ETSN20 — Lab report

Jacob Bentzer and Henrik Vester

November 28, 2024

Contents

1 Introduction

The purpose of this lab report is to compare the efficacy of using white box versus using black box testing and in which scenarios one is to prefer over the other.

1.1 Procedure for White Box Testing

In order to explore the process of white-box testing, we consider an example program `NextDate`, that provides logic for incrementing a date while considering leap years and that different months have a different number of days. We are only interested in testing the `String run(int month, int day, int year)` method.

First, we create a control flow graph of this method where we visualize all execution paths of the method; see figure 1 in appendix C. We find that the number of decision points is $n = 16$. Using this flow graph, we can also calculate McCabe's cyclomatic complexity score, which is $n + 1 = 17$. This is a metric of the number of independent paths through the method.

We have three different coverage goals: (i) statement coverage, (ii) branch coverage, and (iii) predicate coverage.. Having a statement coverage of 100 % means that all statements in the method have been executed at least once during testing, while 100 % branch coverage means that all branches need to have been executed at least once.

Looking at the control flow graph, we can see that there are 17 independent execution paths in the `run` method. There are statements in each branch, which means that we need 17 tests (one for each execution path) to achieve full statement coverage. It follows that we do not need any more tests to achieve full branch coverage.

Full predicate coverage means that we need to test all combinations of valid sub-expressions in each decision point. In the `run` method, the only decision point which needs to be tested further is the first one:

```
if (day < 1 || month < 1 || month > 12 || year < 1801 || year > 2021)
```

We have five subexpressions, but they are not independent. For example, the `month` variable cannot simultaneously have a value less than one and bigger than twelve. The `month` variable can either have a value less than one, between one and twelve inclusive, or greater than twelve for a total of three cases. The same is true for the `year` variable. The `day` variable can be either have a value less than one, or greater than one inclusive for a total of two cases. Therefore, the number of test cases needed for full predicate coverage turns out to be $2 \times 3 \times 3 = 18$. One of these cases is covered through the regular statement and branch coverage tests, meaning we need an additional 17 tests for full predicate coverage. We still need the 17 branch coverage tests to achieve full predicate coverage for all other decision points, meaning we need $17 + 17 = 34$ test cases for full predicate coverage.

1.2 Procedure for Black Box Testing

In the lab for black box testing we tested the output from the component `Triangle` using both Equivalence Partitioning (EP) and Boundary-Value Analysis (BVA).

For equivalence partitioning the first step was to create subdomains, or classes, for which all inputs within the class would be treated the same in the component. The classes were as follows:

- C1: Input: Impossible, where one of the conditions are not fulfilled:
  - $s_1 + s_2 > s_3$
  - $s_1 + s_3 > s_2$
  - $s_2 + s_3 > s_1$
- C2: Input: Scalene, where the following conditions are fulfilled for all variations of $x, y, z = 1, 2, 3$:
  - $s_x \neq s_y \neq s_z$
  - $s_x^2 + s_y^2 \neq s_z^2$

- C3: Input: Equilateral, where the following condition is fulfilled:
  - $s_1 = s_2 = s_3$
- C4: Input: Right-Angled, where the following conditions are fulfilled for all variations of $x, y, z = 1, 2, 3$:
  - $s_x \neq s_y \neq s_z$
  - $s_x^2 + s_y^2 = s_z^2$
- C5: Input: Isosceles, where the following conditions are fulfilled for all variations of $x, y, z = 1, 2, 3$:
  - $s_x = s_y \neq s_z$
  - $s_x^2 + s_y^2 \neq s_z^2$
- C6: Input Isosceles and Right-Angled, where the following conditions are fulfilled for all variations of $x, y, z = 1, 2, 3$:
  - $s_x = s_y \neq s_z$
  - $s_x^2 + s_y^2 = s_z^2$

Some of these classes are expected to behave the same way in some of the methods in the `Triangle` component, wherefore we only need to test one of the classes with the same behavior for that specific method. We proceeded to write test cases for each of the classes and methods where the output had not yet been produced. Of note is that class C6 is not possible in the program due to the fact that all the inputs are in the form of integers and for the conditions to be fulfilled at least one of the inputs would have to be a double. For all test cases see the attached file `TriangleTest.java`, tests 1–22.

