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 Specifications

- **The Goal**: We were given the BORG Calendar Application and relevant configuration files to apply black-box and white-box testing techniques taught in class on 3 methods of our choice.
- **Software to be tested**: BORG Calendar v.1.8.3
- **Testing Framework**: JUnit 4, accessed through both Eclipse UI and command-line
- **Langauge**: Java

2 Black Box Testing

2.1 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.

  IOException: If an I/O error occurs when sending the message.
- **Justification**: Boundary value testing is best suited for methods that have inputs that could be seperated into partitions. For this method the port could be partitioned. We have our valid partition which is between 0 and 65535 (inclusive) and our invalid partitions which is any port<0 or any port>65535. The msg could be anything and the host could be partitioned into valid/invalid hostnames.

- **Evaluation**: The tests below applies weak robust testing and weak normal testingon the method *sendMsg*. Weak normal testing passes but weak robust testing reveals a bug within the method that causes the method to throw an IllegalArgumentException if the port parameter is outside the specified range of valid port values, which is between 0 and 65535, inclusive.

  The tests are designed to not fail on any IOExceptions that may occur such as UnknownHostException or ConnectionException since the method specifies that it may *throw* an IOException.

  Refer to Figure. **??** and Listing. **??** for weak normal testing.
  Refer to Figure. **??**, Figure. **??** and Listing **??** for weak robust testing.

Listing 1: Weak Normal Testing Variables

```
String validHost = "localhost";
port_norm = 2929; // x_norm
port_min = 0; // x_min
port_min_plus = 1; // x_min+
port_max = 65535; // x_max
port_max_minus = 65534; // x_max-
```

Listing 2: Weak Robust Testing Variables

```
String validHost = "localhost";
port_norm = 2929; // x_norm
port_min = 0; // x_min
port_min_plus = 1; // x_min+
port_max = 65535; // x_max
port_max_minus = 65534; // x_max-
// robustness
String invalidHost = "asdfasdf"; //unknownhostexception
int port_min_minus = -1; // x_min-
int port_max_plus = 65536; // x_max_+
```

