

EECS 4313 Assignment 2

Black-box and White-box Testing with JUnit

Student Name — Student Number — EECS Account

Edward Vaisman — 212849857 — eddyv

Robin Bandzar — 212200531 — cse23028

Kirusanth Thiruchelvam — 212918298 — kirusant

Sadman Sakib Hasan — 212497509 — cse23152

March 5, 2018

Contents

1	Black Box Testing	3
2	White Box Testing	4
2.1	Equivalence Class Testing	4
2.1.1	Bug Report	8
2.2	Boundary Value Testing	10
2.3	Decision Table Testing	15
2.3.1	Control Flow Graph	16
2.4	Decision Table Testing	21
2.4.1	Control Flow Graph	22

1 Black Box Testing

- Specification of the selected Java methods.
- Justification of the testing technique chosen, i.e., why is it appropriate for this method.
- Description of your application of the three testing strategies. Be clear about which test cases you implemented.
- Evaluation of the test cases derived by the testing technique. Include the screenshots of the test running results. If you had to complement the derived test cases with special value testing, describe that as well. The marker will not read your code in order to see what you tested. You have to describe it.
- Attaching bug reports if bugs are discovered using your testing methods. You should use the same bug report format as in Assignment 1. Do not file these bug reports to the projects bug report system.

2 White Box Testing

2.1 Equivalence Class Testing

- **Technique:** *Equivalence Class Testing*
- **Class:** *net.sf.borg.common.DateUtil.java*
- **Method:** *minuteString(int mins)*
- **Method Description:** This method generate a human reable string for a particular numbe of minutes.It returns the string interms of hours or minutes or both hours and mintues.
 - **mins** - The argument is an integer
- **Justification:** Equivalence class testing is suitable for this method since the argument of this method is an integer which is an independent variable and input range can be partitioned while assuring disjointness and non-redundancy between each partition set. We have chosen these partition integer range based on when we use minute, minutes, hour, and hours.In order to partition the integer argument into hours and minutes,we divide the Minutes by 60 to get the range of hours and the remainder (minutes % 60) to get the range of the minutes.

Table 1: Input ECT

M1:	$\{\text{mins} \mid \text{mins} \% 60 = 0\}$
M2:	$\{\text{mins} \mid \text{mins} \% 60 = 1\}$
M3:	$\{\text{mins} \mid \text{mins} \% 60 > 1\}$
H1:	$\{\text{mins} \mid \text{mins} / 60 \geq 1 \text{ AND } \text{mins} / 60 < 2\}$
H2:	$\{\text{mins} \mid \text{mins} / 60 \geq 2\}$
H3:	$\{\text{mins} \mid \text{mins} / 60 \geq 0 \text{ AND } \text{mins} / 60\}$

Table 2: Output ECT

O1:	H1, M1 = 1 hour
O2:	H1, M2 = 1 hour 1 minute
O3:	H1, M3 = 1 hour y minutes
O4:	H2, M1 = x hours
O5:	H2, M2 = x hours 1 minute
O6:	H2, M3 = x hours y minutes
O7:	H3, M1 = 0 minutes
O8:	H3, M2 = 1 minute
O9:	H3, M3 = y minutes

– x and y are integer variables

Table 3: ECT-Matrix

M1	O1	O4	O7
M2	O2	O5	O8
M3	O3	O6	O9
	H1	H2	H3

The method did not specify how negative minutes should be treated, so we omit the negative integers as an argument for this method. For example, -75 can be converted as -1 hour and 15 minutes or 45 minutes or any other way. Therefore, this case is tested in the whitebox testing through additional test cases after analyzing structure of the method.

- **Statement Coverage Using Black-Box Methods :** 100% [refer to Figure 1 & 2]
- **Justification for getting 100% in blackbox tests :** Since the method converts the integer in terms of human readable minutes and hours, the range of valid

outputs can be partitioned as described in the method description. Since we know how the conversion of integer values into hours and minutes are implemented, there is a need to check negative integers.