For boundary-value analysis we used the same classes defined for equivalence partitioning, but in creating test cases we selected values to test the boundary between two classes. What constitutes a boundary is not as clear when the classes has more complex conditions, compared to in the condition was just a range of numbers. We decided to define a boundary such as where one class could cross into the another class, without first crossing a third class. For example, C5 (Isosceles) can cross into C2 (Scalene) since changing a single one of the input integers by just incrementing or decrementing by one would make it scalene. On the other hand C3 (Equilateral) must first cross through C5 (Isosceles) before it can reach C2 (Scalene). We therefore see a boundary between Isosceles and Scalene but not between Equilateral and Scalene. Using this methodology we found the following boundaries:

- C2 (Scalene) — C5 (Isosceles)
- C2 (Scalene) — C4 (Right-Angled)
- C2 (Scalene) — C1 (Impossible)
- C3 (Equilateral) — C5 (Isosceles)
- C2 (Equilateral) — C1 (Impossible)
- C5 (Isosceles) — C1 (Impossible)
- C5 (Isosceles) — C4 (Right-Angled)

For all the boundaries we tested the relevant methods for the current border. For a full list of the BVA tests see the attached file `TriangleTest.java`, tests 24–71.

2 Results

We summarize the results of the white-box testing with the three different forms of test coverage goals, and the black-box testing techniques of equivalence partitioning and boundary-value analysis in table 1.

Table 1. A summary of the white-box and black-box testing techniques.

	White box			Black box	
Metric	**Statement**	**Branch**	**Predicate**	**EP**	**BVA**
Test cases	17	17	34	22	48
Coverage (%)	90.5	90.5	90.5	—	—
Defects found	2	2	2	7	18

For white-box testing, we found a total of two defects (see appendix A for a full defect report). Notably, we were fine with a goal of just 100 % statement coverage. Adding an additional 17 tests to achieve full predicate coverage did not uncover any additional defects, nor did it increase the coverage as reported in Eclipse of 90.5 %.

The reason this number is not 100 % is due to a logic error in the following line:

```
if (day <= 31) //if the day is not 31, just increment the next day
```

The intent here has been to treat days in December differently depending on if it is the last day (31) or not. However, because of the greater-than-or-equal sign, we will never proceed to the else-branch. We will also certainly never achieve full coverage, regardless of metric, since there are no valid inputs which result in this execution path.

The equivalence partioning testing results in 22 test cases. These 22 test cases found a total of 7 defects. The boundary-value analysis found 18 more defects, for a total of 25. See appendix B for the full defect report.

Testing through boundary-value analysis is significantly more time-consuming than simple equivalence partioning, requiring almost double the test cases. It might seem at a first glance that it finds more errors, though. However, all 25 defects as described in appendix B cannot be considered independent defects in their own right — most of them are related. We can categorize the found defects into six underlying logical errors in the program:

1. `getArea` for impossible triangles always returns 0, while the right answer is −1.0. Caught in tests 1, 43, 57, and 65.
2. `getArea` does not return the right area, even for possible triangles. Caught in tests 11, 42, 56, and 64.
3. `classify` returns “scalene” for some triangles that should be isosceles. Caught in tests 25 and 58.
4. `classify` sometimes returns “isosscseles” for isosceles triangles, while the specification states that it should return “isosceles” (note the spelling error). Caught in tests 18 and 45.
5. `getPerimeter` returns the wrong perimeter for impossible triangles. Caught in test 20.
6. `classify` returns “scalene” for some triangles that should be impossible and `isImpossible` should return true for these.

Of these, only categories three and six required the use of boundary-value analysis to be found. It can be argued, however, that these are serious defects and that the use of boundary-value analysis was justified in this situation in spite of the added complexity and time.

3 Discussions and Conclusions

We will summarize what we have learned from the white-box and black-box testing, as well as some differences between them and in what situations you might use which.

3.1 White-box Techniques

In the case of white-box testing, we find that the various coverage criteria do not differ that much from each other. Full branch and statement coverage were achieved through the same tests, while full predicate coverage required some more. We found it hard to justify the additional time spent on the predicate coverage tests, because they did not manage to find any additional defects in the program. Predicate testing also leads to a combinatorial explosion in that the number of tests required for full coverage explodes as the number of sub-expressions in a decision point increases.

Furthermore, we struggle to find an example in which full branch coverage is not implied by full statement coverage. This would mean that there would be branches in the program which do not contain any statements, which we do not see why it would be useful.

An interesting phenomenon that we uncovered was that the mere presence of less-than 100 % statement coverage indicated problems—in this case, that there were unfeasible paths in the program as a result of a logic error.

3.2 Black-box Techniques

As noted in the results section of this report, using boundary-value analysis is more time consuming but is a more exhaustive method of testing. In the related lab there were two whole types of errors which were not uncovered by simple equivalent partitioning. Which to use seems to us a simple matter of how important it is that the code works. In cases where there are a lot of subdomains which share a boundary, boundary-value analysis might not be feasible to perform.

It could be appropriate to use t-way testing, or some other type of combinatorial testing, to test the `Triangle` component, since is has a relatively few number of inputs (three). However, each of these inputs has a vast range and very specific combinations of them should be handled differently by the program than others very similar combinations. We do not see a feasible way to use t-way testing under these conditions. We expect t-way testing to be most efficient when there are a relatively small number of combinations which can only assume a few different values each.