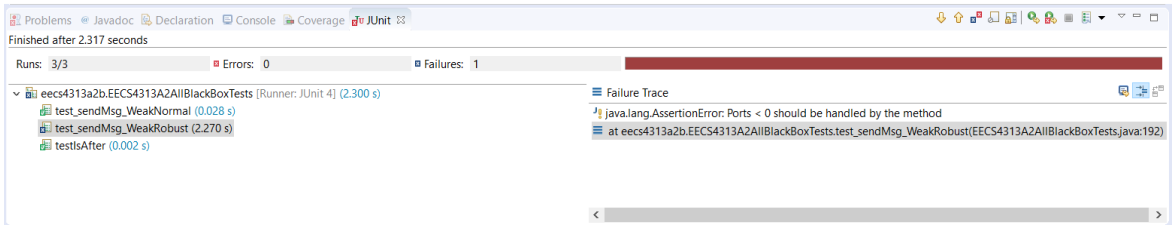


Figure 1: Test results using Weak Robust Boundary Value Testing

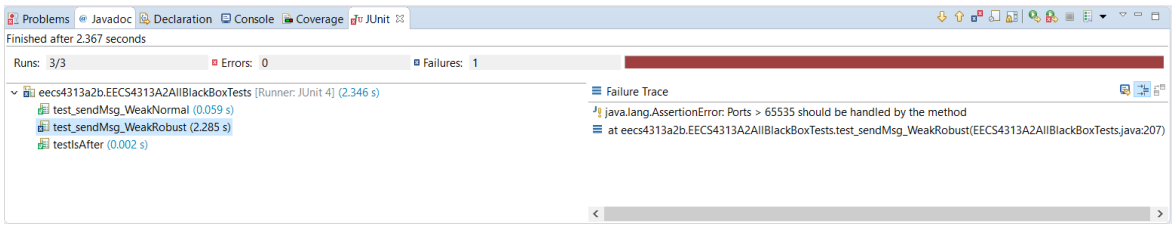


Figure 2: Test results using Weak Robust Boundary Value Testing

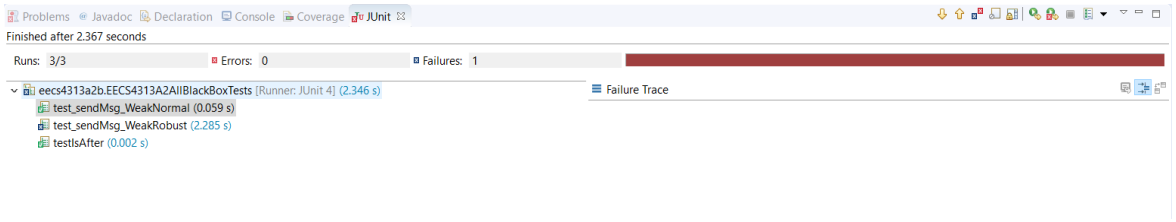


Figure 3: Test results using Weak Normal Boundary Value Testing

2.1.1 Testing Code

```
public class EECS4313A2AllBlackBoxTests implements
    SocketHandler {

  /**
   * process a socket message
   */
  @Override
  public synchronized String processMessage(String msg) {
    return msg;
  }
  @Test
  public void test_sendMsg_WeakNormal() {
    /** Method used: Boundary Value Testing **/
    String validHost = "localhost";

    int port_norm = 2929; // x_norm
    int port_min = 0; // x_min
    int port_min_plus = 1; // x_min+
    int port_max = 65535; // x_max
    int port_max_minus = 65534; // x_max-

    String response = "";
    // port_norm
    String msg = "Port 2929";
    SocketServer ss = new SocketServer(port_norm, this);
    try {
      response = SocketClient.sendMsg(validHost, port_norm,
          msg);
      assertEquals("Testing if a localhost on port_norm sends
          a message", response, msg);
    } catch (IOException e) {
      e.printStackTrace();
    }

    // port_min
    /*
     * Throws connection problem. port 0 isn't available on
        my computer Connect
```

```
 * Exception extends Socket Exception which extends
    IOException
 */
msg = "Port 0";
try {
  ss = new SocketServer(port_min, this);
  response = SocketClient.sendMsg(validHost, port_min,
     msg);
  assertEquals("Testing if a localhost on port_min sends
     a message", response, msg);
} catch (IOException e) {
  e.printStackTrace();
}
// port_min+
msg = "Port 1";
try {
  ss = new SocketServer(port_min_plus, this);
  response = SocketClient.sendMsg(validHost,
     port_min_plus, msg);
  assertEquals("Testing if a localhost on port port_min+
     sends a message", response, msg);
} catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}

// port_max
msg = "Port 65535";
try {
  ss = new SocketServer(port_max, this);
  response = SocketClient.sendMsg(validHost, port_max,
     msg);
  assertEquals("Testing if a localhost on port port_max
     sends a message", response, msg);
} catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}

// port_max-
msg = "Port 65534";
```

```
    try {
      ss = new SocketServer(port_max_minus, this);
      response = SocketClient.sendMsg(validHost,
         port_max_minus, msg);
      assertEquals("Testing if a localhost on port_max- sends
         a message", response, msg);

    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  @Test
  public void test_sendMsg_WeakRobust() {
    /** Method used: Boundary Value Testing **/
    String validHost = "localhost";

    int port_norm = 2929; // x_norm
    int port_min = 0; // x_min
    int port_min_plus = 1; // x_min+
    int port_max = 65535; // x_max
    int port_max_minus = 65534; // x_max-

    // robustness
    String invalidHost = "asdfasdf";
    int port_min_minus = -1; // x_min-
    int port_max_plus = 65536; // x_max_+

    String response = "";
    // port_norm
    String msg = "Port 2929";
    SocketServer ss = new SocketServer(port_norm, this);
    try {
      response = SocketClient.sendMsg(validHost, port_norm,
         msg);
      assertEquals("Testing if a localhost on port_norm sends
         a message", response, msg);
    } catch (IOException e) {
      e.printStackTrace();
    }
```

```
/* Unknown host exception extends IOException */
try {
  response = SocketClient.sendMsg(invalidHost, port_norm,
     msg);
  assertEquals("Testing if an invalid host on port_norm
     sends a message", response, msg);
} catch (IOException e) {
  e.printStackTrace();
}

// port_min
/*
 * Throws connection problem. port 0 isn't available on
    my computer Connect
 * Exception extends Socket Exception which extends
    IOException
 */
msg = "Port 0";
try {
  ss = new SocketServer(port_min, this);
  response = SocketClient.sendMsg(validHost, port_min,
     msg);
  assertEquals("Testing if a localhost on port_min sends
     a message", response, msg);
} catch (IOException e) {
  e.printStackTrace();
}
// port_min+
msg = "Port 1";
try {
  ss = new SocketServer(port_min_plus, this);
  response = SocketClient.sendMsg(validHost,
     port_min_plus, msg);
  assertEquals("Testing if a localhost on port port_min+
     sends a message", response, msg);
} catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}

// port_max
```

```
msg = "Port 65535";
try {
  ss = new SocketServer(port_max, this);
  response = SocketClient.sendMsg(validHost, port_max,
     msg);
  assertEquals("Testing if a localhost on port port_max
     sends a message", response, msg);
} catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}

// port_max-
msg = "Port 65534";
try {
  ss = new SocketServer(port_max_minus, this);
  response = SocketClient.sendMsg(validHost,
     port_max_minus, msg);
  assertEquals("Testing if a localhost on port_max- sends
     a message", response, msg);

} catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}

// port_min-
/*
 * Illegal argument Exception
 */
msg = "Port -1";
try {
  ss = new SocketServer(port_min_minus, this);
  response = SocketClient.sendMsg(validHost,
     port_min_minus, msg);
  assertEquals("Testing if a localhost on port_min- sends
     a message", response, msg);
} catch (IOException e) {
  e.printStackTrace();
} catch (IllegalArgumentException iae) {
  fail("Ports < 0 should be handled by the method");
```

```
    }

    // port_max+
    /*
     * Illegal argument Exception
     */
    msg = "Port 65536";
    try {
      ss = new SocketServer(port_max_plus, this);
      response = SocketClient.sendMsg(validHost,
         port_max_plus, msg);
      assertEquals("Testing if a localhost on port_max+ sends
         a message", response, msg);
    } catch (IOException e) {
      e.printStackTrace();
    } catch (IllegalArgumentException iae) {
      fail("Ports > 65535 should be handled by the method");
    }
  }
```