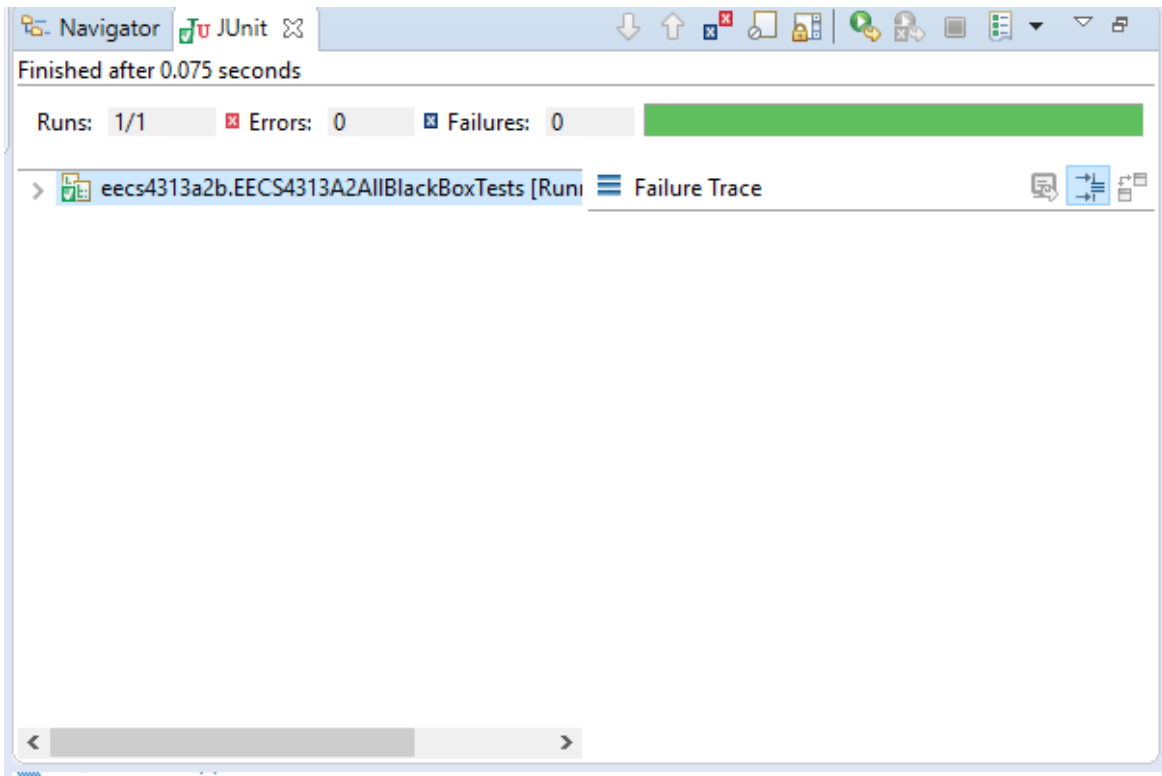


Figure 1: Test run result for the minuteString function

▼ DateUtil.java	56.8 %	133	101	234
▼ DateUtil	56.8 %	133	101	234
isAfter(Date, Date)	0.0 %	0	48	48
setToMidnight(Date)	0.0 %	0	26	26
dayOfEpoch(Date)	0.0 %	0	24	24
minuteString(int)	100.0 %	133	0	133

Figure 2: Statement Coverage View for the minuteString function

- **Additional cases:** Since hours are implemented by dividing integer by 60, the negative integers can produce negative hours. Also, negative hours produced empty string according to the implementation. However, minutes are computed by taking the remainder of the division ($\text{mins} \% 60$) this will produce a positive

number since reminder cannot be negative. Therefore, only two cases we can check in this whitebox testing procedure. They are :

- $\text{Mins}/60 < 0$ and $\text{Mins}\%60 > 1$ - To test negative hours with more than 1 minute [Class 10]
- $\text{Mins}/60 < 0$ and $\text{Mins}\%60 = 1$ - To test negative with 1 minute [Class 11]
This covers the two invalid cases possible for the test based on the implementation of the method. since we can only convert negative integers in terms of minutes according to the implementation of the method. However these two cases produced a bug because the method is producing negative results instead of positive results. The test cases fail for the two cases are shown in figure 3.

