EECS 4313 Assignment 2

Black-box and White-box Testing with JUnit

Anton Sitkovets (212118048)
Mina Zaki (212857975)
Jeremi Boston (212432399)
David Iliaguiev (210479830)

Black Box Report

1. Specification of the selected Java methods

1.1 common > DateUtil > isAfter

public static boolean isAfter(Date d1, Date d2)
This method checks if one date happens on a later date than the other.
When d1 occurs after d2, the return value is true.
When d2 occurs after d1, the return value is false.

- The first argument is the first date in the comparison
- The second argument is the second date in the comparison

1.2 common > DateUtil > MinuteString

public static String minuteString(int mins)
This method takes a number of minutes and returns a human readable representation in hours and minutes.
The return values will look like this for 62 minutes: 1 Hour 2 Minutes.
The return value will look like this for 60 minutes: 1 Hour
The return value will look like this for 12 minutes: 12 Minutes

- The parameter mins is the number of minutes to convert to a minute string

1.3 model > Repeat > getDayList

static public Collection<Integer> getDaylist(String f)
This method takes a string formatted as “dlist xxxx” or “dlist, xxxx”, where xxxx refers to some set of numbers(1-7) and returns a list containing a set of the specified numbers. Here if the user wishes to specify that the appointment repeats on Sunday, Monday and Friday, they can replace xxxx with 126. The numbers coincide with the day of the week where Sunday is 1 and Saturday is 7.
If the f string is “dlist 1234516”, the return value will be a list of [1, 2, 3, 4, 5, 6]. Note that repeated values do not get added multiple times.

- The parameter is for the frequency string which specifies how often an appointment should be repeated per week.

2. Justification of the testing technique chosen.

2.1 common > DateUtil > isAfter

isAfter is a two variable Boolean function which checks if a date occurs after another date. For this method, weak robust equivalence class testing was employed. Equivalence class testing is best used in cases where the function being tested has one or more variables with well defined intervals. Both conditions are met by the *isAfter* function as the variables, dates, are well defined and therefore requires two of them for the function. Furthermore, both dates are independent thus this assures non-redundancy within our test cases and a strong equivalence. Weak robust equivalence was chosen over other variations of ECT as it provides a sense of completeness in testing while also being more reasonable. Strong robustness requires an unreasonable amount of test case which may not provide useful information. For instance, one can imply certain variables do not work if it is not possible to test them within Java. Thus, being a complex function with multiple well defined variables, *isAfter* is well suited for weak robust equivalence class testing.

2.2 common > DateUtil > MinuteString

minuteString is a single variable function which converts time from a minute value to a time value of hours and minutes. This function's only variable is the integer mins, to be converted to hours and minutes. As a result of being a non complex function, minuteString is more suited for boundary value testing than equivalence class testing. The specific extension of boundary value testing employed is robustness testing. The extension will allow for the testing of input values outside and at the edge of the inputs domain to be tested. These extreme input values are often the cause of errors in the systems functionality, and thus this extension is effective at exposing potential user input problems. The effectiveness of robustness testing can also be attributed to the addition of two more values per variable, max+ and min-, these extra variables provide an extra scope to the testing that other BVT extensions do not posses. Furthermore, robustness testing makes for simple and few test cases as opposed to other BVT extensions. For example, robust worst case and worst case testing reject the single fault assumption resulting in a copious amount of test cases. Not only are the other BVT extensions test cases overabundant, but also resource and time intensive, often times near impossible to accomplish. Meaning, robust testing is the more cost efficient option providing better results for test cases made.

2.3 model > Repeat > getDayList

getDayList is a single variable function that contains many conditions of which each can be matched to several actions. These conditions and actions can easily be converted to a decision table and for this reason decision table testing is the chosen testing method. This allows for the multiple independent conditions to be displayed and associated in a neat way. Furthermore, decision table testing works well with applications where a cause and effect relationship exists between the input and the output of the function. Another condition required for decision table testing that *getDayList* covers, is that the order in which the predicates are evaluated do not affect the resulting actions or rules. Decision table testing was chosen over boundary value testing as the getDayList uses logical variables, strings of values forming a date, which do not work well for this method of testing. Similarly, equivalence class testing does not work well with logical variables and is best used on complex multi variable functions whom have well defined variables, which getDayList does not.

