EECS 4313E Assignment 2

Black-box and White-box Testing with JUnit

November 14th, 2021

Muhammad Khan 216850158
Pranav Prashar - 214986962
Vamsi Lingamaneni - 214724058

Black-box Testing with JUnit **3**
1.1 Specification of the selected Java methods 3
1.2 Justification of the testing technique 3
1.2.1 testMinuteStringBVT() - Boundary Values Testing 3
1.2.2 equivalenceStringECT() - Equivalence Class Testing 3
1.2.3 testIsAfter() - Decision Table Testing 3
1.2.4 Boundary Value Testing Justification 4
1.2.5 Equivalence Class Testing Justification 4
1.2.6 Decision Table Testing Justification 4
1.3 Evaluation of the test cases 6
1.3.1 Boundary Values Test Cases Evaluation 6
Boundary Values Testing run results 6
1.3.2 Equivalence Class Testing Evaluation 7
Equivalence Class Testing run results 8
1.3.3 Decision Table Testing Evaluation 8
Decision Table Testing run results 9
Case 1 9
Case 2 9
Case 3 10
1.4 Bug reports 10
White-Box Testing with JUnit **11**
2.1 White Box Description Of Tests 11
2.1.1 Boundary Value Testing 11
Code coverage before whitebox testing: 11
Statement coverage before whitebox testing: 11
Added test cases 11
Explanation of the added test cases 11
Code coverage after white box testing 12
Boundary value testing run results 12
2.1.2 Equivalence Class Testing Added Test Cases 13
Code coverage before white-box testing: 13
Statement coverage before white-box testing: 13
Added test cases 13
Explanation of the added test cases 14
Code coverage after white box testing 14
Statement coverage after white-box testing 14
Equivalence class testing run results 14
2.1.3 Decision Table Testing Added Test Cases 15
Code coverage before white-box testing: 15
Statement coverage before white-box testing: 15
Added test cases 15
Decision table testing run results 16
2.1.4 Control flow graph 16
2.2 Bug reports 17

1. Black-box Testing with JUnit

1.1 Specification of the selected Java methods

To test the borg calendar, three testing methods were created. The first method is testMinuteStringBVT(). This method tests boundary values, testing the values which were at the minimum; in addition to, just above the minimum, nominal, just below the maximum, and maximum (max). The second test method is testMinuteStringECT() and it is testing the class minuteString. Using Equivalence class testing we are looking for equality between two results and checking to see if they are True or False. The third method, testIsAfter(), for which we tested the method isAfter(). Through which we tested using boundary values testing.

1.2 Justification of the testing technique

To test the Borg Calendar, the use of three testing techniques was used. We used Boundary value testing, Equivalence class testing, and Decision table testing. Each of these testing techniques allows us to cover different aspects of the program and allows us to ensure consistency in features.

1.2.1 testMinuteStringBVT() - Boundary Values Testing

The test method testMinuteStringBVT() was used to test the public static String minuteString(int mins). The minuteString() method takes an integer input of minutes and converts it into a String representation in minutes and hours. Depending on the Integer input the method will output a singular or plural string representation of the time. An example output could be 35791394 Hours 7 Minutes which would be the string representation of 2147483647 minutes.

1.2.2 equivalenceStringECT() - Equivalence Class Testing

For equivalence class, testing was done on the minuteString() method, as mentioned above the method takes an integer input of minutes and converts it into a String representation in minutes and hours. Equivalence class testing was done to test the equivalence of the String output of the minuteString method and check to see if the string is valid when compared to the output of a minuteString.

1.2.3 testIsAfter() - Decision Table Testing

For decision table testing, testing was done on the isAfter(Date d1, Date d2) method. This method allows us to check if the d1 of Date type occurs after d2 of Date type. Using Decision table testing, we checked to see whether a date is before, after, and equal to another date. The isAfter() method would return a true or false boolean value, based on whether the value of d1 is after the value of d2. If d1 is after d2, the method returns true, another way around, or if it’s the same date, it returns false.

1.2.4 Boundary Value Testing Justification

Boundary value testing was used to test various ranges of inputs of the program, this testing method was used to test the minuteString() method. For Boundary value testing, we are looking to test Minimum values, Just above the minimum, Nominal, Just below the maximum, Maximum. The reason that boundary value testing is used in this citation is that for the minuteString method there is an upper limit of time which the method can convert. A constraint of the method is that Java only allows a maximum value of 2147483647, using boundary value testing we ensure that the program can reach the maximum value. The method also has a minimum value, as the method is being used to convert an integer value into a String value and the program is dealing with a value of time. We should not be able to have a negative time value, thus we are testing at the lower limit of 0 times. To ensure that we cannot only get the maximum value and minimum value, but we also test at 0, 1, 2147483647/2, 2147483646, 2147483647. Through testing the limits and just below and above the limits, we can ensure that the numbers between the min and max would also work.

1.2.5 Equivalence Class Testing Justification

Through equivalence class testing, we can divide the input into partitions of equivalent data. This is why this testing method is suitable for the minuteString() method. This is because the API for software says that if the input of the minuteString() method is greater than 60 minutes. Then the string output will also have “x” amount of hours and “y” amount of minutes. Essentially, the input is divided into bits and pieces of what is similar to the input and here the same applies. An example of this would be, if the input is 123 minutes, the program would check if the input is greater than 60 and less than max. If it’s greater than 60, then it would divide the input by 60 as well as take mod of the input by 60 to find the reminder. That reminder would be the minutes and the quotient would be the hours. So here it would be, 2 hours is the quotient, and 3 is the reminder. For 123 input in the minuteString() method, the output would be 2 hours 3 minutes. The test cases we have created for this testing method are weak normal robust testing. We have 6 different test cases which touch upon dividing the input into partitions of equivalent data, hence why these test cases are included in the equivalence class testing class.

Through the Equivalence class testing, no abnormal behavior was detected. The output results of test cases matched the expected values. For the Decision table testing case, we did not encounter any special value testing which was needed.

1.2.6 Decision Table Testing Justification

Decision table testing is a “software testing methodology used to test system behavior for various input combinations” - https://www.upgrad.com/blog/decision-table-testing/. This testing method was used to test, isAfter(Date d1, Date d2). The isAfter() method would return a true or false boolean value, based on whether the value of d1 is after the value of d2. If d1 is after d2, the method returns true and if d1 is before or equal to d2, then the method returns false.

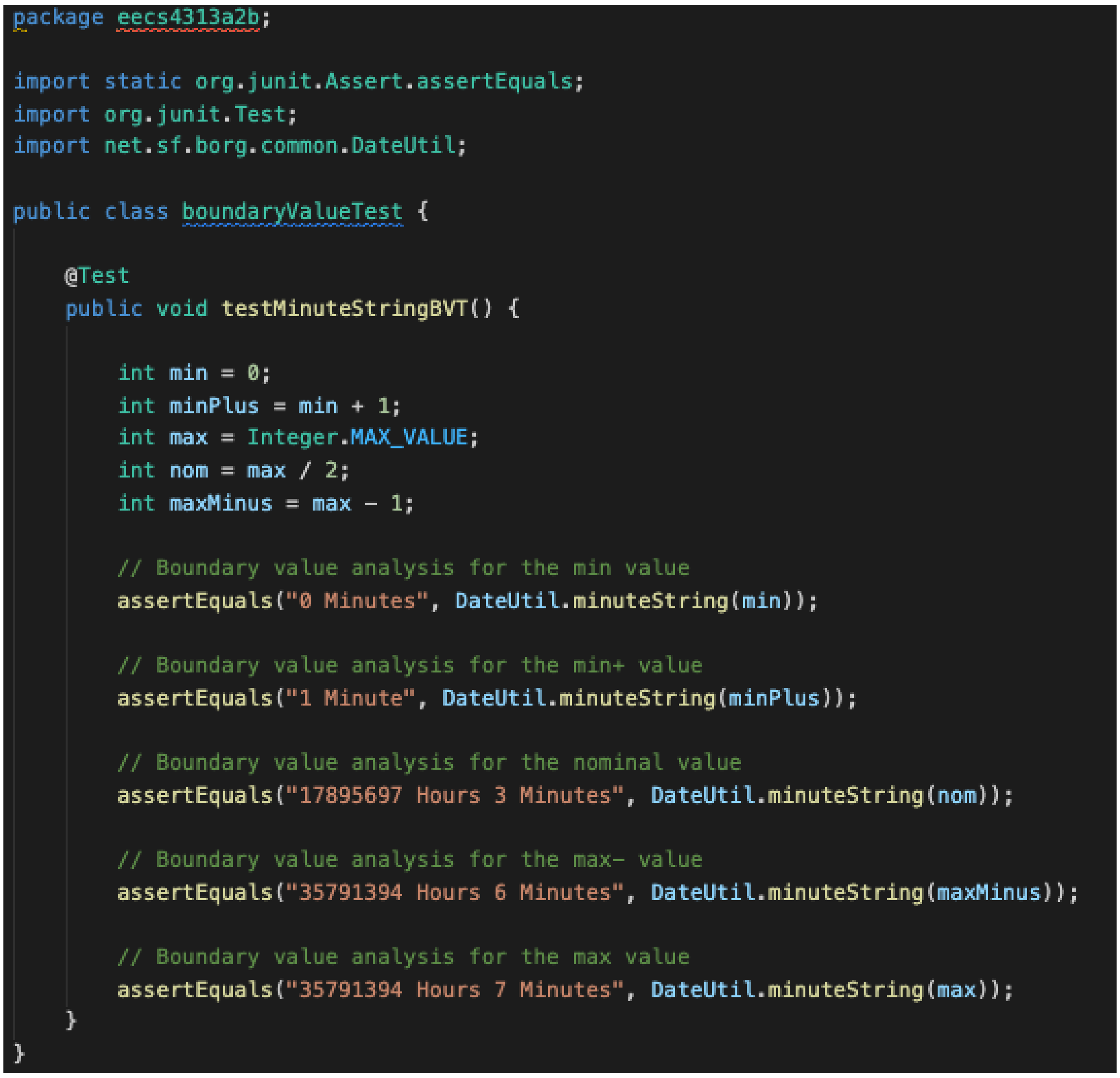
	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Case 1: d1 < d2	T	T	F	T	T	F	F	F
Case 2: d1 > d2	F	T	T	F	T	F	T	F
Case 3: d1 = d2	T	-	T	F	-	T	F	F
Action 1: Date is after							X	
Action 2: Date isn't after				X		X		
Action 3: Impossible	X	X	X		X			X