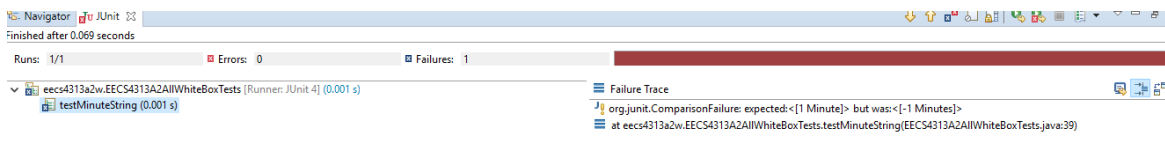


Figure 3: Additional test cases fail due to a bug in the minuteString function

2.1.1 Bug Report

- **Bug Report Title:** DateUtil.minuteString Method does not output the correct result for negative integers
- **Reported by:** Kirusanth Thiruchelvam
- **Date reported:** March 3, 2018
- **Program (or component) name:** net.sf.borg.common.DateUtil.minuteString()
- **Configuration(s):**
 - * Operating System: Windows 10 Pro
 - * Version: 10.0.1.16299 Build 16299
 - * System Manufacturer: SAMSUNG ELECTRONICS CO., LTD.
 - * System Model: QX310/QX410/QX510/SF310/SF410/SF510
 - * BIOS: AMIBIOS Version 03MX.M005.20101011.SCY
 - * Processor: Intel(R) Core(TM) i5 CPU M 460 @ 2.53 GHZ (4CPUs), 2.5GHz
 - * Memory: 8192 MB RAM
 - * Display Device: Intel(R) HD Graphics (Core i5)

- * BORG Calendar Version: 1.8.3
- * Java Version: 1.8.0_161
- **Report type:** Coding Error
- **Reproducibility:** 100%
- **Severity:** Low
- **Problem summary:** When inputting a negative integer as an argument for the minuteString method in DataUtil class. It produces the number in negative which is not correct since the reminder of the negative integer cannot be ne negaitive.
- **Problem description:**
 - Steps to Reproduce
 1. Load the source code of Borg Calender in Java
 2. Create a new junit test case
 3. Call the net.sf.borg.common.DateUtil.minuteString() in the class with -70 as an argument

Results

- **Expected Results:** After inputting the -70, it should gives 10 as the output since hour produce empty string and only minutes produce the results.The expected result is to see positive 10 but it gives -10 since $(-70) \bmod 60$ is 10 minutes.
- **Real Results:** The actual result produce the bug since it shows -10 as the outputs.
- **New or old bug:** New

2.2 Boundary Value Testing

- **Technique:** *Boundary Value Testing*
- **Class:** *net.sf.borg.common.SocketClient.java*
- **Method:** *sendMsg(String host, int port, String msg)*
- **Method Description:** This method sends a given message to a given host, port and returns the response from the socket.
 - the first argument *host* is the host that the socket client should be connected to.
 - the second argument *port* is the port on the host that the socket client should be connected to
 - the third argument *msg* is the message that should be sent over to the host and port given.
- **Statement Coverage Using Original Black-Box Methods:**
82% [refer to Figures 1 & 2]
- **Statement Coverage Including Additional Testcases:**
88.5% [refer to Figures 3 & 4]
- **Branch Coverage Using Original Testcases:**
50% (4/8) [refer to Figures 5]
- **Branch Coverage Including Additional Testcases:**
62.5% (5/8) [refer to Figure 6]
- **The Additional Tests:** The additional tests that were added needed to cause exceptions or pass null values to the method, these two cases are not quantifiable. Thus, our original boundary value testing testsuite needed more testcases to increase coverage. The added function is further described below:
 - The testcase *sendNullMessageTest* considered if the message String sent was null. There is a null checking branch path that gets executed due to this testcase. Also, this causes an IOException to be thrown because the InputStream attempts to read in a null String.

```
@Test
public void sendNullMessageTest () {
    String msg = null;
```

```
SocketServer ss;
String response;
try {
    ss = new SocketServer(2920, null);
    response = SocketClient.sendMsg("localhost",
        2920, msg);
    assertEquals("Testing if a localhost on port 2920
        sends a message", response, msg);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

- **Why not 100%?:** In the *sendMsg* function there are two try catches to handle *IOExceptions*. To be able to get both inner exception segments 1 and 2 fully covered, a more granular approach using sockets and concurrency would be needed. At that point you are not testing the code, but rather how Java handles socket connections.

Segment 1