2.1.2 Bug Report

- **Bug Report Title**: Socket port values below 0 or above 65535 causes application to not be runnable after restart.
- **Reported by**: Edward Vaisman
- **Date reported**: March, 3rd, 2018
- **Program (or component) name**: BORG Calendar version 1.8.3 - SocketServer Constructor, SocketClient sendMsg, PrefName.SOCKETPORT
- **Configuration(s)**:
  System Info
  - Operating System: Windows 10 Home 64-bit (10.0, Build 16299) (16299.rs3_release.170928-1534)
  - Language: English (Regional Setting: English)
  - System Manufacturer: Dell Inc.
  - System Model: Inspiron 7559
  - Display Device: Intel(R) HD Graphics 530
  - Processor: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz (8 CPUs), 2.6GHz
  - Memory: 8192MB RAM
  - BORG Calendar Version: 1.8.3
  - Java Version: 1.8.0_161

  BORG Settings
  - Socket Port: -2929
- **Report type**: Coding Error.
- **Reproducibility**: 100% (Tested on 4 seperate machines.)
- **Severity**: High (Fatal)
- **Problem summary**: After changing the socket port to -2929 and restarting the application causes BORG to be unusable even after a clean install.
- **Problem description**:
  Applying boundary value junit testing to the sendMsg method in SocketClient reveals that PrefName.SOCKETPORT is allowed to store socketports that aren't valid. As a result it causes an unhandled exception to throw in

SocketClient and SocketServer when trying to use a port that isn't valid. The reproduction steps describe how to reach this bug within the application.