For the isAfter() method three possible outcomes are possible; the value of d1 < d2, value of d1 > d2 and lastly value of d1 = d2. The method isAfter() outputs the result as a boolean value of either true or false. Knowing that the possible outcomes are; the value of d1 < d2, value of d1 > d2 and lastly value of d1 = d2. We can say that there are $2^3 = 8$possible outcomes. From the above table, we can see that there are 5 rules which would not be possible outcomes. For example rule 1, we can see that if case 1 is true and case 3 is true, this means that d1 is less than d2 and equal to d2 which is impossible. But rule 4, 6 and 7 are cases where d1 < d2 is true, d1 = d2 is true and lastly d1 > d2 is true respectively. For rules 2 and 5, we do not care if d1 = d2 is true or false because we already know that it is impossible since d1 > d2 and d1 < d2 is true which cannot be possible as a date cannot be before and after each other at the same time.

For the Junit testing class testIsAfter(), we are testing to see if the values of 2 dates fit the possible cases where d1 < d2, d1 == d2 and lastly d1 > d2. This can be seen below in our code snippet in the Evaluation of the test cases.

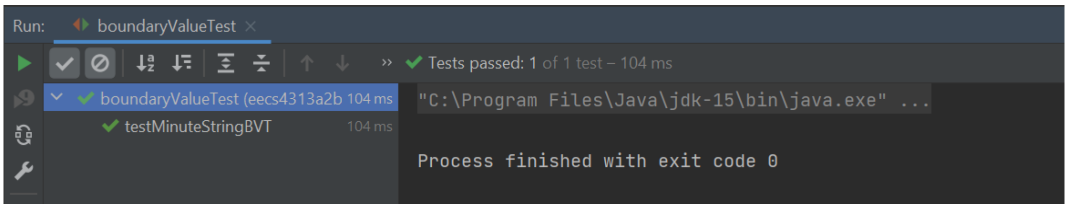
1.3 Evaluation of the test cases

1.3.1 Boundary Values Test Cases Evaluation



```java
package eecs4313a2b;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import net.sf.borg.common.DateUtil;

public class boundaryValueTest {

    @Test
    public void testMinuteStringBVT() {

        int min = 0;
        int minPlus = min + 1;
        int max = Integer.MAX_VALUE;
        int nom = max / 2;
        int maxMinus = max - 1;

        // Boundary value analysis for the min value
        assertEquals("0 Minutes", DateUtil.minuteString(min));

        // Boundary value analysis for the min+ value
        assertEquals("1 Minute", DateUtil.minuteString(minPlus));

        // Boundary value analysis for the nominal value
        assertEquals("17895697 Hours 3 Minutes", DateUtil.minuteString(nom));

        // Boundary value analysis for the max- value
        assertEquals("35791394 Hours 6 Minutes", DateUtil.minuteString(maxMinus));

        // Boundary value analysis for the max value
        assertEquals("35791394 Hours 7 Minutes", DateUtil.minuteString(max));
    }
}
```

Boundary Values Testing run results



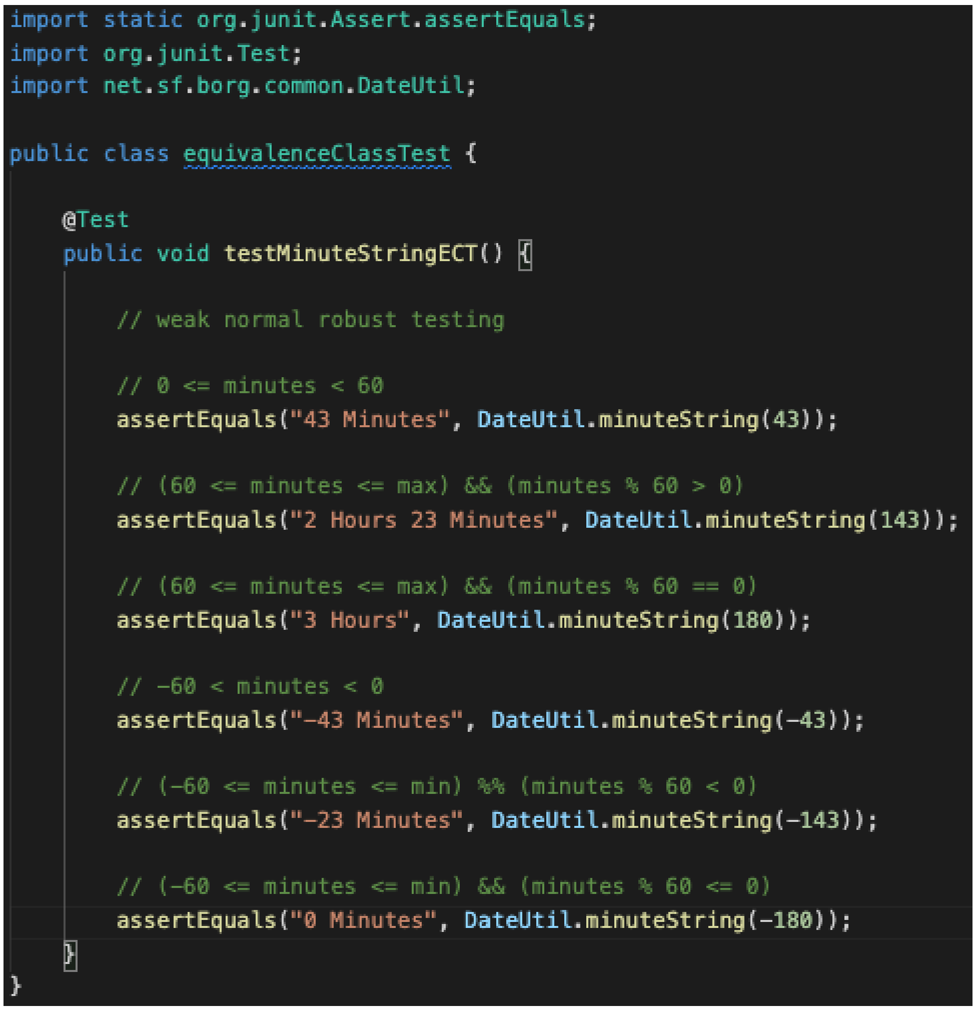
```
Tests passed: 1 of 1 test – 104 ms
boundaryValueTest (eecs4313a2b 104 ms
    testMinuteStringBVT    104 ms

"C:\Program Files\Java\jdk-15\bin\java.exe" ...

Process finished with exit code 0
```

In the above test case, we are testing the various boundary conditions. As we are dealing with hours and minutes, we have decided to start with a minimum value of 0. Through boundary values testing, it should cover the minimum; in addition to, just above the minimum, nominal, just below the maximum, and maximum (max). Above the minimum value is 0, with minPlus being just above the minimum value. Next up we have the max value, in Java, the max value can be found using

Integer.MAX_VALUE which gives us a value of 2147483647. After that, we are testing the value of nominal or middle value, in our case we have chosen to check the value of the maximum value divided by 2 giving us the middle value or nominal value. And lastly, we have the maxMinus, which is the maximum value -1 giving us a value of 2147483646. By doing this we are testing. For the first test case we are putting the minimum value of 0, and invoking the method minuteString() to convert the integer value to a String value. Through testing the maximum values and minimum values, it ensures the values in between would also be valid and working.