3. Description of your application of the three testing strategies.

3.1 common > DateUtil > isAfter

For this function we will be testing using two date values. The date variables are created with three values in the order of year, month, and date. Using the assertTrue and assertFalse methods, numerous ranges will be tested. The following are the ranges used and their according assert tests:

- first and second date are real dates and the first date is before the second
    - assertFalse(DateUtil.isAfter(d1, d2));
- first and second date are real, and the first date is after second
    - assertTrue(DateUtil.isAfter(d2, d1));
- first date is not a real date and is after the second date, and the second date is a real date
    - assertTrue(DateUtil.isAfter(invalid_date, pre_invalid_date));
- first date is a real date and before the second date, and the second date is not real
    - assertFalse(DateUtil.isAfter(pre_invalid_date, invalid_date));
- first date is real and after the second date, and the second date is not real
    - assertTrue(DateUtil.isAfter(d2, invalid_date));
- first date is not a real date and before the second, and the second date is real
    - assertFalse(DateUtil.isAfter(invalid_date, d2));

Furthermore, the following are the variables employed in the previous test cases and their values;

D1 is a date variable representing August 15th 2017, D2 is a date variable represents the following day which is August 16th 2017, Invalid_date is an invalid date representing February 31st 2017, and Pre_invalid_date is the date of January 1st 2017.

3.2 common > DateUtil > MinuteString

Robustness boundary value testing implies test cases be written so that the "min" input variable of the minuteString function assumes the several boundary values. In the case of robustness testing, two extra boundaries, min- and max+, are added on top of the normal required boundaries. The following are the various boundaries and their assigned values for the testing of this function:

- Min-: -1 minutes
- Min: 0 minutes
- Min+: 1 minute
- Nominal: 1073741823 minutes
- Max-: 2147483646 minutes
- Max: 2147483647 minutes
- Max+: 2147483648 minutes

Each of these values were assigned to the "mins" input variable, and used to see if the minuteString function returned the appropriate time in hours and minutes. For example, the nominal value represents 17895697 hours and 3 minutes. Thus, assigning the nominal value to the "mins" input variable should return the same value, and as a result leads to the following testcase assertEquals("17895697 Hours 3 Minutes", DateUtil.minuteString(1073741823)). Applying the same checks using all of the boundary values result in a complete boundary value robustness testing. Here, -1 minutes was used for the lower boundary Min- because for the use case of this function, it does not make sense to have negative minutes so the minimum value is 0 minutes.

3.3 model > Repeat > getDayList

Using the configurations of getDayList, each condition and action has been mapped to a decision table, as can be seen in figure 1. To test this function, an array list was created with the intention of matching its values with the expected output of the getDayList function. Each condition was then used on getDayList to determine if it returned its expected output. For example, the test case used in the testing of mapping the C1 condition to action A1 is assertEquals(sq, Repeat.getDaylist("")). The getDaylist function is called with an empty input, as per condition C1, and sq is an empty arraylist representing the action A1. This same methodology is employed to test the various mappings that the getDayList function offers. However, special care was taken in the case of condition C5, as every permutation of the set of values from [1-7] needed to be tested. This stems from the fact that the input variable of getDayList can be any sequence of numbers from “1” to “1234567”. In order to create each sequence of numbers for testing, the java class AllPermuations was employed. Using the

sequences created by AllPermutations, test cases were created. The test cases compared the created sequences to an arraylist containing the same integer values. These tests ensured that every possible sequence of strings, within the range, were tested.

