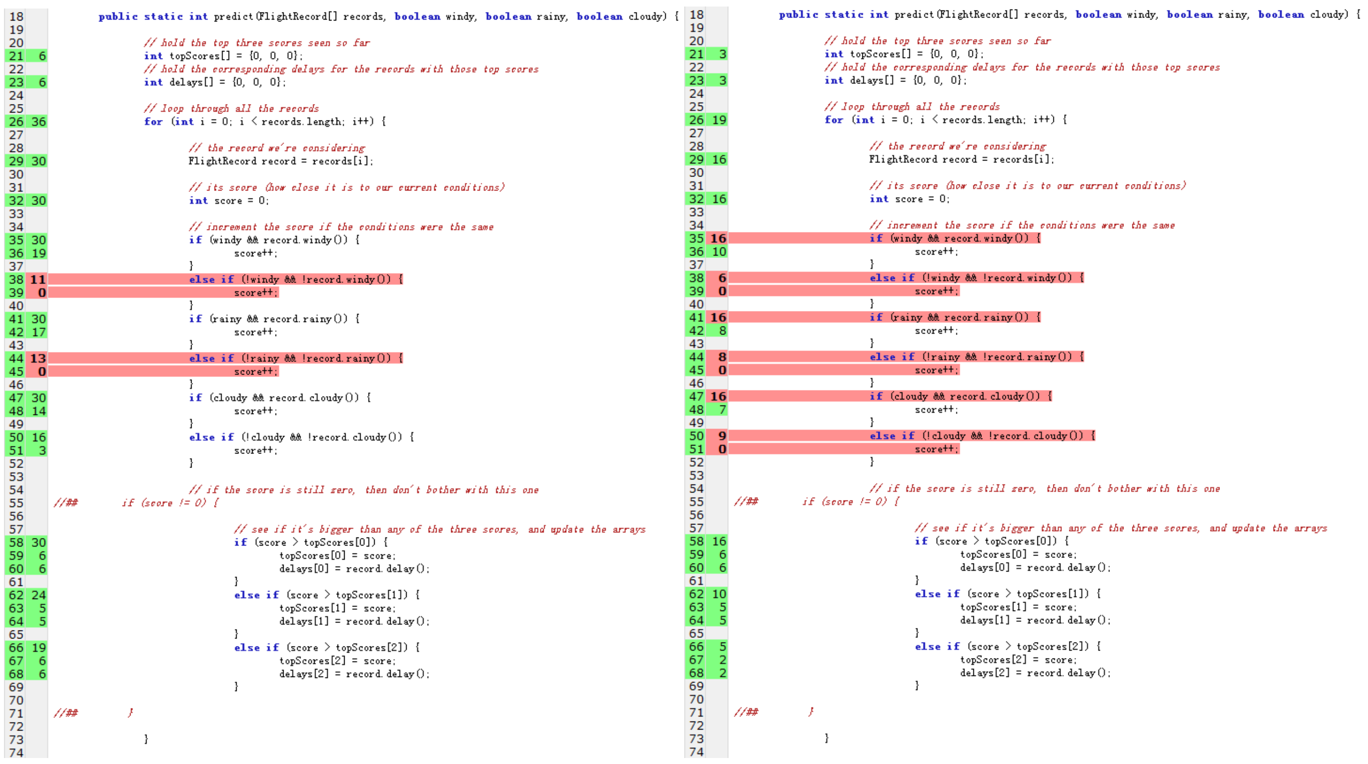
First of all, we generate a static slice code. Line 55, 71 can be removed because they do not affect the return value. That is, the judgement sentence if(score !=0) is of no use because the following sentences involve it. So it can be removed. The slice code is shown in Figure 5.



```java
public class DelayPredictor {
    public static int predict(FlightRecord[] records, boolean windy, boolean rainy, boolean cloudy) {
        int topScores[] = {0, 0, 0};
        int delays[] = {0, 0, 0};
        for (int i = 0; i < records.length; i++) {
            FlightRecord record = records[i];
            int score = 0;
            if (windy && record.windy()) {
                score++;
            }
            else if (!windy && !record.windy()) {
                score++;
            }
            if (rainy && record.rainy()) {
                score++;
            }
            else if (!rainy && !record.rainy()) {
                score++;
            }
            if (cloudy && record.cloudy()) {
                score++;
            }
            else if (!cloudy && !record.cloudy()) {
                score++;
            }

           }
           if (score > topScores[0]) {
               topScores[0] = score;
               delays[0] = record.delay();
           }
           else if (score > topScores[1]) {
               topScores[1] = score;
               delays[1] = record.delay();
           }
           else if (score > topScores[2]) {
               topScores[2] = score;
               delays[2] = record.delay();
           }
       }
       int expectedDelay = (delays[0] + delays[1] + delays[2]) / 3;
       return expectedDelay;
       }
}
```

