EECS 4313 Assignment 3
Data Flow Testing, Slice-Based Testing and Mutation Testing

Student Name — Student Number — EECS Account
Edward Vaisman — 212849857 — eddyv
Robin Bandzar — 212200531 — cse23028
Kirusanth Thiruchelvam — 212918298 — kirusant
Sadman Sakib Hasan — 212497509 — cse23152

March 25, 2018

Contents

1 BORG Calendar

1.1 Mutation Testing

Mutation tests were run using the previous unit test suite that we created for assignment 2. The program used to run the Mutation tests was *Eclispe* with the *Pitclipse* plugin. The three methods that we tested are listed with their results in the following subsections.

1.1.1 minuteString Function

```
100         public static String minuteString(int mins) {
101
102 1               int hours = mins / 60;
103 1               int minsPast = mins % 60;
104
105                 String minutesString;
106                 String hoursString;
107
108 2               if (hours > 1) {
109                         hoursString = hours + " " + Resource.getResourceString("Hours");
110 2               } else if (hours > 0) {
111                         hoursString = hours + " " + Resource.getResourceString("Hour");
112                 } else {
113                         hoursString = "";
114                 }
115
116 2               if (minsPast > 1) {
117                         minutesString = minsPast + " " + Resource.getResourceString("Minutes");
118 2               } else if (minsPast > 0) {
119                         minutesString = minsPast + " " + Resource.getResourceString("Minute");
120 2               } else if (hours >= 1) {
121                         minutesString = "";
122                 } else {
123                         minutesString = minsPast + " " + Resource.getResourceString("Minutes");
124                 }
125
126                 // space between hours and minutes
127 2               if (!hoursString.equals("") && !minutesString.equals(""))
128                         minutesString = " " + minutesString;
129
130 1               return hoursString + minutesString;
131         }
```

Figure 1: Code for the minuteString method

102	1. Replaced integer division with multiplication → KILLED
103	1. Replaced integer modulus with multiplication → KILLED
108	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
110	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
116	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
118	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
120	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
127	1. negated conditional → KILLED 2. negated conditional → KILLED
130	1. mutated return of Object value for net/sf/borg/common/DateUtil::minuteString to (if (x != null) null else throw new RuntimeException) → KILLED

Figure 2: Mutations for the minuteString method

As one can see in Figures 1 and 2 that the previous tests effectively kill all the mutants so no further changes are needed.

1.1.2 isAfter Function

```
40          public static boolean isAfter(Date d1, Date d2) {
41
42                  GregorianCalendar tcal = new GregorianCalendar();
43  1               tcal.setTime(d1);
44  1               tcal.set(Calendar.HOUR_OF_DAY, 0);
45  1               tcal.set(Calendar.MINUTE, 0);
46  1               tcal.set(Calendar.SECOND, 0);
47                  GregorianCalendar dcal = new GregorianCalendar();
48  1               dcal.setTime(d2);
49  1               dcal.set(Calendar.HOUR_OF_DAY, 0);
50  1               dcal.set(Calendar.MINUTE, 10);
51  1               dcal.set(Calendar.SECOND, 0);
52
53  1               if (tcal.getTime().after(dcal.getTime())) {
54  1                       return true;
55                  }
56
57  1               return false;
```

Figure 3: Code for the isAfter method

```
43  1. removed call to java/util/GregorianCalendar::setTime → KILLED
44  1. removed call to java/util/GregorianCalendar::set → SURVIVED
45  1. removed call to java/util/GregorianCalendar::set → SURVIVED
46  1. removed call to java/util/GregorianCalendar::set → SURVIVED
48  1. removed call to java/util/GregorianCalendar::setTime → KILLED
49  1. removed call to java/util/GregorianCalendar::set → SURVIVED
50  1. removed call to java/util/GregorianCalendar::set → SURVIVED
51  1. removed call to java/util/GregorianCalendar::set → SURVIVED
53  1. negated conditional → KILLED
54  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
57  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
```

Figure 4: Mutations for the isAfter method

Looking at the result of the mutation test result from Figure 4, it is inevitable that the method *isAfter* will still pass tests even after removing the lines 44-46 and 49-51. This means those lines (line 44-46 and line 49-51) can never have any affect on the expected output of the method.