C1: Input is empty	T	F	F	F	F	F	F	F	F	F	F
C2: Input has a comma separator between "dlist" and the day values	F	T	F	F	F	T	F	T	F	T	F
C3: Input doesn't have a comma separator between "dlist" and the valid day values	F	F	T	F	F	F	T	F	T	F	T
C4: Input has "dlist" but no numbers afterwards	F	F	F	T	F	F	F	F	F	F	F
C5: Input contains a valid sequence of numbers	F	F	F	F	T	T	T	F	F	F	F
C6: Non number after "dlist"	F	F	F	F	F	F	F	T	T	F	F
C7: Non number in	F	F	F	F	F	F	F	F	F	T	T

between numbers											
A1: Return empty list	X			X				X	X		
A2: Return a list of the extracted numbers		X	X		X	X	X			X	

Figure 1. getDayList decision table.

4. Evaluation of the test cases

For each function, except minuteString, the test suites cover all of the required cases for the given testing methods used. The exception within minuteString is due to the fact that the max+ value is too large, and cannot be used within Java. However, the testing suites of each function can always be strengthened with the use of special value testing. In the case of minuteString, one can add complementary test cases using minute values less than -59. The range used for this special value test were discovered during testing. It was noticed that values less than -59 will only give a negative minutes value but not a negative hours values. In contrast, special value testing is not as effective at completing the tests used for the isAfter function. This difference is due to the fact that the values used in the isAfter function do not have to be valid for the function to work, thus it is futile to test unique dates which break other calendars. Furthermore, the typical unique dates for leap years are already tested within the current testing suite of isAfter. Finally, the test cases used for the getDayList function cover all of the possible conditions and their resulting actions. This function also does not entirely benefit from special value testing due to the fact that if any value besides a permutation of 1-7 is given after dlist the function simply does not return anything. Thus there aren't likely any tricky values that can be used for testing, and each permutation is already tested by the suite. However, the capacity of the function can be tested, such as inputting a large amount of values after dlist. Thus, each test suite successfully tests each of the functions within the confines of the employed testing methods, and in some cases the functions can benefit from additional special value testing.

White-Box Testing Report

1. Coverage Measurements

For the (original) White-Box tests that were done on the three methods the coverage can be seen in the following tables:

1.1 Instruction Coverage

Method	Coverage (%)	Covered Instructions	Missed Instructions	Total Instructions
isAfter(Date, Date)	100.0	48	0	48
minuteString(int)	86.1	99	16	115
getDaylist(String)	92.9	118	9	127

1.2 Statement Coverage

Method	Coverage (%)	Covered Statements	Missed Statements	Total Statements
isAfter(Date, Date)	100.0	13	0	13
minuteString(int)	78.9	15	4	19
getDaylist(String)	95.7	22	1	23

1.3 Branch Coverage

Method	Coverage (%)	Covered Branches	Missed Branches	Total Branches
isAfter(Date, Date)	100.0	2	0	2
minuteString(int)	78.6	11	3	14
getDaylist(String)	90.0	18	2	20

2. Increasing Coverage

2.1 common > DateUtil > isAfter

Seeing as the isAfterTest test method created gave us 100% coverage, the original Black-Box testing covers all instruction, statement and branches. Therefore, there is no changes needed to be made to increase the coverage of the tests. We can see this by the figure below showing all statement being covered in green.