```
} catch (IOException e) {
    if (s != null)
        s.close();
    throw e;
}
```

Segment 2

```
finally {
    try {
        if (s != null)
            s.close();
    } catch (IOException e2) {
        // empty
    }
}
```

```

33 public static String sendMsg(String host, int port, String msg) throws IOException {
34     Socket s = null;
35     String line = null;
36     try {
37         s = new Socket(host, port);
38         BufferedReader sin = new BufferedReader(new InputStreamReader(s
39             .getInputStream()));
40         PrintStream sout = new PrintStream(s.getOutputStream());
41         sout.println(msg);
42         line = sin.readLine();
43         // Check if connection is closed (i.e. for EOF)
44         if (line == null) {
45             log.info("Connection closed by server.");
46         }
47     } catch (IOException e) {
48         if (s != null)
49             s.close();
50         throw e;
51     }
52     // Always be sure to close the socket
53     finally {
54         try {
55             if (s != null)
56                 s.close();
57         } catch (IOException e2) {
58             // empty
59         }
60     }
61
62     return line;
63 }

```

Figure 4: Statement Coverage View for the sendMsg function before additional testcases

▼ SocketClient.java		55.7 %	54	43	97
▼ SocketClient		55.7 %	54	43	97
sendMessage(String)		0.0 %	0	20	20
sendMsg(String, int, String)		82.0 %	50	11	61

Figure 5: Statement Coverage Metrics for the sendMsg function before additional testcases

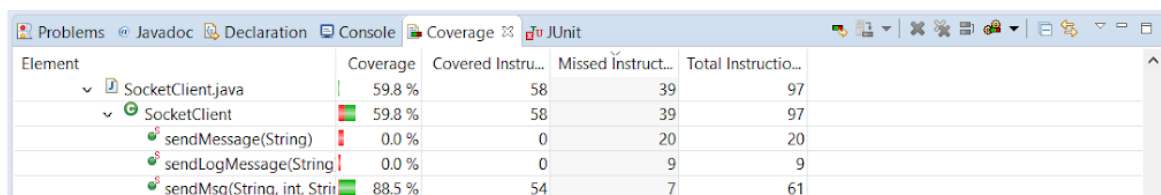
```

public static String sendMsg(String host, int port, String msg) throws IOException {
    Socket s = null;
    String line = null;
    try {
        s = new Socket(host, port);
        BufferedReader sin = new BufferedReader(new InputStreamReader(s
            .getInputStream()));
        PrintStream sout = new PrintStream(s.getOutputStream());
        sout.println(msg);
        line = sin.readLine();
        // Check if connection is closed (i.e. for EOF)
        if (line == null) {
            log.info("Connection closed by server.");
        }
    } catch (IOException e) {
        if (s != null)
            s.close();
        throw e;
    }
    // Always be sure to close the socket
    finally {
        try {
            if (s != null)
                s.close();
        } catch (IOException e2) {
            // empty
        }
    }

    return line;
}

```

Figure 6: Statement Coverage View for the sendMsg function after adding additional testcases



Element	Coverage	Covered Instru...	Missed Instruct...	Total Instructio...
SocketClient.java	59.8 %	58	39	97
SocketClient	59.8 %	58	39	97
sendMessage(String)	0.0 %	0	20	20
sendLogMessage(String)	0.0 %	0	9	9
sendMsg(String, int, Strit	88.5 %	54	7	61

Figure 7: Statement Coverage Metrics for the sendMsg function after adding additional testcases





▼ SocketClient.java		33.3 %	4	8	12
▼ SocketClient		33.3 %	4	8	12
sendMessage(String)		0.0 %	0	4	4
sendMsg(String, int, String)		50.0 %	4	4	8

Figure 8: Branch Coverage Metrics for the sendMsg function before adding additional test-cases





▼ SocketClient.java		41.7 %	5	7	12
▼ SocketClient		41.7 %	5	7	12
sendMessage(String)		0.0 %	0	4	4
sendMsg(String, int, String)		62.5 %	5	3	8

Figure 9: Branch Coverage Metrics for the sendMsg function after adding additional testcases

2.3 Decision Table Testing

- **Technique:** *Decision Table Testing*
- **Class:** *net.sf.borg.common.DateUtil.java*
- **Method:** *isAfter(Date d1, Date d2)*
- **Method description:** The method checks if a given date *d1* falls on a later calendar day than date *d2*. It returns **true** if *d1* does fall on a later calendar day than *d2* and **false** otherwise.
 - **d1** - The first argument is of type Java Date Object.
 - **d2** - The second argument is of type Java Date Object.
- **Justification:** Decision table testing technique is an appropriate testing technique for this method because there are decision making to be done among the input variables. It consists of logical relationships among the input variables, i.e date *d1* appearing before, after or at the same time as date *d2*, which directly affects the output.

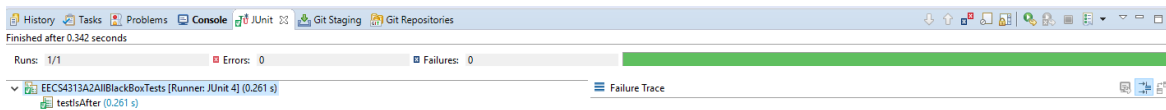


Figure 10: Test run result for the isAfter function