Steps to Reproduce in Application

1. Run the application

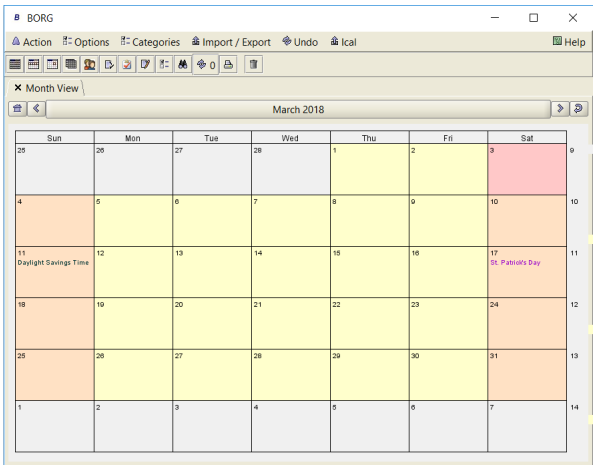


Figure 4: Run the application

2. Select "Options" → "Edit Preferences".

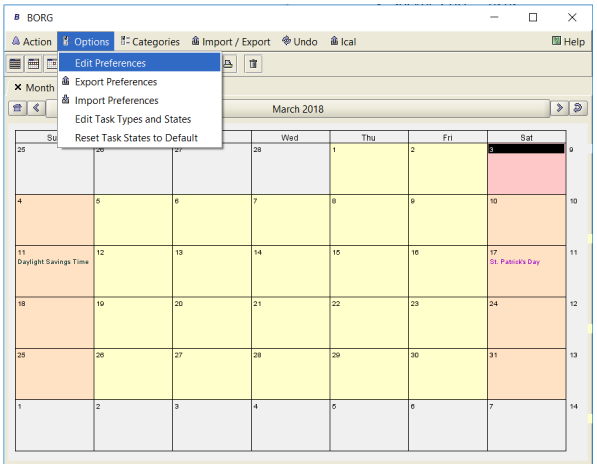


Figure 5: Opening the options window

3. The “Options” window appears. Select the “Miscellaneous” Tab.

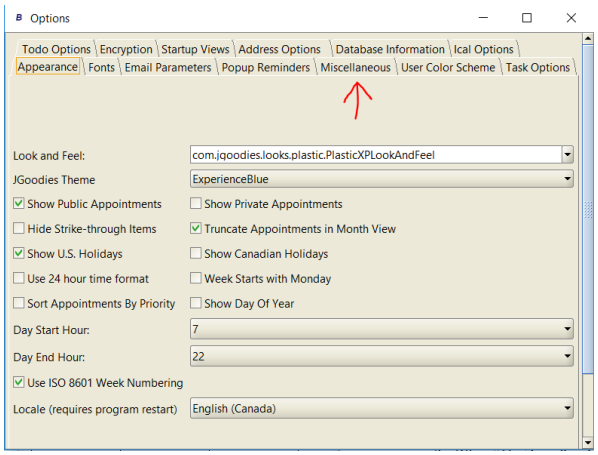


Figure 6: Selecting the Miscellaneous Tab

4. Change Socket Port from 2929 to -2929 and press apply.

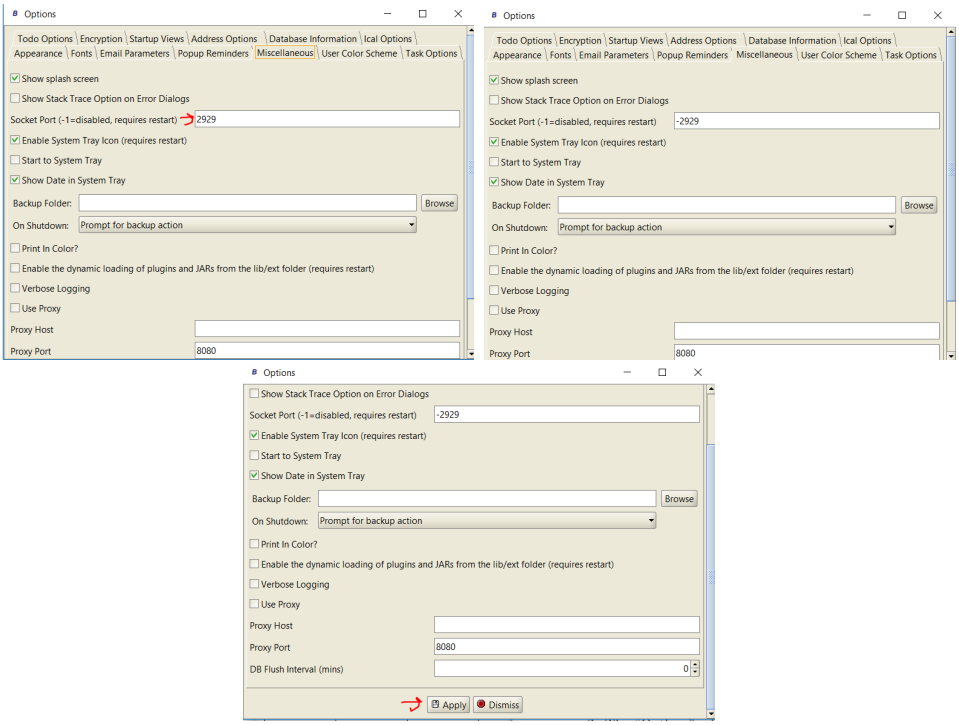


Figure 7: Changing the port (Part 1)

5. Restart BORG.

Results

* Expected Results: Error message prompting the use of a valid socket port.
* Real Results: Unable to run the application and thus not able to access calendar data.
    1. Unable to run BORG after application restart
    2. Unable to run BORG after clean uninstall and re-install
    3. Unable to run BORG off a USB
    4. Unable to run BORG after system restart
    5. Unable to run BORG after java re-install

Additional Tests
Goals: To verify if the error only occurs with negative numbers/any port that is not 2929 or exactly with port -2929 or an error that occurs when attempting to change the port at all.

1. Attempt to reproduce with port 20. No errors.
2. Attempt to reproduce with port 1. No errors.
3. Attempt to reproduce with port -1. No errors.
4. Attempt to reproduce with port -20. Bug occured.
5. Attempt to reproduce by changing the port to -2929, applying the changes, and then setting it back to 2929 and applying changes. Bug occured.
6. Attempt to reproduce with port 65536. Bug occured.

– **New or old bug**: New

2.2 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** - 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 the 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.

  The method did not specifity 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 after analyzing structure of the method.