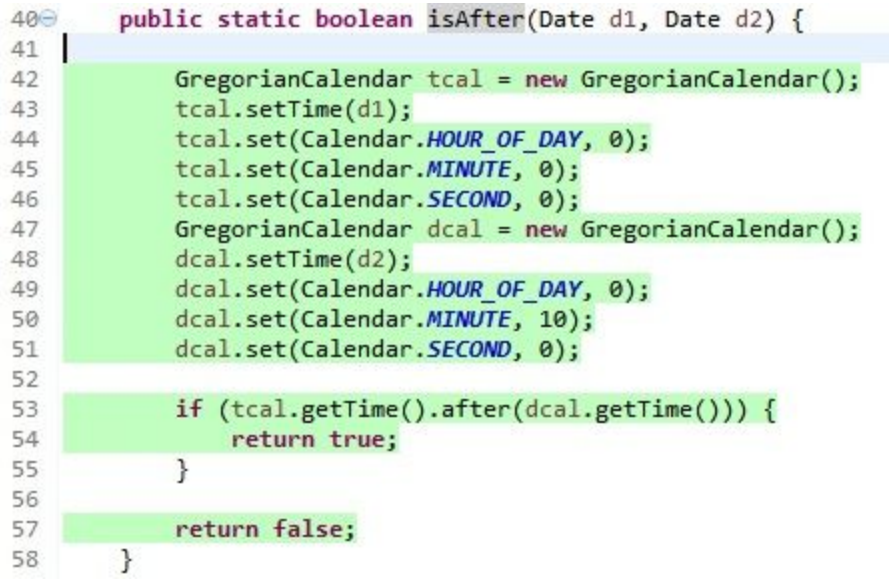

Figure 2: Statement Coverage for isAfter method

2.2 common > DateUtil > MinuteString

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

Figure 3: Statements and Branches covered for minuteString method

For the minuteStringTest test method there is an instruction coverage of 86.1%, but looking at every instruction is not feasible, though something better to look at is statement and branch coverage. The statement coverage attained was 78.9% and for branches 78.6% coverage was attained. With the number of statements missed close to the number of branches missed we can assume that if we cover the branches missed we will cover all the statements. In the figure above we can see the branches missed in yellow and the statements missed in red. The reason we see the else statement in red and its assignment statement in green is because the branch before is not being tested for its “T” true case. We can see this more clearly in the control flow graph of the minuteString method below.

