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. 3 and Listing. 1 for weak normal testing.
  Refer to Figure. 1, Figure. 2 and Listing 2 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_+
```

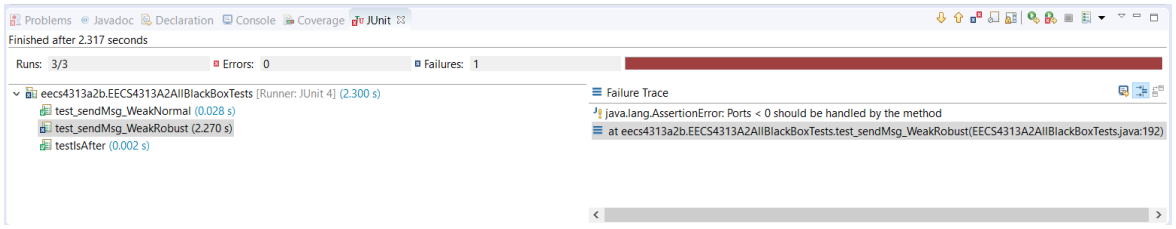

Figure 1: Test results using Weak Robust Boundary Value Testing

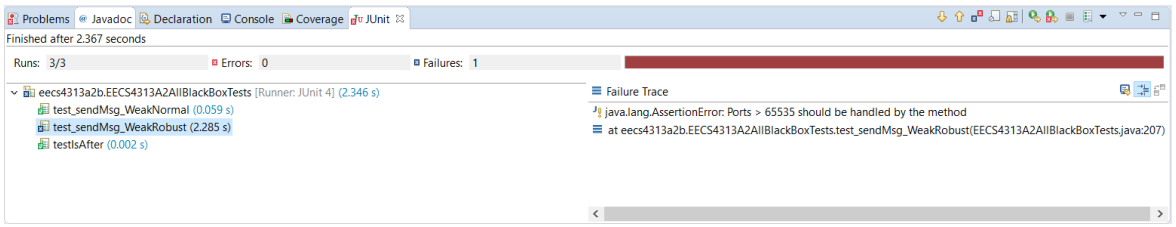

Figure 2: Test results using Weak Robust Boundary Value Testing

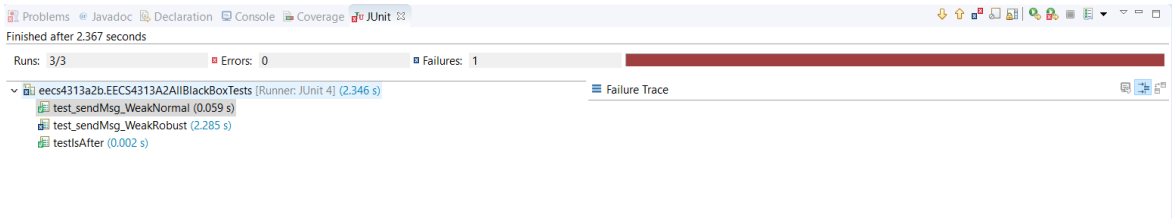

Figure 3: Test results using Weak Normal Boundary Value Testing

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

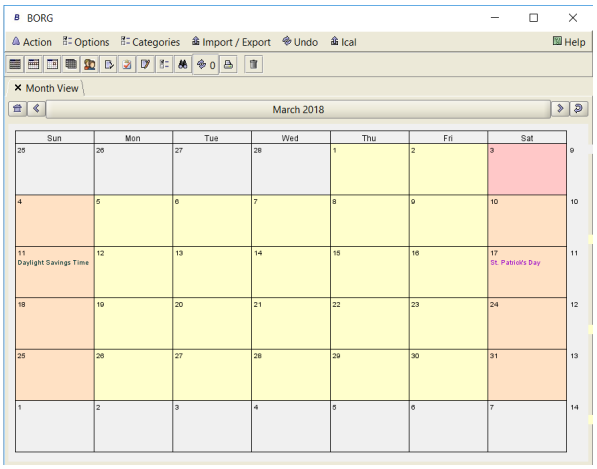

Figure 4: Run the application

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

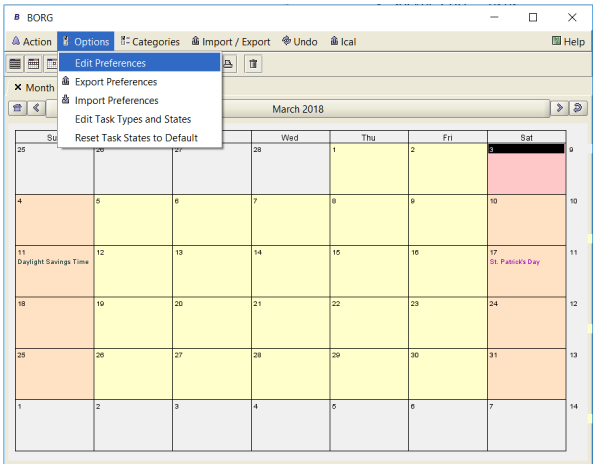