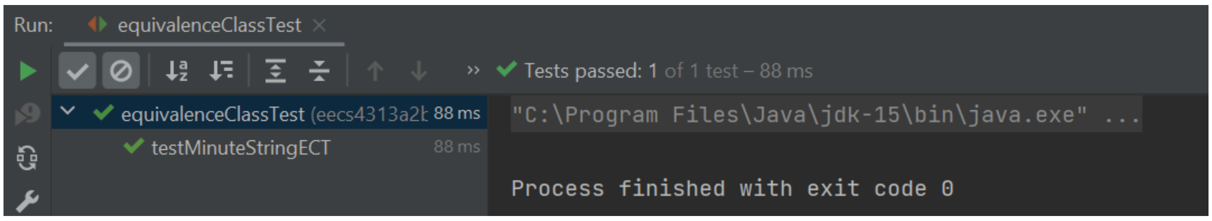
Through the JUnit Testing, no abnormal behavior was detected. The output results of test cases matched the expected values. For the Boundary Case value testing, we did not encounter any special value testing which was needed.

1.3.2 Equivalence Class Testing Evaluation



```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import net.sf.borg.common.DateUtil;

public class equivalenceClassTest {

    @Test
    public void testMinuteStringECT() {

        // weak normal robust testing

        // 0 <= minutes < 60
        assertEquals("43 Minutes", DateUtil.minuteString(43));

        // (60 <= minutes <= max) && (minutes % 60 > 0)
        assertEquals("2 Hours 23 Minutes", DateUtil.minuteString(143));

        // (60 <= minutes <= max) && (minutes % 60 == 0)
        assertEquals("3 Hours", DateUtil.minuteString(180));

        // -60 < minutes < 0
        assertEquals("-43 Minutes", DateUtil.minuteString(-43));

        // (-60 <= minutes <= min) %% (minutes % 60 < 0)
        assertEquals("-23 Minutes", DateUtil.minuteString(-143));

        // (-60 <= minutes <= min) && (minutes % 60 <= 0)
        assertEquals("0 Minutes", DateUtil.minuteString(-180));
    }
}
```

Equivalence Class Testing run results


Run: equivalenceClassTest

Tests passed: 1 of 1 test – 88 ms

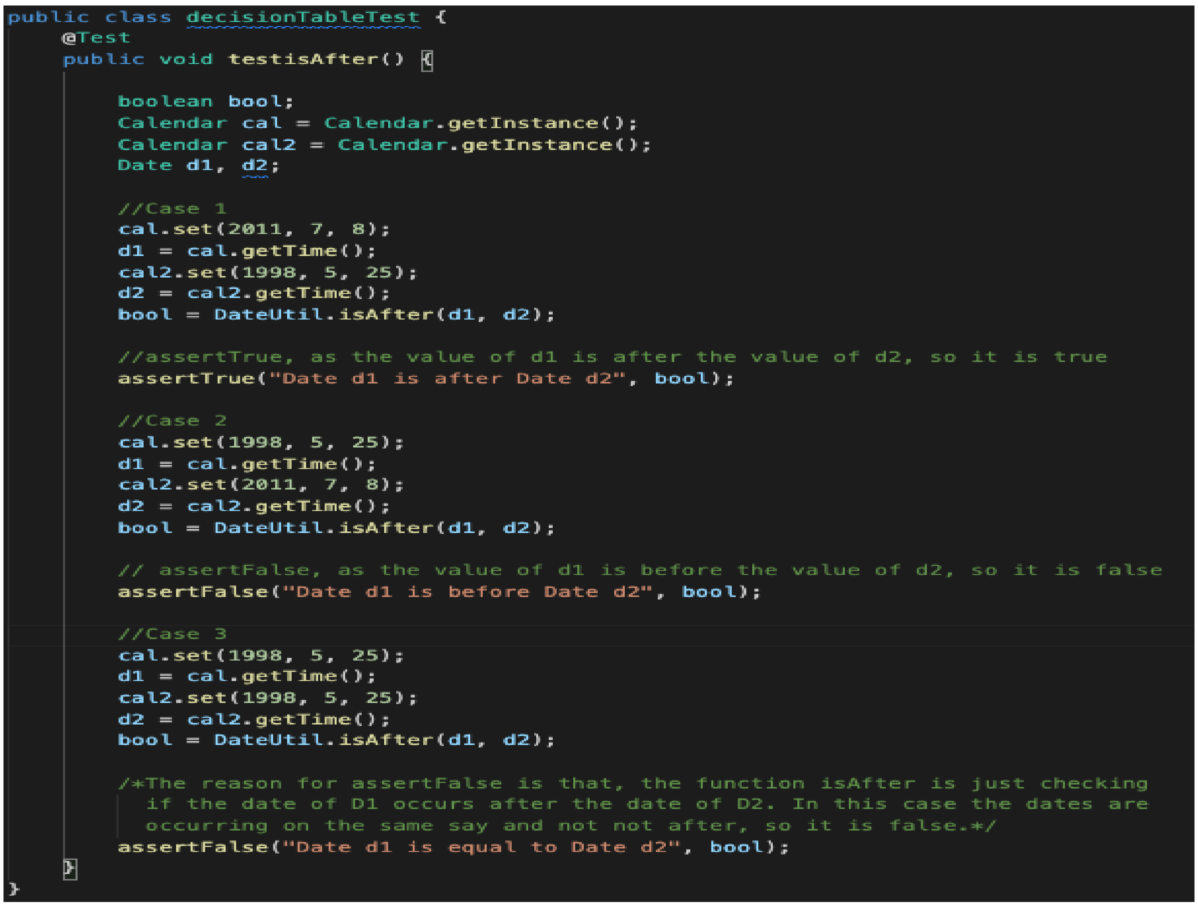
equivalenceClassTest (eecs4313a2b 88 ms
testMinuteStringECT 88 ms

```
"C:\Program Files\Java\jdk-15\bin\java.exe" ...

Process finished with exit code 0
```

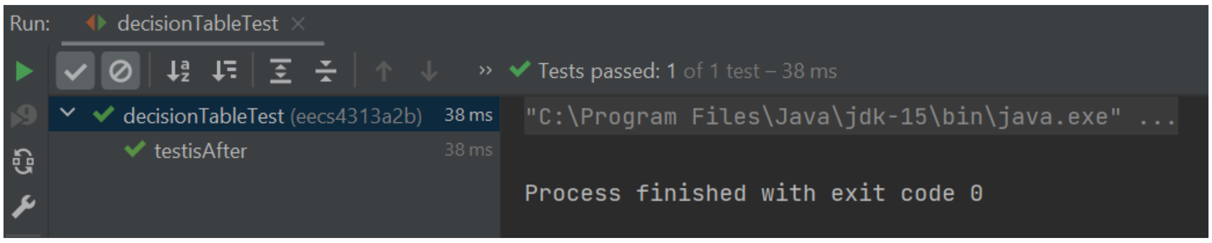
Through equivalence class testing, we were able to prove a few features of the software are working correctly. As our test cases touch upon, showing how an input of minutes greater than 60 for the minuteString() method is supposed to output a string of both hours and minutes. As well as the negative inputs, if the mod % 60 = 0, then they are set to 0.

Through the JUnit Testing, no abnormal behavior was detected. The output results of test cases matched the expected values. For the Equivalence class testing Case, we did not encounter any special value testing which was needed.