Table 1: Input ECT

M1: {mins | mins % 60 = 0}
M2: {mins | mins % 60 = 1}
M3: {mins | mins % 60 > 1}
H1: {mins | mins / 60 ≥ 1 AND mins / 60 < 2}
H2: {mins | mins / 60 ≥ 2}
H3: {mins | mins / 60 ≥ 0 AND 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

- **Evaluation** : The tests are shown below suitable for strong normal equivalence class testing technique since it covers the all the range of outputs for valid inputs and invalid inputs (negative integers) are not tested due to lack of specfication information regarding these values.

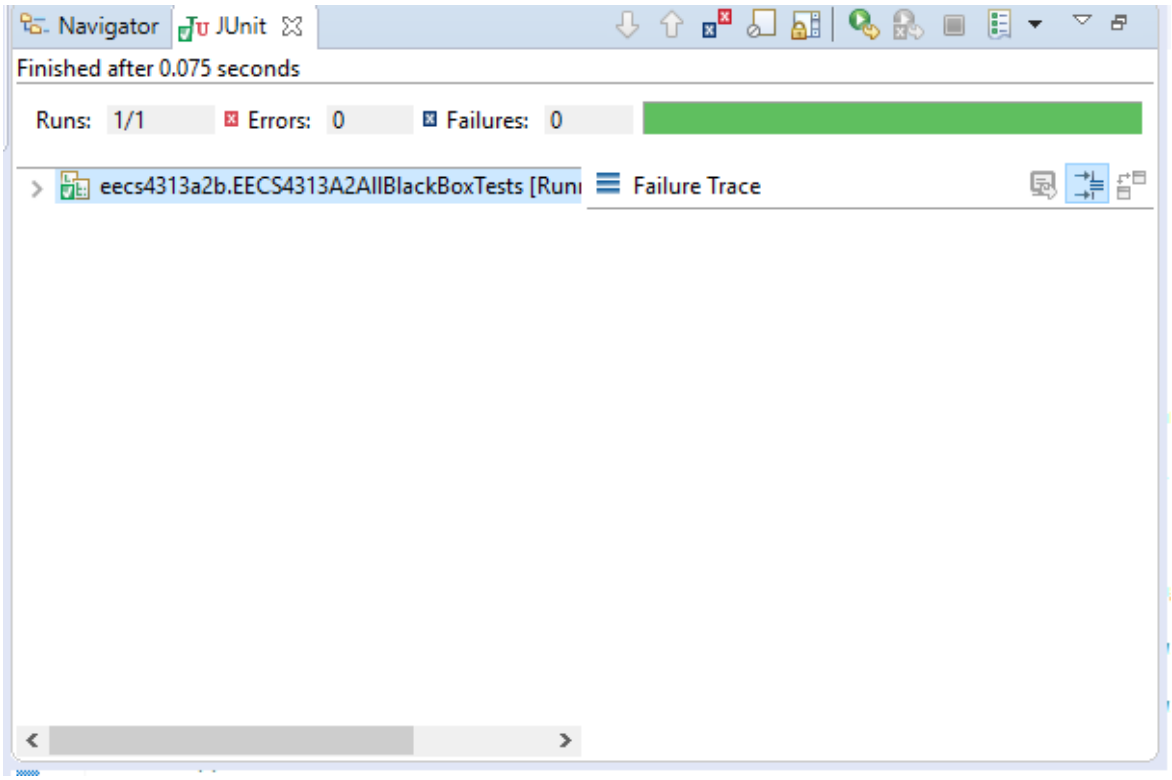


Figure 8: Test results using Strong Normal ECT

2.2.1 Testing Code

```
package eecs4313a2b;

import static org.junit.Assert.*;

import org.junit.Test;

import net.sf.borg.common.DateUtil;

public class EECS4313A2AllBlackBoxTests {

  @Test
  public void testMinuteString() {

    // Hour
    // Class 1: Mins/60 = 1 and Mins\%60 = 0 - Testing 1 hour
       [Range: [1]]
    assertEquals("1 Hour", DateUtil.minuteString(60));
    // Class 2: Mins/60 = 1 and Mins\%60 = 1 -Testing 1 hour with
       1 minute [Range:[1] hour and [1] minute]
    assertEquals("1 Hour 1 Minute", DateUtil.minuteString(61));
    // Class 3: Mins/60 = 1 and Mins\%60 > 1 -Testing 1 hour with
       some minutes [Range:[1] hour and (1,59] minutes]
    assertEquals("1 Hour 15 Minutes", DateUtil.minuteString(75));
    // ------------------------------------------------------
    // Hours
    // Class 4: Mins/60 > 1 and Mins\%60 = 0 -Testing hours more
       than one [Range:(1, infinity) hours]
    assertEquals("3 Hours", DateUtil.minuteString(180));
    // Class 5: Mins/60 > 1 and Mins\%60 = 1 -Testing hours more
       than 1 hour with some minutes [Range:(1,infinity) hours
       and [1] minute]
    assertEquals("2 Hours 1 Minute", DateUtil.minuteString(121));
    // Class 6: Mins/60 > 1 and Mins\%60 > 1 -Testing hours more
       than 1 hour with some minutes [Range:(1,infinity) hours
       and (1,59] minutes]
    assertEquals("2 Hours 25 Minutes", DateUtil.minuteString(145));
    // ------------------------------------------------------
    // Minutes
```

```
    // Class 7: Mins/60 = 0 and Mins\%60 = 0 -Testing 0 minutes
       [Range:[0] minute]
    assertEquals("0 Minutes", DateUtil.minuteString(0));
    // Class 8: Mins/60 = 0 and Mins\%60 = 1 -Testing 1 minute
       [Range:[1] minute]
    assertEquals("1 Minute", DateUtil.minuteString(1));
    // Class 9: Mins/60 = 0 and Mins\%60 > 1 - Testing minutes
       that are less than 1 hour [Range: (1,59] minutes]
    assertEquals("50 Minutes", DateUtil.minuteString(50));
  }
}
```