Figure 5: Opening the options window

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

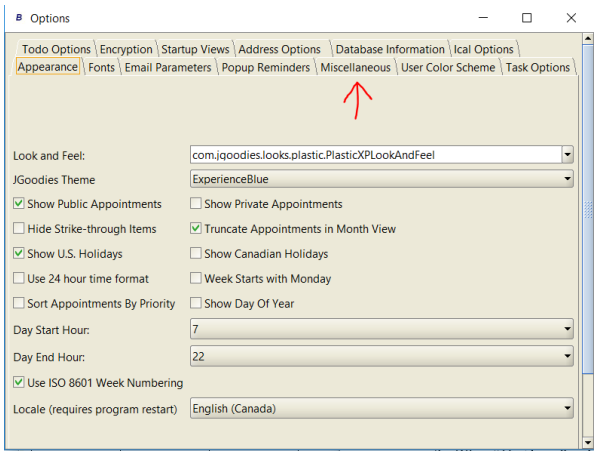

Figure 6: Selecting the Miscellaneous Tab

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

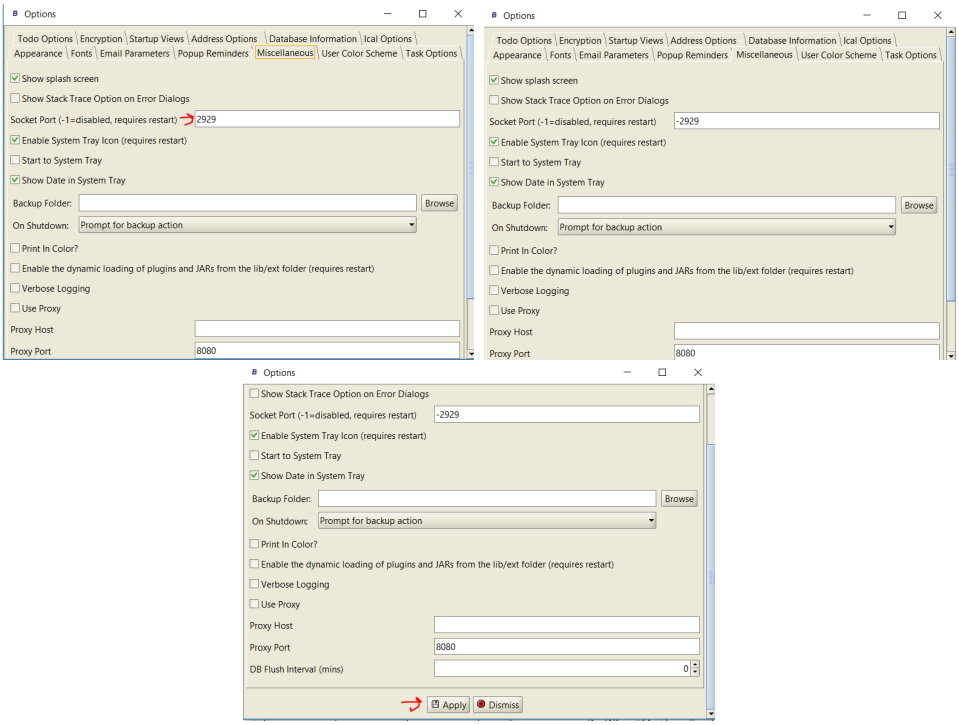

Figure 7: Changing the port (Part 1)

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** - The argument is an integer
- **Justification**: Equivalence class testing is suitable for this method since argument of this method is an integer which is an independent variable and the entire range of input 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 paritions for this method are:
  - **Mins / 60 = 1 and Mins % 60 = 0 :** To test 1 hour.