1.3.3 Decision Table Testing Evaluation


```
public class decisionTableTest {
    @Test
    public void testisAfter() {

        boolean bool;
        Calendar cal = Calendar.getInstance();
        Calendar cal2 = Calendar.getInstance();
        Date d1, d2;

        //Case 1
        cal.set(2011, 7, 8);
        d1 = cal.getTime();
        cal2.set(1998, 5, 25);
        d2 = cal2.getTime();
        bool = DateUtil.isAfter(d1, d2);

        //assertTrue, as the value of d1 is after the value of d2, so it is true
        assertTrue("Date d1 is after Date d2", bool);

        //Case 2
        cal.set(1998, 5, 25);
        d1 = cal.getTime();
        cal2.set(2011, 7, 8);
        d2 = cal2.getTime();
        bool = DateUtil.isAfter(d1, d2);

        // assertFalse, as the value of d1 is before the value of d2, so it is false
        assertFalse("Date d1 is before Date d2", bool);

        //Case 3
        cal.set(1998, 5, 25);
        d1 = cal.getTime();
        cal2.set(1998, 5, 25);
        d2 = cal2.getTime();
        bool = DateUtil.isAfter(d1, d2);

        /*The reason for assertFalse is that, the function isAfter is just checking
          if the date of D1 occurs after the date of D2. In this case the dates are
          occurring on the same say and not not after, so it is false.*/
        assertFalse("Date d1 is equal to Date d2", bool);
    }
}
```

Decision Table Testing run results



Run: decisionTableTest

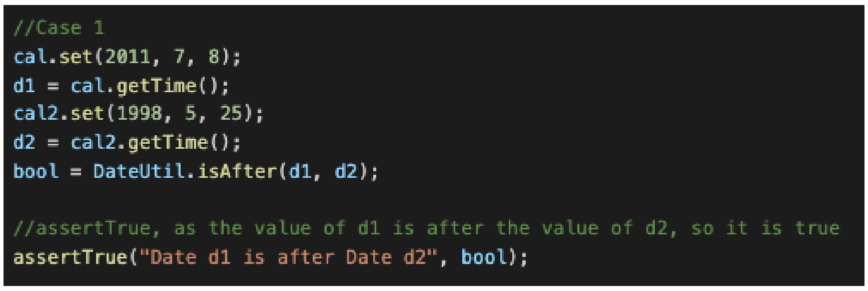
Tests passed: 1 of 1 test – 38 ms

decisionTableTest (eecs4313a2b) 38 ms
testisAfter 38 ms

```
"C:\Program Files\Java\jdk-15\bin\java.exe" ...

Process finished with exit code 0
```

By developing a Decision table as seen in 1.2.6 Decision Table Testing Justification, we can see that three cases are possible. As we are testing the method isAfter(Date 1, Date 2) which takes inputs of type Date, the Junit test first instantiates two Calendar dates. After this, we are creating two Date objects, d1 and d2 which we use later to convert the calendar object to a Data object. Shown below we can see the three test cases which were deemed possible through the Decision table.

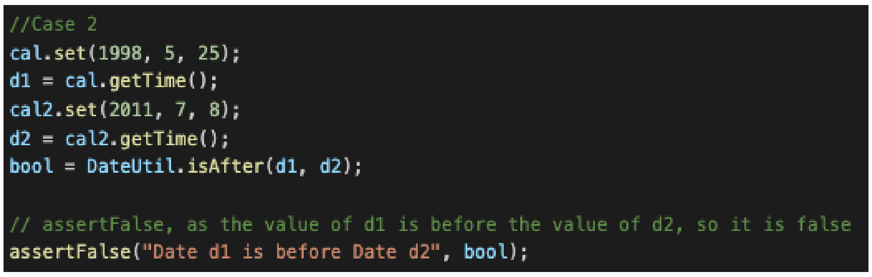
Case 1



```
//Case 1
cal.set(2011, 7, 8);
d1 = cal.getTime();
cal2.set(1998, 5, 25);
d2 = cal2.getTime();
bool = DateUtil.isAfter(d1, d2);

//assertTrue, as the value of d1 is after the value of d2, so it is true
assertTrue("Date d1 is after Date d2", bool);
```

In the Above Case 1, JUnit testing is used to test the method isAfter(). First, the program creates a calendar object with the format: year, month date. As the isAfter() method takes type Date, the value of Date d1 is set to the value of cal through the use of cal.getTime(). This returns a Date object that represents this Calendar's time value. Next, a second calendar object is created and set to the value of Date d2.
Lastly, the variable bool is set to the boolean result of isAfter(d1,d2) giving us a True or false value based on the value of d1 vs d2. In Case 1, we are expecting “Date d1 is after Date d2” and a value of True.

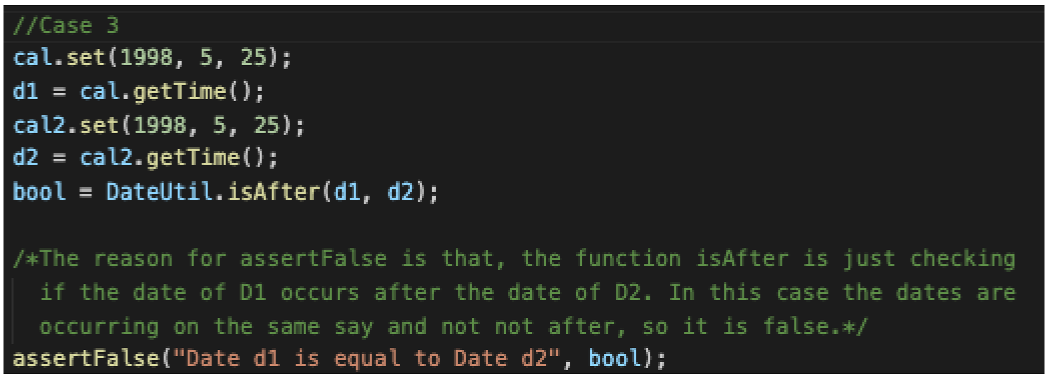
Case 2



```
//Case 2
cal.set(1998, 5, 25);
d1 = cal.getTime();
cal2.set(2011, 7, 8);
d2 = cal2.getTime();
bool = DateUtil.isAfter(d1, d2);

// assertFalse, as the value of d1 is before the value of d2, so it is false
assertFalse("Date d1 is before Date d2", bool);
```

In Case 2, we are expecting “Date d1 is before Date d2” and a value of False as the value of D1 is 1998 and the value of d2 is 2011. The value of d1 is before the value of d2, the method isAfter() tests to see if the value of d1 is after the value of d2.

Case 3



```
//Case 3
cal.set(1998, 5, 25);
d1 = cal.getTime();
cal2.set(1998, 5, 25);
d2 = cal2.getTime();
bool = DateUtil.isAfter(d1, d2);

/*The reason for assertFalse is that, the function isAfter is just checking
  if the date of D1 occurs after the date of D2. In this case the dates are
  occurring on the same say and not not after, so it is false.*/
assertFalse("Date d1 is equal to Date d2", bool);
```

In Case 3, we are expecting “Date d1 is equal to Date d2” and a value of False as the value of d1 = d2. The method isAfter() tests to see if the value of d1 is after the value of d2, here the dates occurring on the same day and not after, so it is false.

Through the JUnit Testing, no abnormal behavior was detected. The output results of test cases matched the expected values. For the Decision table value testing Case, we did not encounter any special value testing which was needed.

1.4 Bug reports

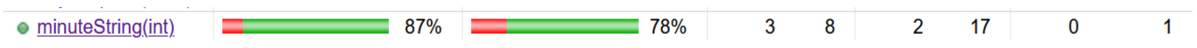
No bugs were found while testing in any of the methods for black-box testing, all features worked as expected. Through testing, we have ensured that features are working properly.