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.

	Rule 1-2	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**	**Rule 8**
C1: d1 < d2	T	T	T	F	F	F	F
C2: d1 = d2	T	F	F	T	T	F	F
C3: d1 > d2	-	T	F	T	F	T	F
A1: Date is after						X	
A2: Date is not after			X		X		
A3: Impossible	X	X		X			X

Table 4: Decision Table for the isAfter method

Rationale: The decision table above outlines 8 rules. The rules are derived from three equivalence classes: d1 is less than d2, d1 is equal to d2 and d1 is greater than d2. Each equivalence class can have 2 different values (i.e T or F), giving us $2^3 = 8$ rules. Out of the 8 rules, 5 of the rules points to *Impossible* cases, hence cannot be converted to test cases. The other 3 rules are converted into test cases. Below is the code snippet and the test run screenshot for the test cases derived using the decision table technique.

2.3.1 Testing Code

```
@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);
}
```

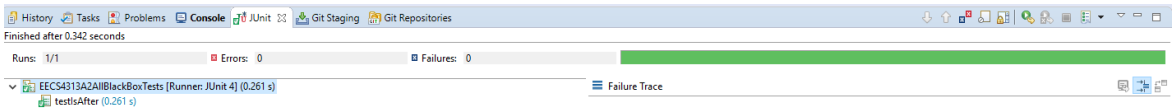


Figure 9: Test results using Decision Table Testing Technique

3 White Box Testing

3.1 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 5: Input ECT

M1: {mins | mins % 60 = 0}
M2: {mins | mins % 60 = 1}
M3: {mins | mins % 60 > 1}
H1: {mins | mins / 60 ≥ 1 AND mins / 60 < 2}
H2: {mins | mins / 60 ≥ 2}
H3: {mins | mins / 60 ≥ 0 AND mins / 60}

Table 6: 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 7: ECT-Matrix

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

The method did not specifity 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 thorugh additional test cases after analyzing structure of the method.

- **Statement Coverage Using Black-Box Methods :** 100% [refer to Figure **??** & **??**]

- **Justification for getting 100% in blackbox tests :** Since the method converts the interger in terms of human readable minutes and hours, the range of valid

outputs can be partition 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.