    - Range of hours: [1]
    - Range of minutes: [0]
  - **Mins / 60 = 1 and Mins % 60 = 1 :** To test the 1 hour with 1 minute.
    - Range of hours: (1,+∞)
    - Range of minutes: [1]
  - **Mins / 60 = 1 and Mins % 60 > 1 :** To test 1 hour with minutes more than 1 minute.
    - Range of hours: [1]
    - Range of Minutes : (1, 59]

- **Mins / 60 > 1 and Mins % 60 = 0 :** To test the hours more than 1 hour.
  - Range of hours: (1,+∞)
  - Range of minutes: [0]
- **Mins / 60 > 1 and Mins % 60 = 1 :** To test hours more than 1 hour with 1 minute.
  - Range of hours: (1,+∞)
  - Range of Minutes :[1]
- **Mins / 60 > 1 and Mins % 60 > 1 :** To test the hours more than 1 hour with minutes more than 1 minute.
  - Range of hours: (1,+∞)
  - Range of Minutes : (1, 59]
- **Mins / 60 = 0 and Mins % 60 = 0 :** To test 0 minutes.
  - Range of hours: [0]
  - Range of minutes: [0]
- **Mins / 60 = 0 and Mins % 60 = 1 :** To test 1 minute.
  - Range of hours: [0]
  - Range of minutes: [1]
- **Mins / 60 = 0 and Mins % 60 > 1 :** To test minutes more than 1 minute and less than 60 minutes or 1 hour.
  - Range of hours: [0]
  - Range of minutes: (1,59]

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.

- **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.

- **Class 1:** Mins / 60 = 1 and Mins % 60 = 0
- **Class 2:** Mins / 60 = 1 and Mins % 60 = 1
- **Class 3:** Mins / 60 = 1 and Mins % 60 > 1
- **Class 4:** Mins / 60 > 1 and Mins % 60 = 0
- **Class 5:** Mins / 60 > 1 and Mins % 60 = 1
- **Class 6:** Mins / 60 > 1 and Mins % 60 > 1
- **Class 7:** Mins / 60 = 0 and Mins % 60 = 0
- **Class 8:** Mins / 60 = 0 and Mins % 60 = 1
- **Class 9:** Mins / 60 = 0 and Mins % 60 > 1

Positive Minutes (Mins % 60 > 1)			Class 9	Class 3	Class 6
Positive Minute (Mins % 60 = 1)			Class 8	Class 2	Class 5
0 Minutes (Mins % 60 = 0)			Class 7	Class 1	Class 4
Negative Minute (Mins % 60 = -1)					
Negative Minutes (Mins % 60 < -1)					
	Negative Hours (Mins / 60 < -1)	Negative Hour (Mins / 60 = -1)	0 Hours (Mins / 60 = 0)	Positive Hour (Mins / 60 = 1)	Positive Hours (Mins / 60 > 1)

Figure 8: Proving the test cases produce Strong Normal ECT

In the Figure 1, The boxes colored in green represent all the valid output results and red boxes represent the invalid results which cannot be tested since the method did not specify how the negative integers should be treated. The Class 1,Class 2....Class 9 represents the partitions that we derived from how to convert the integers into readable string.