2. White-Box Testing with JUnit

2.1 White Box Description Of Tests

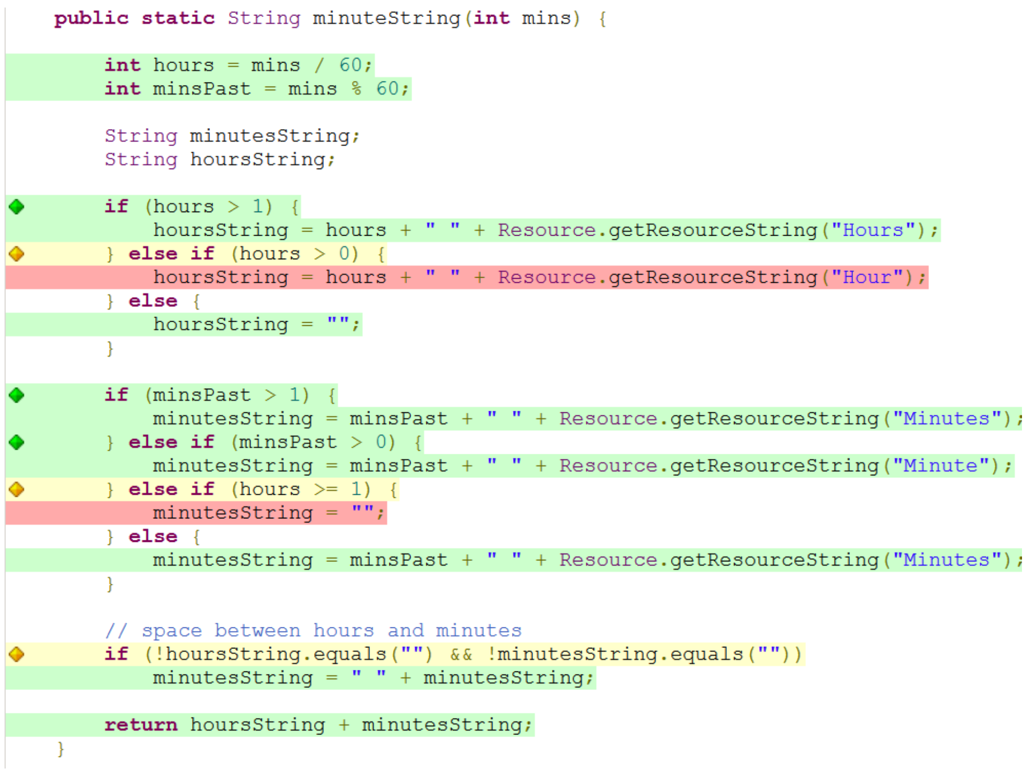
2.1.1 Boundary Value Testing

Code coverage before whitebox testing:



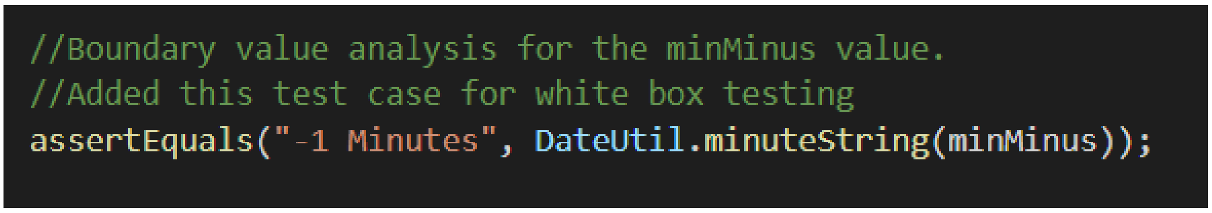
minuteString(int)	🟥🟩	87%	🟥🟩	78%	3	8	2	17	0	1

Statement coverage before whitebox testing:



```
    public static String minuteString(int mins) {

        int hours = mins / 60;
        int minsPast = mins % 60;

        String minutesString;
        String hoursString;

        if (hours > 1) {
            hoursString = hours + " " + Resource.getResourceString("Hours");
        } else if (hours > 0) {
            hoursString = hours + " " + Resource.getResourceString("Hour");
        } else {
            hoursString = "";
        }

        if (minsPast > 1) {
            minutesString = minsPast + " " + Resource.getResourceString("Minutes");
        } else if (minsPast > 0) {
            minutesString = minsPast + " " + Resource.getResourceString("Minute");
        } else if (hours >= 1) {
            minutesString = "";
        } else {
            minutesString = minsPast + " " + Resource.getResourceString("Minutes");
        }

        // space between hours and minutes
        if (!hoursString.equals("") && !minutesString.equals(""))
            minutesString = " " + minutesString;

        return hoursString + minutesString;
    }
```

Added test cases



```
//Boundary value analysis for the minMinus value.
//Added this test case for white box testing
assertEquals("-1 Minutes", DateUtil.minuteString(minMinus));
```

Explanation of the added test cases

Adding a negative value test case helps us see how the method deals with negative values of minutes so it helped in that case.

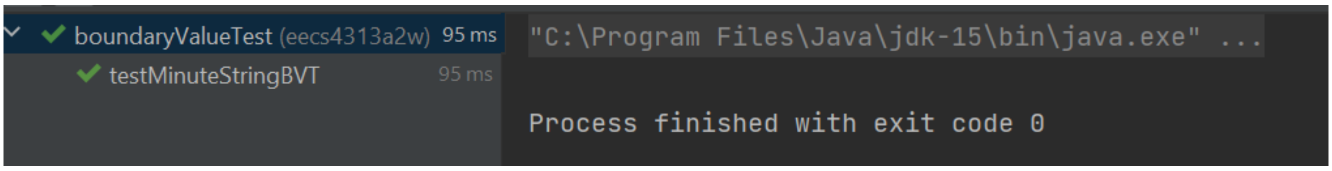
Code coverage after white box testing

The code coverage and statement coverage after white box testing remained the same because we tested for a negative value that outputs “minutes”, which was already accounted for in blackbox testing. For boundary testing, there is a certain upper and lower limit that is tested. During black box testing, we have already checked the upper and lower limits as well as some values.

Adding the test case for a negative value would not change the code coverage because the branch that has not been accounted for is the branch where “Hour” is outputted. Since that is not the goal of boundary value testing, the code coverage remains the same.

The test case for max+ was not added because the max value selected is Integer.MAX_VALUE and if you add 1 to that, in java it will overflow and become Integer.MIN_VALUE.

Boundary value testing run results

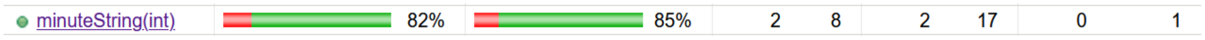


```
✓ boundaryValueTest (eecs4313a2w) 95 ms
    ✓ testMinuteStringBVT          95 ms
```

```
"C:\Program Files\Java\jdk-15\bin\java.exe" ...

Process finished with exit code 0
```

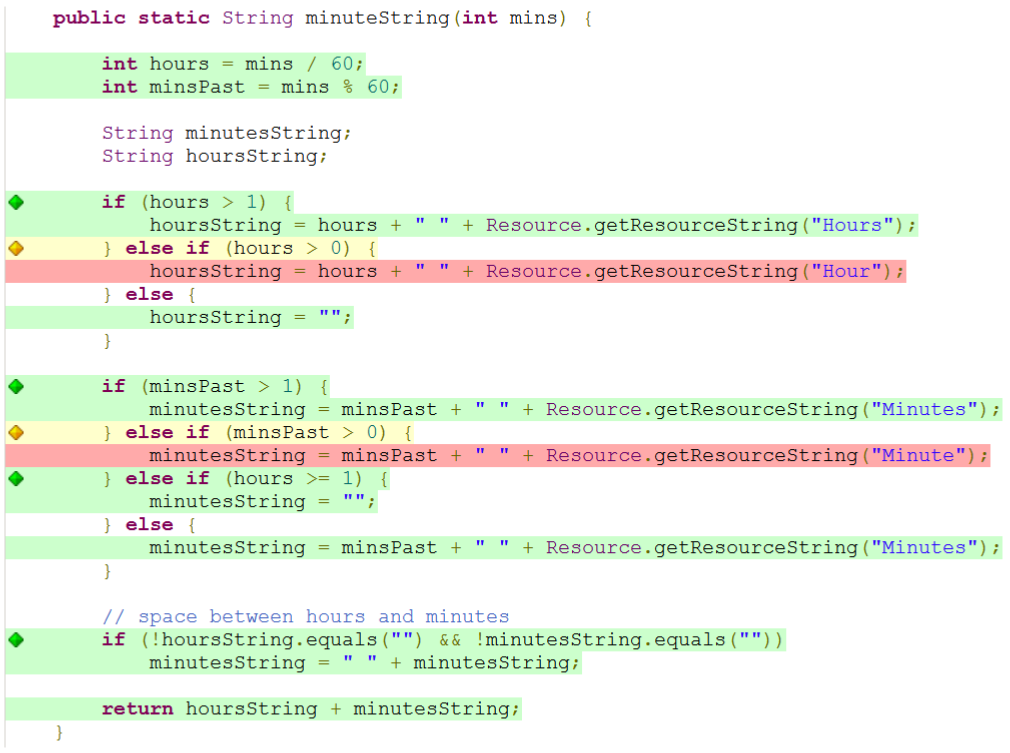
2.1.2 Equivalence Class Testing Added Test Cases