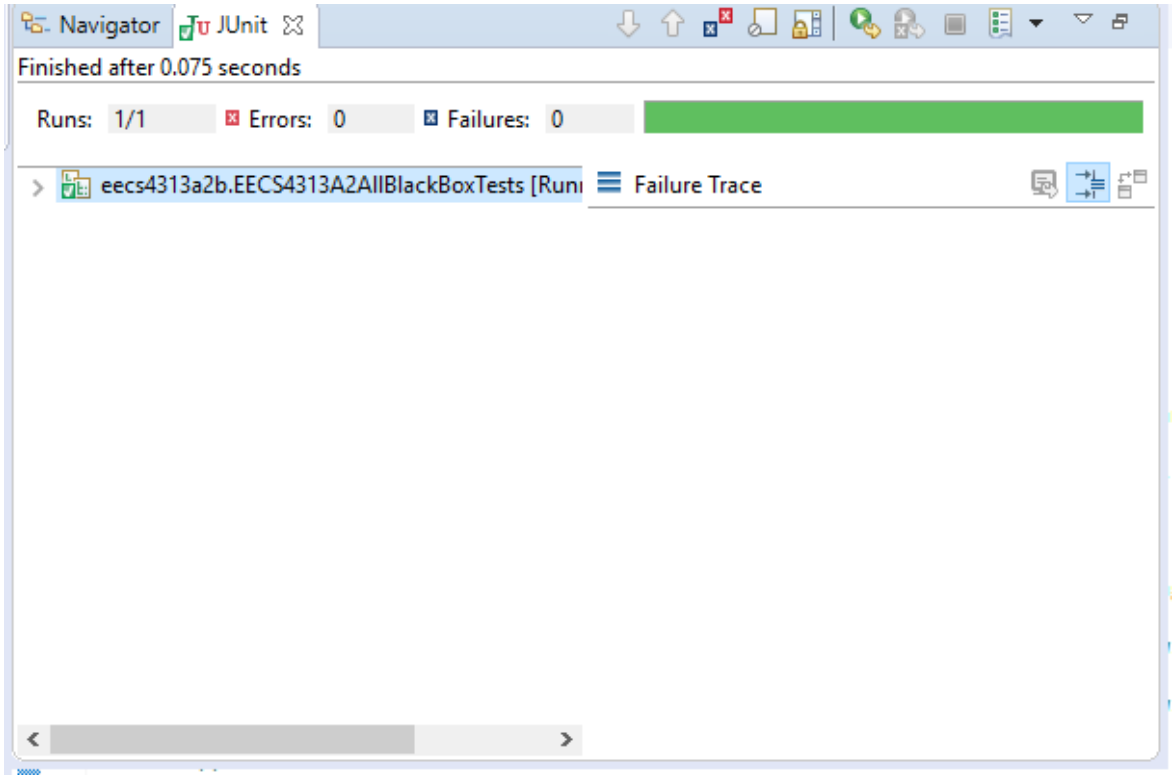


Figure 10: Test run result for the minuteString functionn

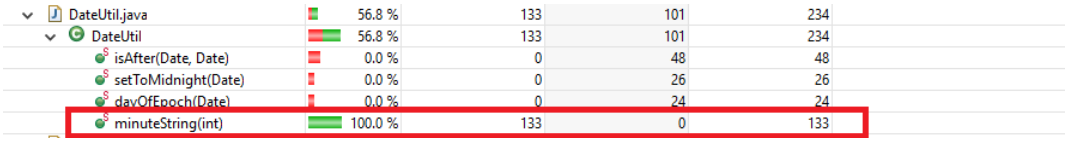


Figure 11: Statement Coverage View for the minuteString function

- **Additional cases:** Since hours are implemented by dividing integer by 60, the negative integers can produce negative hours. Also, negative hours produced emptystring according to the implementation.However, minutes are computed by taking the remainder of the division (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 :

- Mins/60 < 0 and Mins%60 > 1 - To test negative hours with more than 1 minute [Class 10]
- Mins/60 < 0 and 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 **??**.

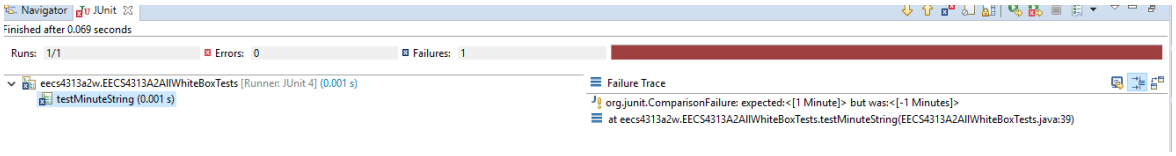


Figure 12: Additional test cases fail due to a bug in the minuteString function

3.1.1 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 inputing 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 inputing 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) mod 60 is 10 minutes.
- Real Results: The actual result produce the bug since it shows -10 as the outputs.
- **New or old bug**: New