Figure 5

2. Compare Dynamic Slices



```
18          public static int predict(FlightRecord[] records, boolean windy, boolean rainy, boolean cloudy) {
19
20                  // hold the top three scores seen so far
21  6               int topScores[] = {0, 0, 0};
22                  // hold the corresponding delays for the records with those top scores
23  6               int delays[] = {0, 0, 0};
24
25                  // loop through all the records
26 36               for (int i = 0; i < records.length; i++) {
27
28                          // the record we're considering
29 30                       FlightRecord record = records[i];
30
31                          // its score (how close it is to our current conditions)
32 30                       int score = 0;
33
34                          // increment the score if the conditions were the same
35 30                       if (windy && record.windy()) {
36 19                               score++;
37                          }
38 11                       else if (!windy && !record.windy()) {
39  0                               score++;
40                          }
41 30                       if (rainy && record.rainy()) {
42 17                               score++;
43                          }
44 13                       else if (!rainy && !record.rainy()) {
45  0                               score++;
46                          }
47 30                       if (cloudy && record.cloudy()) {
48 14                               score++;
49                          }
50 16                       else if (!cloudy && !record.cloudy()) {
51  3                               score++;
52                          }
53
54                          // if the score is still zero, then don't bother with this one
55      //##        if (score != 0) {
56
57                                  // see if it's bigger than any of the three scores, and update the arrays
58 30                               if (score > topScores[0]) {
59  6                                       topScores[0] = score;
60  6                                       delays[0] = record.delay();
61                                  }
62 24                               else if (score > topScores[1]) {
63  5                                       topScores[1] = score;
64  5                                       delays[1] = record.delay();
65                                  }
66 19                               else if (score > topScores[2]) {
67  6                                       topScores[2] = score;
68  6                                       delays[2] = record.delay();
69                                  }
70
71      //##         }
72
73                  }
74
```

```
18          public static int predict(FlightRecord[] records, boolean windy, boolean rainy, boolean cloudy) {
19
20                  // hold the top three scores seen so far
21  3               int topScores[] = {0, 0, 0};
22                  // hold the corresponding delays for the records with those top scores
23  3               int delays[] = {0, 0, 0};
24
25                  // loop through all the records
26 19               for (int i = 0; i < records.length; i++) {
27
28                          // the record we're considering
29 16                       FlightRecord record = records[i];
30
31                          // its score (how close it is to our current conditions)
32 16                       int score = 0;
33
34                          // increment the score if the conditions were the same
35 16                       if (windy && record.windy()) {
36 10                               score++;
37                          }
38  6                       else if (!windy && !record.windy()) {
39  0                               score++;
40                          }
41 16                       if (rainy && record.rainy()) {
42  8                               score++;
43                          }
44  8                       else if (!rainy && !record.rainy()) {
45  0                               score++;
46                          }
47 16                       if (cloudy && record.cloudy()) {
48  7                               score++;
49                          }
50  9                       else if (!cloudy && !record.cloudy()) {
51  0                               score++;
52                          }
53
54                          // if the score is still zero, then don't bother with this one
55      //##        if (score != 0) {
56
57                                  // see if it's bigger than any of the three scores, and update the arrays
58 16                               if (score > topScores[0]) {
59  6                                       topScores[0] = score;
60  6                                       delays[0] = record.delay();
61                                  }
62 10                               else if (score > topScores[1]) {
63  5                                       topScores[1] = score;
64  5                                       delays[1] = record.delay();
65                                  }
66  5                               else if (score > topScores[2]) {
67  2                                       topScores[2] = score;
68  2                                       delays[2] = record.delay();
69                                  }
70
71      //##         }
72
73                  }
74
```

Figure6

Then, we compare the coverage of dynamic slices for 6 passing test cases and 3 failing cases using Cobertura. As in shown in Figure6, we can clearly see the passing test cases(left) and failing test cases(right) side by side in the same Cobertura report.

From Figure6, there are not many unusual lines which would indicate a bug. As there were six passing test cases and three failed test cases, most of the lines in DelayPredictor had been run twice as much as those in DelayPredictor2. While the lines of 58,59,62,63 have been run the same times on both the failed test cases and passing test cases, this is somewhat abnormal compared with other lines of codes.

3. Identify Likely Causes

Then, in order to know the reasons for failure, we used Eclipsed debugger to examine the variables step by step. We debugged line by line, and we found that after running through the lines from 58 to 68, the final topScores[] and delays[] were not what we had expected when running test cases 3, 6 and 8. So I think the process of generating topScores[] and delays[] is wrong.

4. Fix the Bug

To fix the bug, We tried to fix the bug by adding and deleting lines while at the same time keeping some of the original codes. However, we failed in the trial as the algorithm to generate the topScores[] is wrong which just simply replaced the first smaller score. Then I have to rewrite the codes from line 58 to line 68 which has indicates the smallest score and index in the list and then compare the score with the smallest score of the list. You can see the details in the fixed codes.
After running the test suite again, there is still a test case failed, test case 3, which means there are still bugs in the codes. As the codes are short and I have already debugged part of the codes, then I decided to continue debugging line by line. Things are all good until I debugged to the second last line where I

expect int expectedDelay to be 20 instead of 13. As the last line is a return, this line must have a bug. To fix this bug, I rewrite several lines of codes considering whether there are zeros in topScore[]. If there are there zeros in topScores[], we return expectedDelay as zero. And we only calculate the average delays of the non-zero value in topScores[]. After running the test suit again, all tests passed, indicating that the bug had been fixed.
In retrospect, we fixed the bug, which greatly matched to our fault localization result. After having gone through Step 3 and then fixing the bugs in Step 4, it seemed to be very accurate about the ranking of the likelihood of being faulty in Step 2.

5. Regression testing

Finally, we conducted regression testing to make sure that the change to the airline software system did not introduce new errors in the unchanged part of the program. That is, after we had made sure our sliced code was passing all the tests, we modified the original code with the same fix.
We ran all the unit tests again, and they all passed. So we successfully fixed the bug.