```

30-  /**
31   * Checks if one date falls on a later calendar day than another.
32   *
33   * @param d1
34   *         the first date
35   * @param d2
36   *         the second date
37   *
38   * @return true, if is after
39   */
40- public static boolean isAfter(Date d1, Date d2) {
41
42     GregorianCalendar tcal = new GregorianCalendar();
43     tcal.setTime(d1);
44     tcal.set(Calendar.HOUR_OF_DAY, 0);
45     tcal.set(Calendar.MINUTE, 0);
46     tcal.set(Calendar.SECOND, 0);
47     GregorianCalendar dcal = new GregorianCalendar();
48     dcal.setTime(d2);
49     dcal.set(Calendar.HOUR_OF_DAY, 0);
50     dcal.set(Calendar.MINUTE, 10);
51     dcal.set(Calendar.SECOND, 0);
52
53     if (tcal.getTime().after(dcal.getTime())) {
54         return true;
55     }
56
57     return false;
58 }

```

Figure 11: Statement Coverage View for the isAfter function

▼ DateUtil	22.2 %	48	168	216
minuteString(int)	0.0 %	0	115	115
setToMidnight(Date)	0.0 %	0	26	26
dayOfEpoch(Date)	0.0 %	0	24	24
isAfter(Date, Date)	100.0 %	48	0	48

Figure 12: Statement Coverage Metrics for the isAfter function

2.3.1 Control Flow Graph

The following is the code snippet for the *isAfter* function:


```

/**
 * Checks if one date falls on a later calendar day than
 * another.
 *
 * @param d1
 *         the first date
 * @param d2
 *         the second date
 *
 * @return true, if is after
 */
1. public static boolean isAfter(Date d1, Date d2) {

2.   GregorianCalendar tcal = new GregorianCalendar();
3.   tcal.setTime(d1);
4.   tcal.set(Calendar.HOUR_OF_DAY, 0);
5.   tcal.set(Calendar.MINUTE, 0);
6.   tcal.set(Calendar.SECOND, 0);
7.   GregorianCalendar dcal = new GregorianCalendar();
8.   dcal.setTime(d2);
9.   dcal.set(Calendar.HOUR_OF_DAY, 0);
10.  dcal.set(Calendar.MINUTE, 10);
11.  dcal.set(Calendar.SECOND, 0);