3.2 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 innner exception segments 1 and 2 fully covered, a more granuallar 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 13: 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 14: Statement Coverage Metrics for the sendMsg function before additional testcases

```java
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 15: Statement Coverage View for the sendMsg function after adding additional test-cases

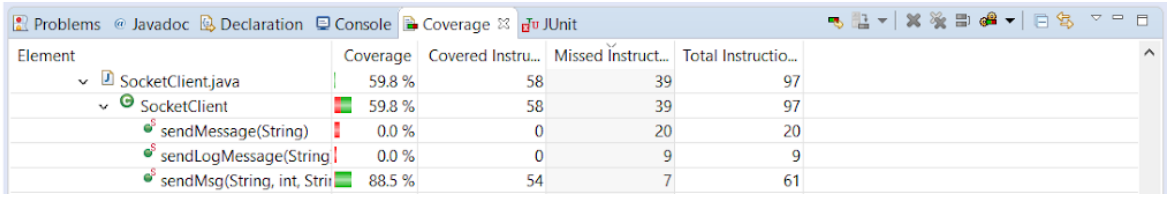
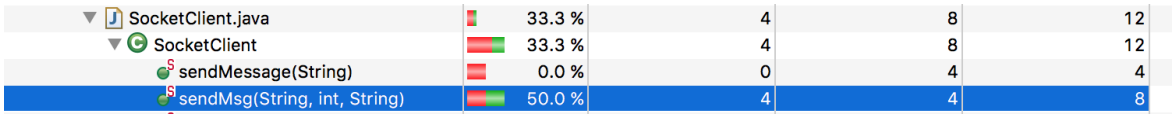
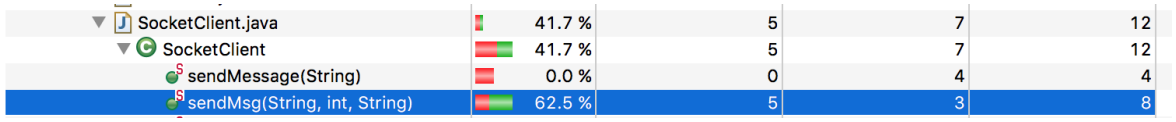


Figure 16: Statement Coverage Metrics for the sendMsg function after adding additional test-cases



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 17: 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 18: Branch Coverage Metrics for the sendMsg function after adding additional testcases

3.3 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.

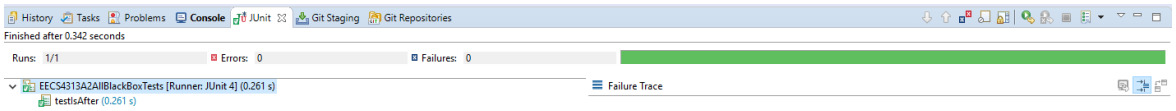


Figure 19: Test run result for the isAfter function

```java
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 20: 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 21: Statement Coverage Metrics for the isAfter function

3.3.1 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 8: CFG Segment Table for the isAfter method

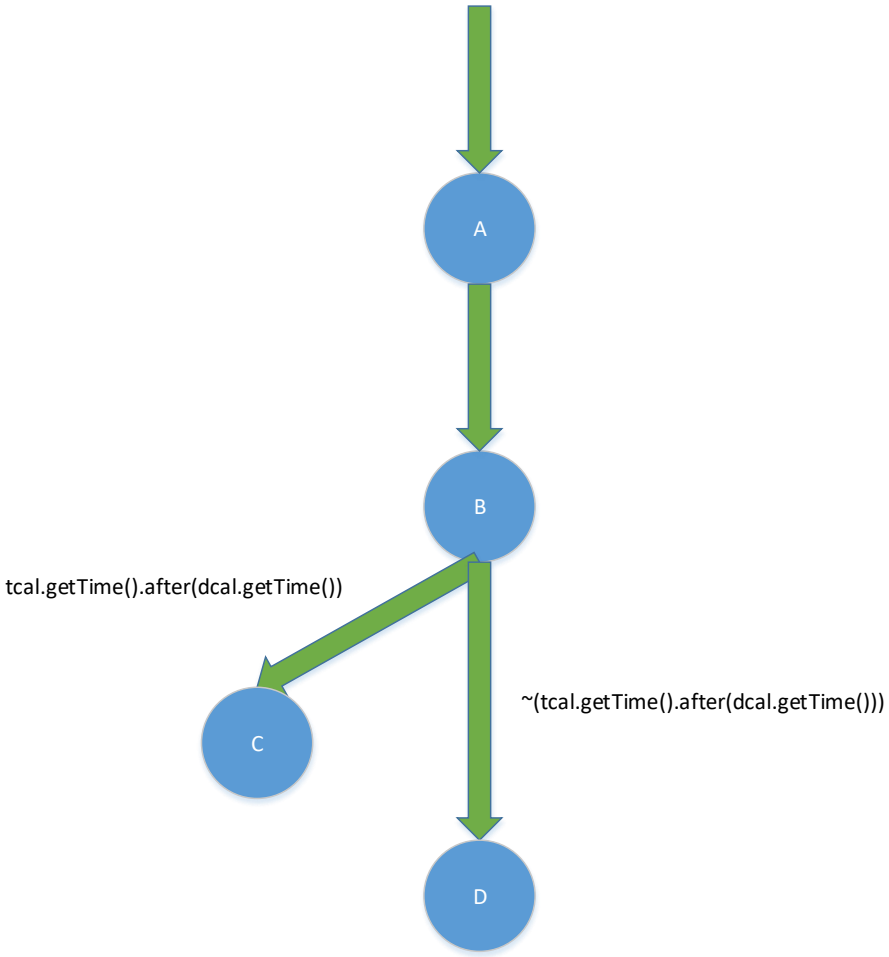


Figure 22: Control Flow Graph for the isAfter function

- **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 **??**, 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 **??**, 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);
```