Code coverage before white-box testing:



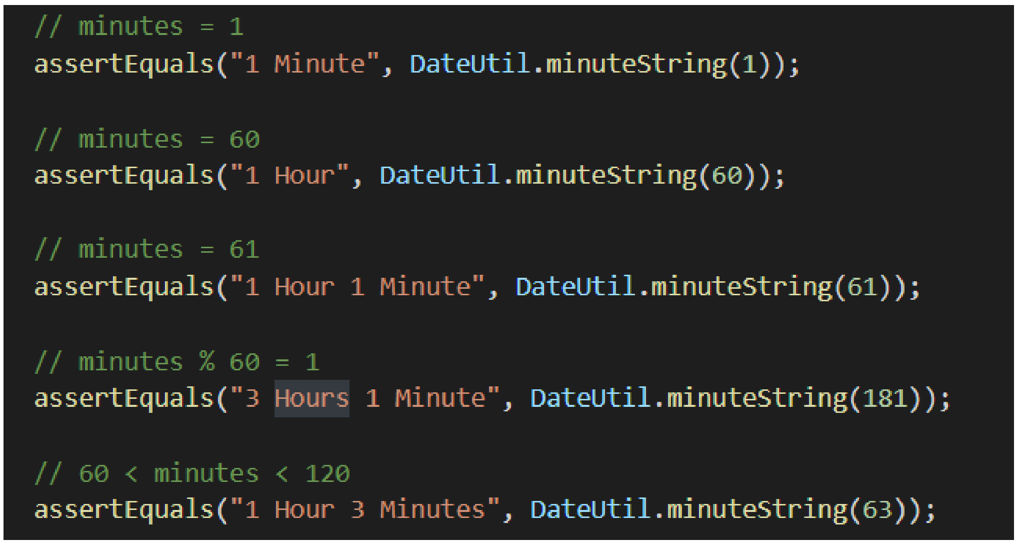
minuteString(int)	82%	85%	2	8	2	17	0	1

Statement coverage before white-box testing:



```java
    public static String minuteString(int mins) {

        int hours = mins / 60;
        int minsPast = mins % 60;

        String minutesString;
        String hoursString;

        if (hours > 1) {
            hoursString = hours + " " + Resource.getResourceString("Hours");
        } else if (hours > 0) {
            hoursString = hours + " " + Resource.getResourceString("Hour");
        } else {
            hoursString = "";
        }

        if (minsPast > 1) {
            minutesString = minsPast + " " + Resource.getResourceString("Minutes");
        } else if (minsPast > 0) {
            minutesString = minsPast + " " + Resource.getResourceString("Minute");
        } else if (hours >= 1) {
            minutesString = "";
        } else {
            minutesString = minsPast + " " + Resource.getResourceString("Minutes");
        }

        // space between hours and minutes
        if (!hoursString.equals("") && !minutesString.equals(""))
            minutesString = " " + minutesString;

        return hoursString + minutesString;
    }
```

Added test cases

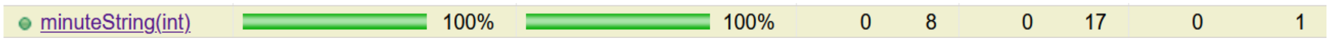


```java
// minutes = 1
assertEquals("1 Minute", DateUtil.minuteString(1));

// minutes = 60
assertEquals("1 Hour", DateUtil.minuteString(60));

// minutes = 61
assertEquals("1 Hour 1 Minute", DateUtil.minuteString(61));

// minutes % 60 = 1
assertEquals("3 Hours 1 Minute", DateUtil.minuteString(181));

// 60 < minutes < 120
assertEquals("1 Hour 3 Minutes", DateUtil.minuteString(63));
```

Explanation of the added test cases

The test cases that were added are to account for cases in which the output is a singular form of minute or hour. It accounts for the cases where the output is 1 Minute only without any hour output, output where it is 1 Hour only without any minute output, output where it is 1 Hour and 1 Minute (both singular), output where it is plural hours and 1 minute, and output where it is singular Hour and plural minutes.

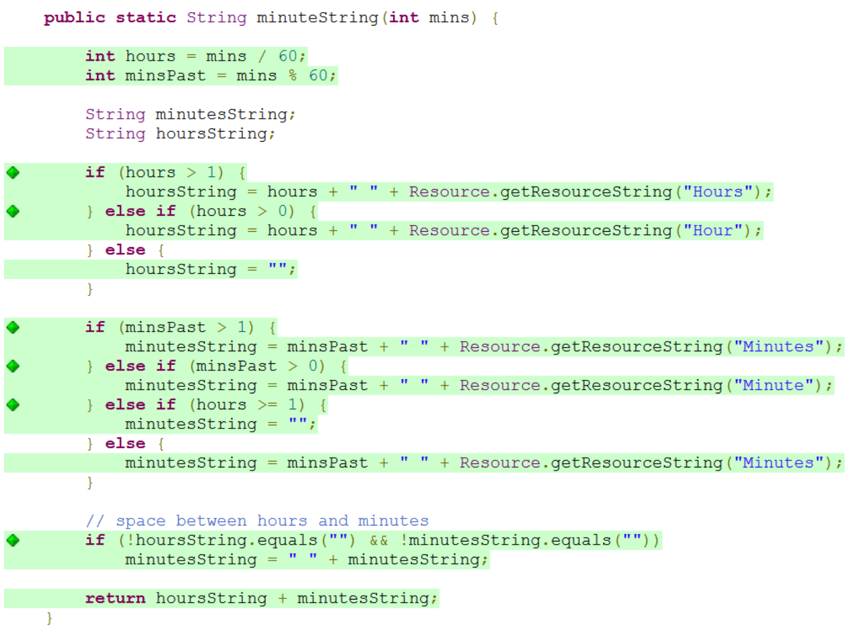
Code coverage after white box testing



minuteString(int)	🟩	100%	🟩	100%	0	8	0	17	0	1

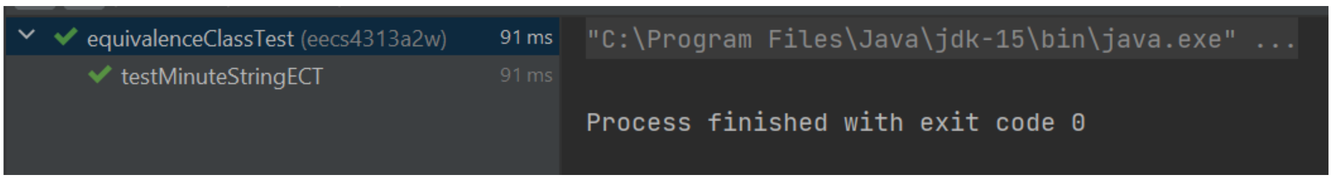
Statement coverage after white-box testing

The code coverage and statement coverage become 100% after adding the test cases in white-box testing. The additional tests that were added were the ones that tested for Hour and Minute singular which brought the code coverage to 100%.



```
public static String minuteString(int mins) {

    int hours = mins / 60;
    int minsPast = mins % 60;

    String minutesString;
    String hoursString;

    if (hours > 1) {
        hoursString = hours + " " + Resource.getResourceString("Hours");
    } else if (hours > 0) {
        hoursString = hours + " " + Resource.getResourceString("Hour");
    } else {
        hoursString = "";
    }

    if (minsPast > 1) {
        minutesString = minsPast + " " + Resource.getResourceString("Minutes");
    } else if (minsPast > 0) {
        minutesString = minsPast + " " + Resource.getResourceString("Minute");
    } else if (hours >= 1) {
        minutesString = "";
    } else {
        minutesString = minsPast + " " + Resource.getResourceString("Minutes");
    }

    // space between hours and minutes
    if (!hoursString.equals("") && !minutesString.equals(""))
        minutesString = " " + minutesString;

    return hoursString + minutesString;
}
```

Equivalence class testing run results

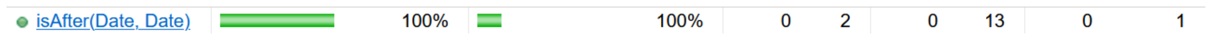


```
✓ equivalenceClassTest (eecs4313a2w)   91 ms
    ✓ testMinuteStringECT              91 ms
```

```
"C:\Program Files\Java\jdk-15\bin\java.exe" ...

Process finished with exit code 0
```