This can be proved by adding a simple test case which creates two date objects and set their minute, hour and second to some value. No matter what value for minute, hour and second is passed with the Date object, the function will always create a Gregorian Calendar instance with values of 0 for the first date object. Similarly, it will create another Gregorian Calendar instance with values 0 for hours and seconds and 10 for minutes for the second date object.

Example test case:

```
@Test
public void testIsAfterBug() {
  // Both happens at the same day, but at different hours
  // but the implementation ignores the hours, minutes
  // and seconds
  Date d1 = new Date( 2018 , 8 , 24 , 8 , 30 , 24 );
  Date d2 = new Date( 2018 , 8 , 24 , 9 , 30 , 24 );

  // This works correctly. The assert returns true
  assertFalse(DateUtil.isAfter(d1, d2));
  // This does not work correctly. The assert returns false,
  // when it should be true
  assertTrue(DateUtil.isAfter(d2, d1));
}
```

Bug Report:

- **Bug Report Title**: The *isAfter* function ignores hours, minutes and seconds fields from the passed in date object.
- **Reported by**: Sadman Sakib Hasan
- **Date reported**: Saturday, March 24, 2018
- **Program (or component) name**: BORG Calendar version 1.8.3
- **Configuration(s)**:
  System Info

  - Operating System: Windows 10 Pro 64-bit (10.0, Build 16299)
  - Language: English (Regional Setting: English)
  - System Manufacturer: Hewlett-Packard
  - System Model: HP 15 TouchSmart Notebook PC
  - BIOS: F.10
  - Processor: AMD A6-5200 APU with Radeon(TM) HD Graphics (4 CPUs), 2.0GHz
  - Memory: 6144MB RAM
  - Java Version: 1.8.0_151-b12

- **Report type**: Coding Error
- **Reproducibility**: 100%
- **Severity**: Medium
- **Problem summary**: The *isAfter* method automatically sets the HOUR_OF_DAY, MINUTE, and SECOND values of the newly created Gregorian Calendar object from the passed in dates d1 and d2 to a constant value.
- **Problem description**: The *isAfter* method in *net.sf.borg.common.DateUtil* does not take into account the hours, minutes, and seconds of the given date objects when comparing the two dates. A newly Gregorian Calendar object is created for both argument, d1 and d2, and the hours, minutes and seconds are set to 0 for *d1*'s calendar instance whereas the hours and seconds are set to 0 and minute set to 10 for *d2*'s calendar instance. This causes an issue when there are comparison between two dates that are in the same day, and hours, minutes and seconds differ as shown in the example test case 1.1.2.
- **Steps to Reproduce**:
  - Create two date objects, d1 and d2, with the day set to the same value and hours, minutes and seconds set to different values.
  - Compare the two dates using *isAfter* function.
  - The *isAfter* function will always return false regardless of the fact if d1 is ahead on hours/minutes/seconds or vice versa.
- **New or old bug**: New

1.1.3 sendMsg Function

```
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 1                       sout.println(msg);
42                          line = sin.readLine();
43                          // Check if connection is closed (i.e. for EOF)
44 1                       if (line == null) {
45                                  log.info("Connection closed by server.");
46                          }
47                  } catch (IOException e) {
48 1                       if (s != null)
49 1                               s.close();
50                          throw e;
51                  }
52                  // Always be sure to close the socket
53                  finally {
54                          try {
55 2                               if (s != null)
56 2                                       s.close();
57                          } catch (IOException e2) {
58                                  // empty
59                          }
60                  }
61
62 1                return line;
63          }
```