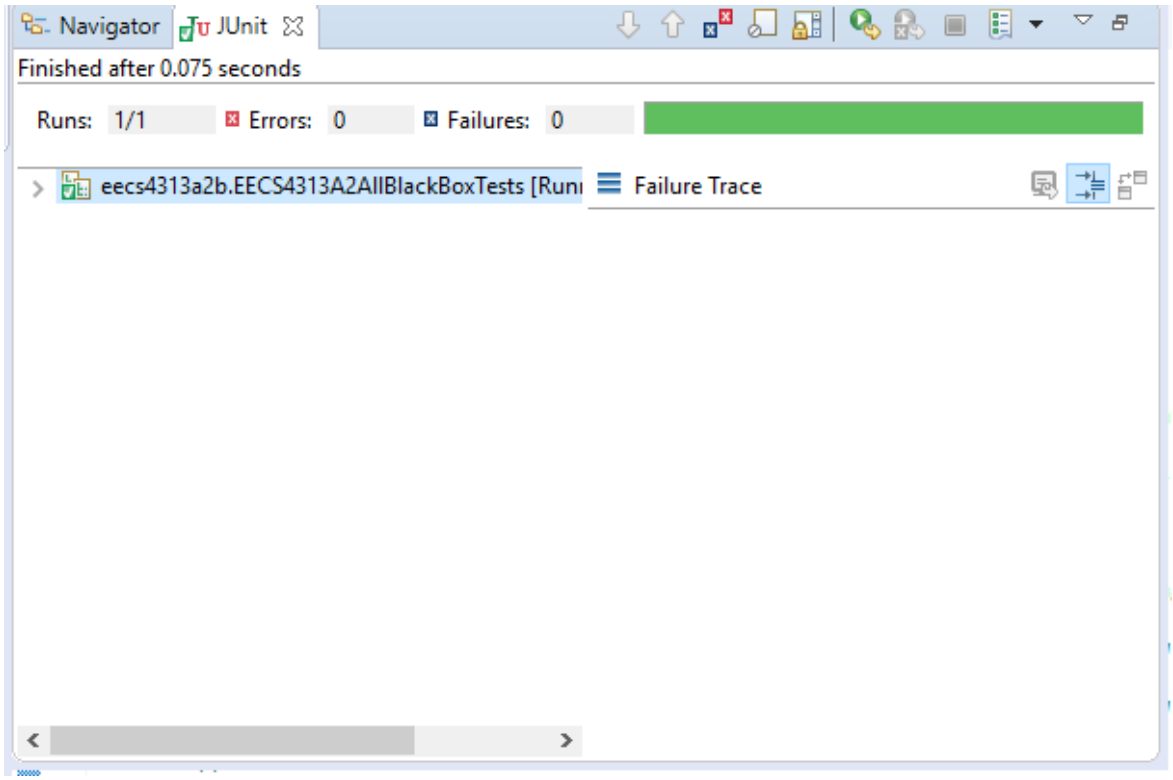

Figure 9: Test results using Strong Normal ECT

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 1: 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);
}
```

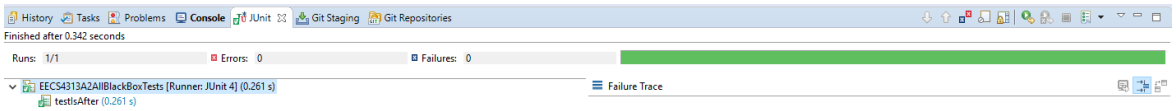

Figure 10: Test results using Decision Table Testing Technique

3 White Box Testing

- The statement coverage measurements for your Assignment 2 test suite.
- A description of the test cases that you added in this assignment to improve statement coverage. The marker will not read your code in order to see what you tested. You have to describe it.
- The statement coverage measurements for your final submission. Include the screenshots of the test running results and the screenshots of the coverage measurement. If your coverage is not 100
- The Control Flow Graph you created. Indicate the segments clearly (you will probably need to include the code for this).
- The path coverage discussion described in section 2 above.
- 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.
- An appendix with the specification of the methods you are testing (if there are new ones).