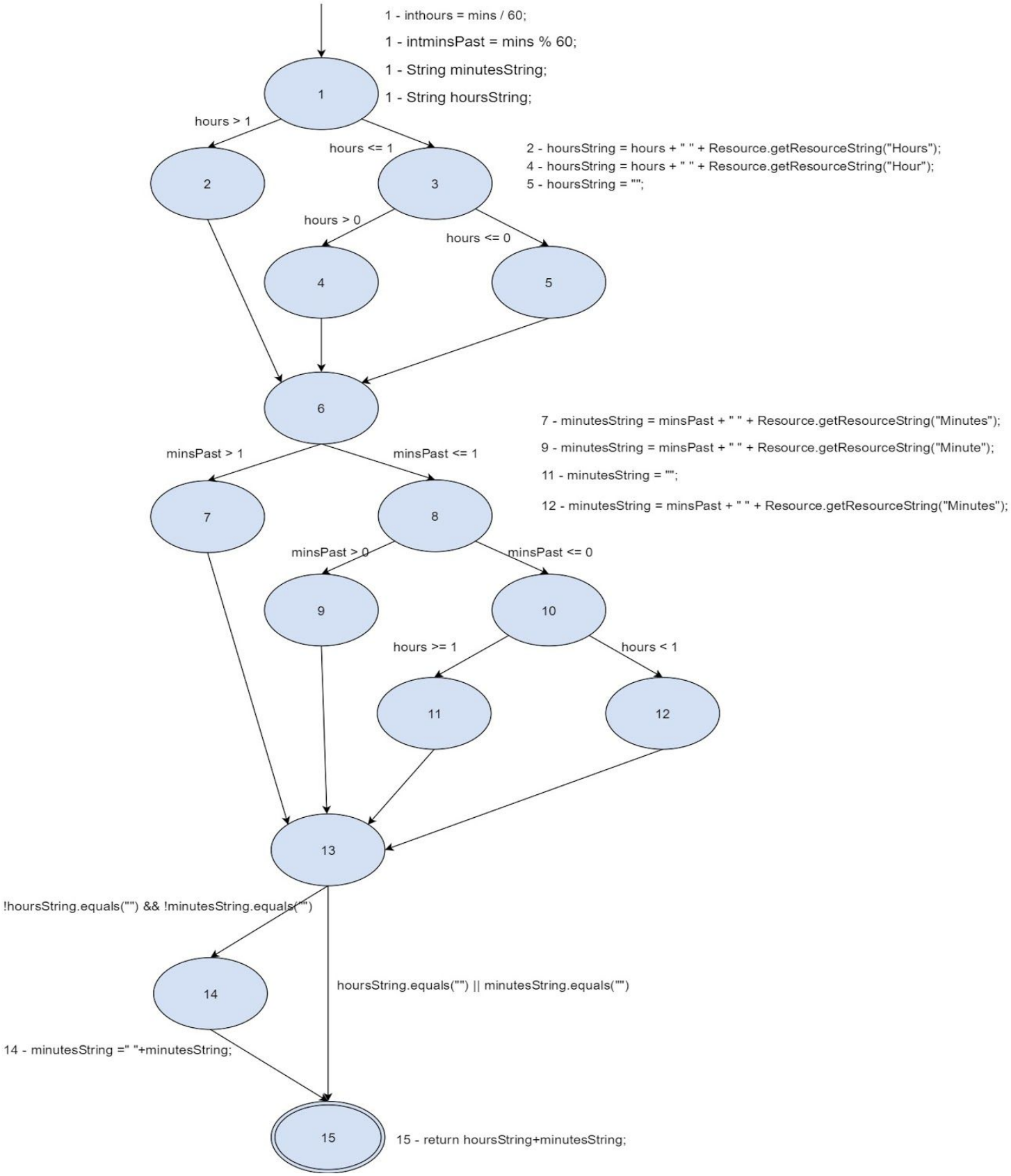

Figure 4: Control Flow Graph for minuteString method

The branch from node 3 to node 4 is not being taken as well as the branch from node 10 to node 11. The final branches are also not being tested for its “F” path. Therefore, it is not the statements that are not being covered but that the branches aren’t being taken to test. We can try to create separate test cases for each of these branches to cover, but there is one test case that allows for us to take all three missed branches. The test case is:

“assertEquals("1 Hour", DateUtil.minuteString(60));”,

to assert that giving the parameter 60 to the method will return the string “1 hour”. The first untaken branch (node 3 to node 4 in CFG below) will be taken then the second untaken branch (node 10 to node 11 in CFG below) will be taken and finally the last branch will be “F” and it will return the minuteString covering all branches and statements. The addition brings the instruction, statement and branch coverages up to 100% as the tests go over all branches and statements.

2.3 model > Repeat > getDayList

The getDaylist method is easier to get to 100% coverage than the minuteString method. During our Black-Box testing we covered many cases due to the complex nature of the branches this method has. This was to cover all the different ways the list of days can be built, but there is one case that checks for formatting of the input string that we did not consider and this is the cause of our 95.7% statement coverage.

```
281         int i2 = f.indexOf(',', DAYLIST.length() + 1);
282         String list = null;
283         if (i2 != -1)
284             list = f.substring(DAYLIST.length() + 1, i2);
```

Above we see that the statement on line 281, this statement looks for a comma in a particular spot in the input string, if the index of the comma does not exist in the particular spot then the index is in the correct format and can be searched to insert the integer related to days into a list for a particular appointment. In our tests we never considered if the input string is in the wrong format and hence the red statement on line 284, our tests will always consider when this index check is -1 and will never take the true branch of the decision on line 283. The way to remedy this is to expect an empty list when inputting an incorrectly formatted day list string like so “assertEquals(sq, Repeat.getDaylist("dlist ,1"));”. Notice how the input string is “dlist ,1” the comma is one index farther than it should be, the correct format of the input string would be “dlist, 1”. The reason this check is done in the method is so the remainder of the branch decisions done will correctly identify day-number representations, thus if the format is incorrect the method just returns an empty day list. This brings our statement coverage up to 100% as well as interaction and branch coverages.

2.4 Final Statement Coverage Measurements

The following is the new White-Box statement coverage measurements:

Method	Coverage (%)	Covered Statements	Missed Statements	Total Statements
isAfter(Date, Date)	100.0	13	0	13
minuteString(int)	100.0	19	0	19
getDaylist(String)	100.0	23	0	23