Figure 5: Code for the sendMsg method

```
41  1. removed call to java/io/PrintStream::println → TIMED_OUT
44  1. negated conditional → SURVIVED
48  1. negated conditional → TIMED_OUT
49  1. removed call to java/net/Socket::close → NO_COVERAGE
55  1. negated conditional → SURVIVED
    2. negated conditional → KILLED
56  1. removed call to java/net/Socket::close → SURVIVED
    2. removed call to java/net/Socket::close → TIMED_OUT
62  1. mutated return of Object value for net/sf/borg/common/SocketClient::sendMsg to ( if (x != null) null
    else throw new RuntimeException ) → KILLED
```

Figure 6: Mutations for the sendMsg method

The results show that not all mutants have been killed. From Figure 5 we can see that our mutation testing results can be possibly improved if more tests on the server and socket state are created.

After Additional Test Cases

```
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 1                        sout.println(msg);
42                          line = sin.readLine();
43                          // Check if connection is closed (i.e. for EOF)
44 1                        if (line == null) {
45                                  log.info("Connection closed by server.");
46                          }
47                  } catch (IOException e) {
48 1                        if (s != null)
49 1                                s.close();
50                          throw e;
51                  }
52                  // Always be sure to close the socket
53                  finally {
54                          try {
55 2                                if (s != null)
56 2                                        s.close();
57                          } catch (IOException e2) {
58                                  // empty
59                          }
60                  }
61
62 1                return line;
63          }
```

Figure 7: Updated code results for the sendMsg method

The test case shown below was added and improved results to what is shown in Figures 7 and 8. This test case kills the mutant at lines 44, 48 without increasing line coverage. Thus, showing how test cases built for coverage alone are not sufficient.

The remaining bugs all relate to the *close* function of the Socket object instantiated in the *sendMsg* method. When Pitclipse mutates the *close* function and removes it, this action causes a timeout. Which is why the *sendMsg* method is defensively

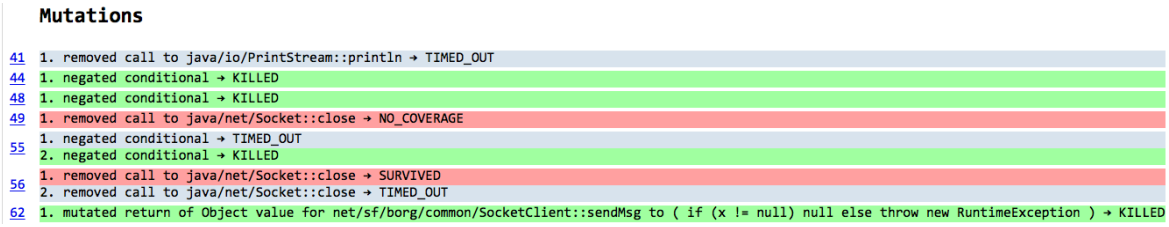

Figure 8: Updated mutations for the sendMsg method

programmed to ensure that the socket connection is closed after the message is sent. Due to the amount of defensive programming and case of timeouts it is not possible to achieve full mutation testing coverage for this method consistently between all machines.

Listing 1: Additional test case for sendMsg

```
@Test
public void checkServerAlive() {
String msg = null;
SocketServer ss;
String response;
   try {
      ss = new SocketServer(2922, this);
      response = SocketClient.sendMsg("localhost", 2922, msg);
      assertTrue(ss.isAlive());
   } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
   }
}
```

- The data flow analysis you performed and the calculation of the coverage metrics. You must show which test cases are responsible for which dc-paths.
- A description of the test cases you added to improve coverage. If your coverage was already high, discuss how your testing was able to achieve this.
- The slices that you identified and the percentage of slices that your testing covers. You must show which test cases are responsible for which slices.
- A description of the test cases you added to improve slice coverage. If your coverage was already high, discuss how your testing was able to achieve this.

- Evaluate the effectiveness of your test cases using mutation testing. Discuss and address any issues if you have found in your written report.
- 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

2 JPetStore

- The test scenarios that you have created;
- The request rates and the duration of the load tests;
- The analysis of your load tests and the description of any problems that you have found (if there are any).