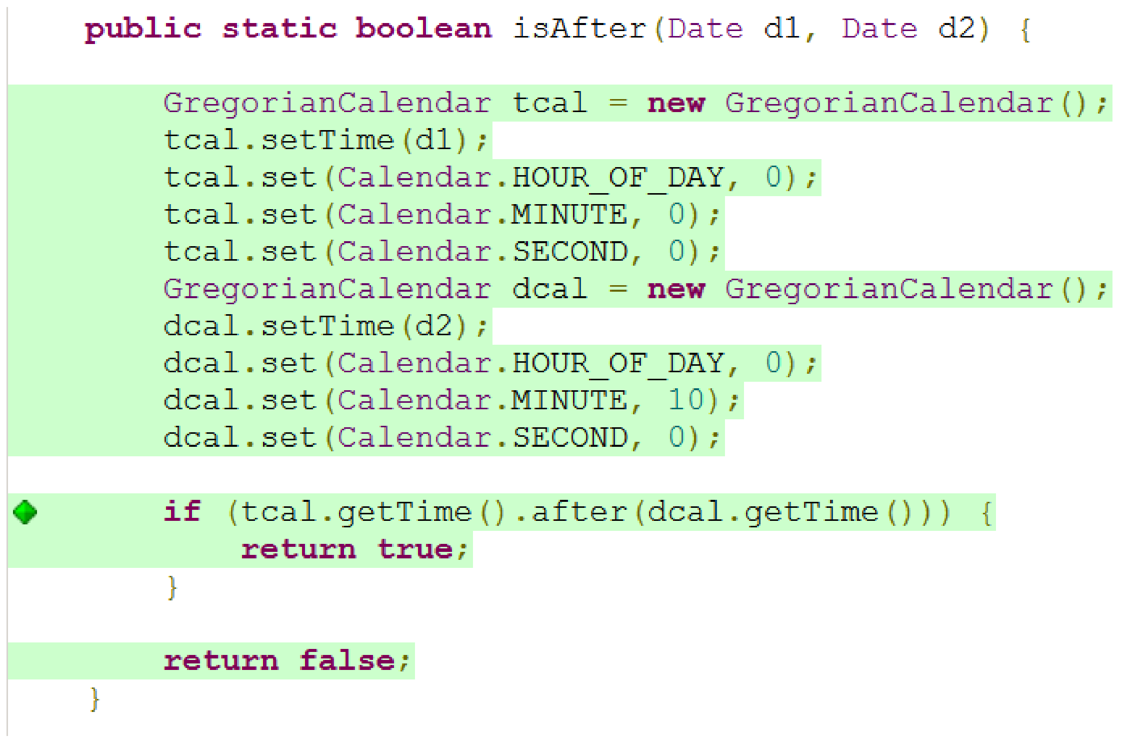
2.1.3 Decision Table Testing Added Test Cases

Code coverage before white-box testing:



● isAfter(Date, Date)	🟩	100%	🟩	100%	0	2	0	13	0	1

Statement coverage before white-box testing:



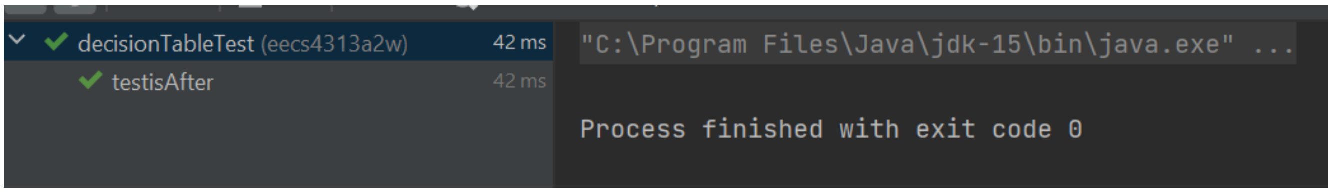
```
    public static boolean isAfter(Date d1, Date d2) {

        GregorianCalendar tcal = new GregorianCalendar();
        tcal.setTime(d1);
        tcal.set(Calendar.HOUR_OF_DAY, 0);
        tcal.set(Calendar.MINUTE, 0);
        tcal.set(Calendar.SECOND, 0);
        GregorianCalendar dcal = new GregorianCalendar();
        dcal.setTime(d2);
        dcal.set(Calendar.HOUR_OF_DAY, 0);
        dcal.set(Calendar.MINUTE, 10);
        dcal.set(Calendar.SECOND, 0);

        if (tcal.getTime().after(dcal.getTime())) {
            return true;
        }

        return false;
    }
```

Added test cases

It was not needed to add any more test cases as all the possible test cases were satisfied and the code coverage was 100%. Through the use of a Decision table, which is shown in Decision Table Testing Justification, It showed that at most 23=8 test cases can be tested. But 5 of 8 test cases were not valid given the properties which they behad to satisfy. The test cases which are being run, allow for coverage of 100% as they are testing all the possible causes of the program and ensuring that they are running correctly. The method isAfter() outputs the result as a boolean value of, either true or false. Knowing that the possible outcomes are; the value of d1 < d2, value of d1 > d2 and lastly value of d1 = d2. We can say that there are $2^3 = 8$possible outcomes, through our decision table testing all the possibilities are covered. The test case covers all cases when; the value of d1 < d2, value of d1 > d2 and lastly value of d1 = d2.

This means that adding more test cases would be beneficial, but there are only 3 cases that need to be tested. This is covered in the test case which we were running to check if; the value of d1 < d2, value of d1 > d2, and lastly value of d1 = d2.

Decision table testing run results

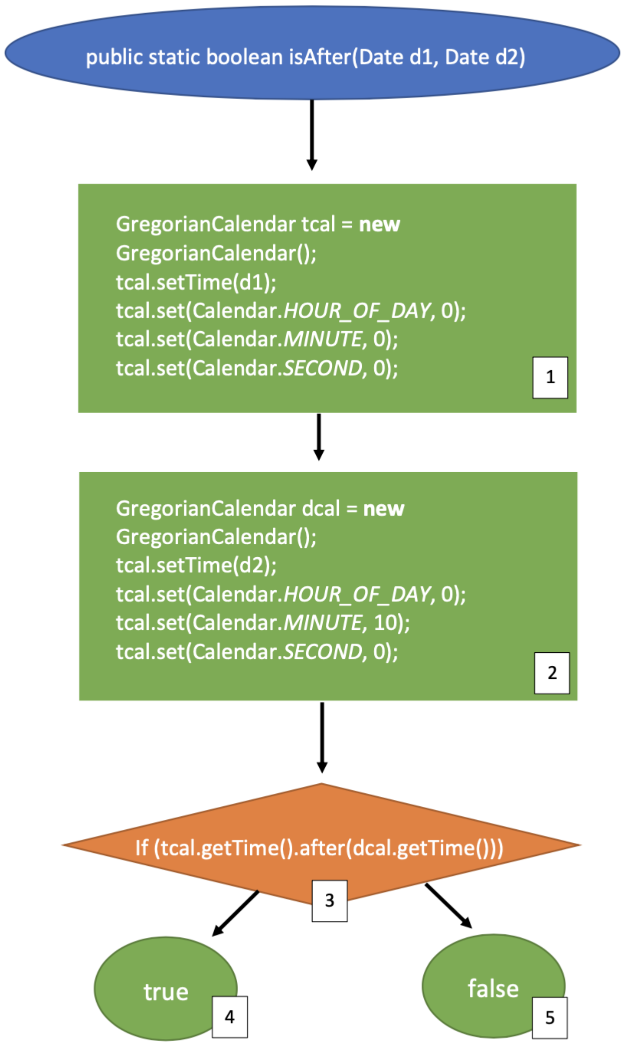


```
decisionTableTest (eecs4313a2w)   42 ms
    testisAfter                   42 ms

"C:\Program Files\Java\jdk-15\bin\java.exe" ...

Process finished with exit code 0
```