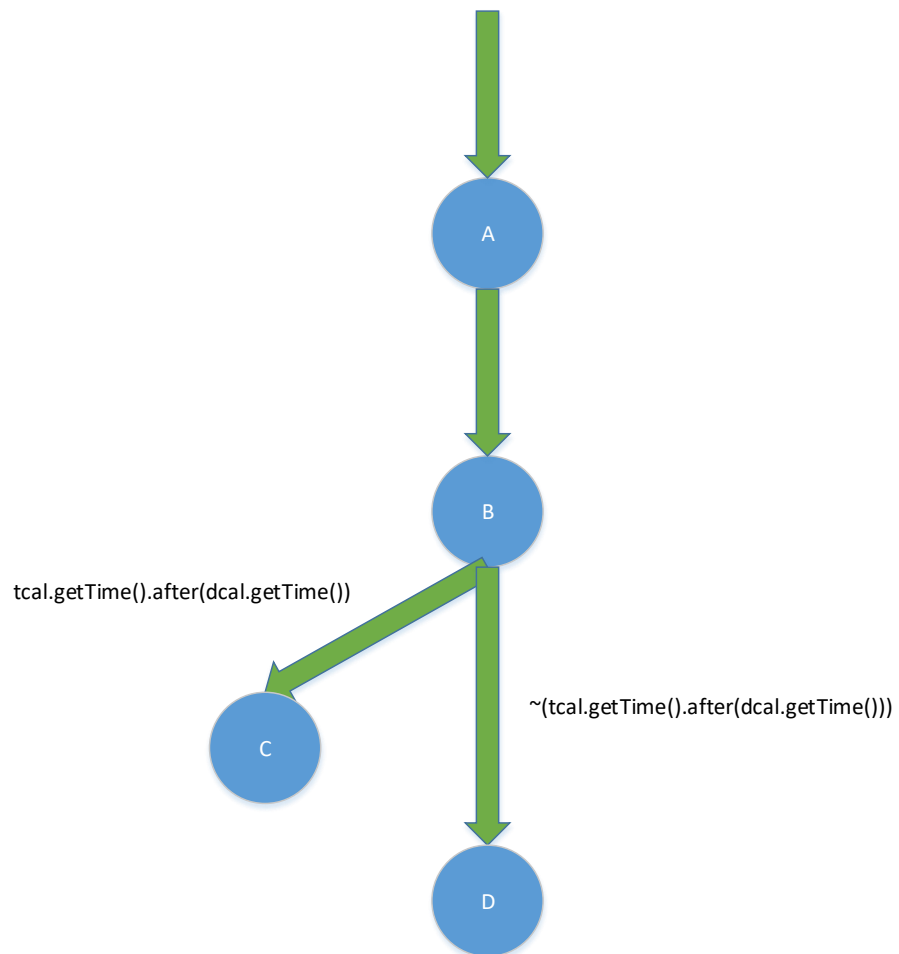
12.  if (tcal.getTime().after(dcal.getTime())) {
13.    return true;
14.  }

15.  return false;
16. }

```

Name	Covered Statements
A	from line 1 to 11
B	if (tcal.getTime().after(dcal.getTime()))
C	return true;
D	return false;

Table 4: CFG Segment Table for the isAfter method

Figure 13: Control Flow Graph for the `isAfter` function

- **Number of paths in the method: 2**
 - **Path P1** - The first path is when the method returns *true* if date d1 is after date d2. Following the CFG diagram above, refer to diagram 17, this is path is: **ABC**.
 - **Path P2** - The second path is when the method returns *false* if date d1 is not after date d2 (i.e d1 is either equal to or before d2). Following the CFG diagram above, refer to diagram 17, this is path is: **ABD**.
- **Estimated % of path covered in the test: 100%**

The following code snippet depicts the test case and path coverages for *isAfter* function:

```
@Test
public void testIsAfter() {
    /** Method used: Decision Table Testing **/

    Date d1 = new Date(117, 11, 3);
    Date d2 = new Date(117, 11, 3);
    boolean result;

    // date d1 is equal to d2
    result = DateUtil.isAfter(d1, d2);
    assertFalse("Date d1 is equal to d2", result);

    // date d1 is before d2
    d1.setDate(2);
    result = DateUtil.isAfter(d1, d2);
    assertFalse("Date d1 is before d2", result);

    // date d1 is after d2
    d1.setDate(4);
    result = DateUtil.isAfter(d1, d2);
    assertTrue("Date d1 is after d2", result);
}
```

Path coverage **P1**:

```
// date d1 is after d2
d1.setDate(4);
result = DateUtil.isAfter(d1, d2);
assertTrue("Date d1 is after d2", result);
```

Path coverage **P2**:

```
// date d1 is equal to d2
result = DateUtil.isAfter(d1, d2);
assertFalse("Date d1 is equal to d2", result);

// date d1 is before d2
d1.setDate(2);
result = DateUtil.isAfter(d1, d2);
assertFalse("Date d1 is before d2", result);
```