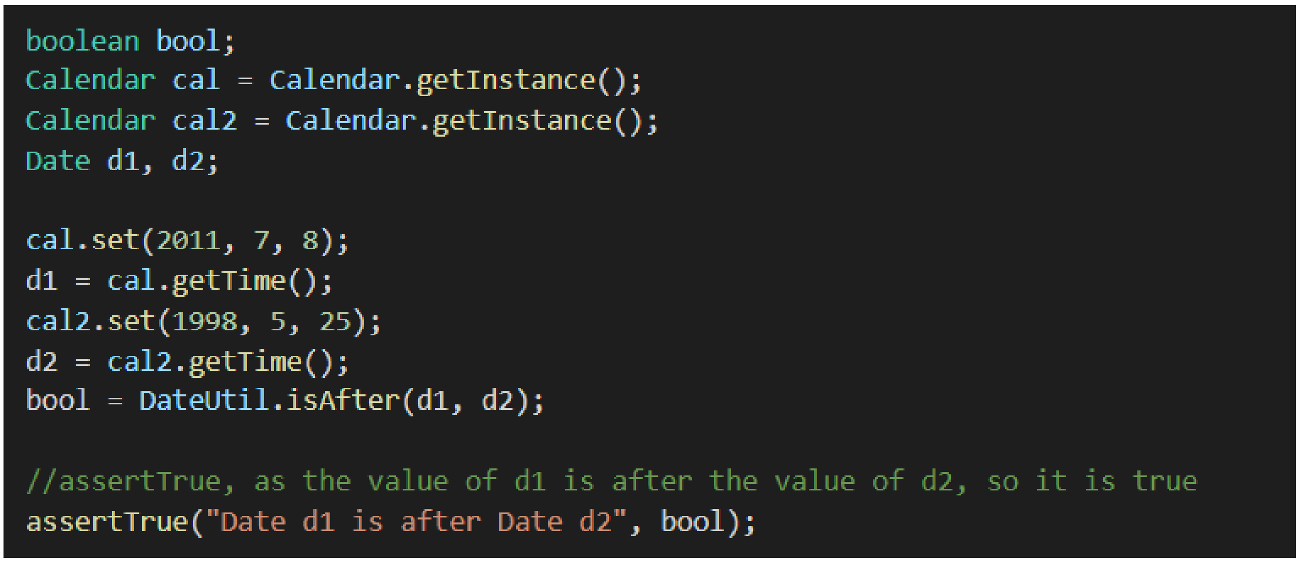
2.1.4 Control flow graph

There are 5 segments for the isAfter(Date d1, Date d2) method. The first and second segment of the method is creating GregorianCalendar objects tcal and dcal; and then initializing their dates d1 and d2 to their respective GregorianCalendar objects tcal and dcal. The third segment of the method is the if statement which checks if d1 is after d2. If the "if" statement is true then it will return true, which is the fourth statement; and if the if statement is false, then it will return false, which is the fifth and last statement. These five segments are constructed into a control flow graph shown in the figure below.

There are a total of 2 paths for this method, isAfter(). The first path is from segment 1 -> 2 -> 3 -> 4 mentioned above. This is when tcal is after dcal or also can be stated as to when input d1is after input d2. Like the if statement in segment 3, is true it will output true for the method. The second path is from segment 1 -> 2 -> 3 -> 5, as the if statement at segment fails. It will then output false for the method, which is segment 5. So the two paths are, 1 2 3 4 and 1 2 3 5.

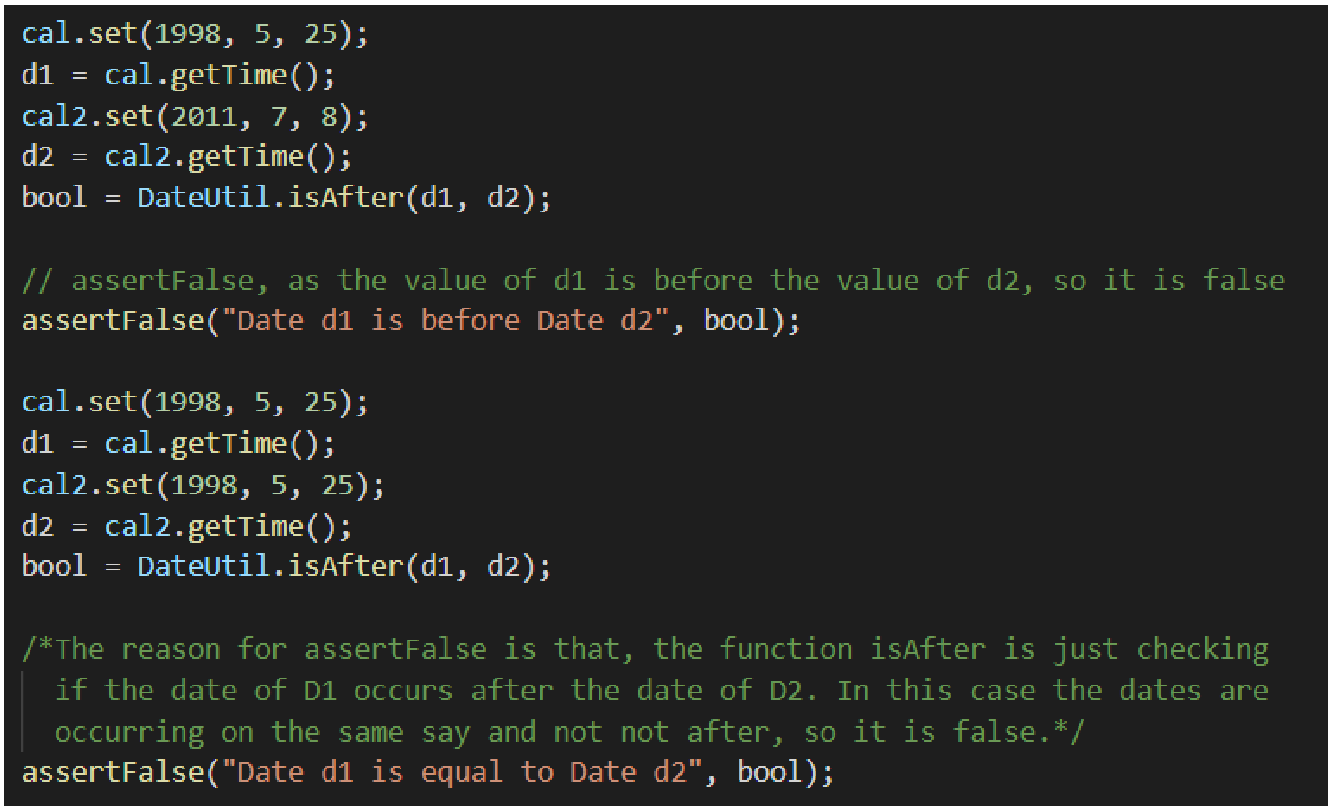

public static boolean isAfter(Date d1, Date d2)
GregorianCalendar tcal = new GregorianCalendar(); tcal.setTime(d1); tcal.set(Calendar.HOUR_OF_DAY, 0); tcal.set(Calendar.MINUTE, 0); tcal.set(Calendar.SECOND, 0);
1
GregorianCalendar dcal = new GregorianCalendar(); tcal.setTime(d2); tcal.set(Calendar.HOUR_OF_DAY, 0); tcal.set(Calendar.MINUTE, 10); tcal.set(Calendar.SECOND, 0);
2
If (tcal.getTime().after(dcal.getTime()))
3
true
4
false
5

The code below accounts for the case where the return value is true. This is path 1, which is 1 2 3 4:



```java
boolean bool;
Calendar cal = Calendar.getInstance();
Calendar cal2 = Calendar.getInstance();
Date d1, d2;

cal.set(2011, 7, 8);
d1 = cal.getTime();
cal2.set(1998, 5, 25);
d2 = cal2.getTime();
bool = DateUtil.isAfter(d1, d2);

//assertTrue, as the value of d1 is after the value of d2, so it is true
assertTrue("Date d1 is after Date d2", bool);
```

The code below accounts for the case where the return value is false. This is path 2, which is 1 2 3 5:



```java
cal.set(1998, 5, 25);
d1 = cal.getTime();
cal2.set(2011, 7, 8);
d2 = cal2.getTime();
bool = DateUtil.isAfter(d1, d2);

// assertFalse, as the value of d1 is before the value of d2, so it is false
assertFalse("Date d1 is before Date d2", bool);

cal.set(1998, 5, 25);
d1 = cal.getTime();
cal2.set(1998, 5, 25);
d2 = cal2.getTime();
bool = DateUtil.isAfter(d1, d2);

/*The reason for assertFalse is that, the function isAfter is just checking
  if the date of D1 occurs after the date of D2. In this case the dates are
  occurring on the same say and not not after, so it is false.*/
assertFalse("Date d1 is equal to Date d2", bool);
```

2.2 Bug reports

No bugs were found while testing in any of the methods for white-box testing, all features worked as expected. Through testing, we ensured that features are working properly.