Another testing method that is always worth trying is error guessing. Trying some exploratory tests might catch errors that the systematic approach misses, especially if done in areas you think might be probelamatic.

3.3 White-box testing vs. Black-box

From these results, it is hard to directly compare white-box and black-box testing since we tested two different programs (`NextDate` and `Triangle`, respectively). However, we can present some overall conclusions.

First, we find that white-box testing generally results in fewer test cases than it's closest equivalent of boundary-value analysis. This comes as no surprise, as we are helped by the program structure when designing the tests.

We argue, however, that black-box testing is better when we are concerned about the adherence to some kind of program specification. One notable defect that we did not uncover in the `NextDate` program was that the month of July was not handled correctly at all, and was not tested for anywhere in the code. Since we are only concerned with the actual program structure when white-box testing, we can easily be fooled if this structure contains logical errors or omissions like this one. Had we been black-box testing `NextDate`, it is much more likely that we would have found this severe defect, since July would have been one of the equivalence partitions.

A Defect Report for `NextDate`

Test	Method	Description	Expected	Actual	Remark
9	`run`	Returns 12/32/2001 for an input of 12/31/2001, while it should be 1/1/2002.	1/1/2002	12/32/2001	Severe Defect
11	`run`	Returns 12/32/2021 for an input of 12/31/2021, while it should be "Invalid Next Year"	"Invalid Next Year"	12/32/2021	Severe Defect

B Defect Report for `Triangle`

Test	Method	Description	Expected	Actual	Remark
1	`getArea`	For an impossible triangle, the area should be -1.0, where actual output is 0.	-1.0	0	Severe Defect
2	`classify`	Impossible triangle should return "impossible" but returns "scalene"	"impossible"	"scalene"	Severe Defect
3	`isImpossible`	For an impossible triangle `isImpossible` should return true but returns false.	true	false	Severe Defect
7	`isScalene`	Should return false for an impossible triangle, but returns true	false	true	Severe Defect
11	`getArea`	For a scalene triangle (3, 4, 6), should return 5.33 but returns 0	5.33	0	Severe Defect
18	`classify`	For an isosceles triangle, `classify` should return "isosceles" but returns "isossceles"	"isosceles"	"isossceles"	Minor Defect

Test	Method	Description	Expected	Actual	Remark
20	`getPerimeter`	Should return -1 for an impossible triangle (-1, -1, -1), but returns -3	-1	-3	Severe Defect
22	`classify`	Writing the characters 'a','b','c' creates a triangle.	Don't know, just a weird interaction but might be hard to address.	"scalene"	Quirk of the system
25	`classify`	Should return "isosceles" for a triangle (2, 3, 3) but returns "scalene"	"isosceles"	"scalene"	Severe Defect
37	`classify`	For an impossible triangle (2, 3, 5), `classify` should return "impossible", but returns "scalene"	"impossible"	"scalene"	Severe Defect
39	`isScalene`	Should return false an impossible triangle (2, 3, 5), but returns true	false	true	Severe Defect
41	`isImpossible`	Should return true for triangle (2, 3, 5) but returns true	true	false	Severe Defect
42	`getArea`	Should return 2.905 for a triangle (2, 3, 4), but returns 0	2.905	0	Severe Defect
43	`getArea`	Should return -1 for triangle (2,3,5) but returns 0	-1	0	Severe Defect
45	`classify`	Triangle (2, 2, 1) should return "isosceles", but returns "isossceles"	"isosceles"	"isossceles"	Same error as in test 18
51	`classify`	Impossible triangle (1,1,2) should return "impossible" but returns "isosceles"	"impossible"	"isosceles"	Severe Defect

Test	Method	Description	Expected	Actual	Remark
55	`isImpossible`	Triangle (1, 1, 2) should return true, but returns false	true	false	Severe Defect
56	`getArea`	Triangle (1, 1, 1) should return 0.433, but returns 0	0.433	0	Severe Defect
57	`getArea`	Triangle (1,1,2) should return area -1.0 but returns 0.	-1.0	0	Severe Defect
58	`classify`	Should return “isosceles” for triangle (1, 2, 2), but returns “scalene”	“isosceles”	“scalene”	Severe Defect
59	`classify`	Triangle (1,2,3) should be impossible but is scalene	“impossible”	“scalene”	Severe Defect
63	`isImpossible`	Should return true for Triangle(1, 2, 3), but returns false	true	false	Severe Defect
64	`getArea`	Triangle (1,2,2) should have area 0.9862 but returns 0.	0.9862	0	Severe Defect
65	`getArea`	Should return -1 for impossible triangle (1, 2, 3), but returns 0	-1.0	0	Severe Defect
66	`classify`	Triangle (3,4,4) should be isosceles but returns scalene.	“isoceles”	“scalene”	Severe Defect

C Control Flow Graph for `NextDate`

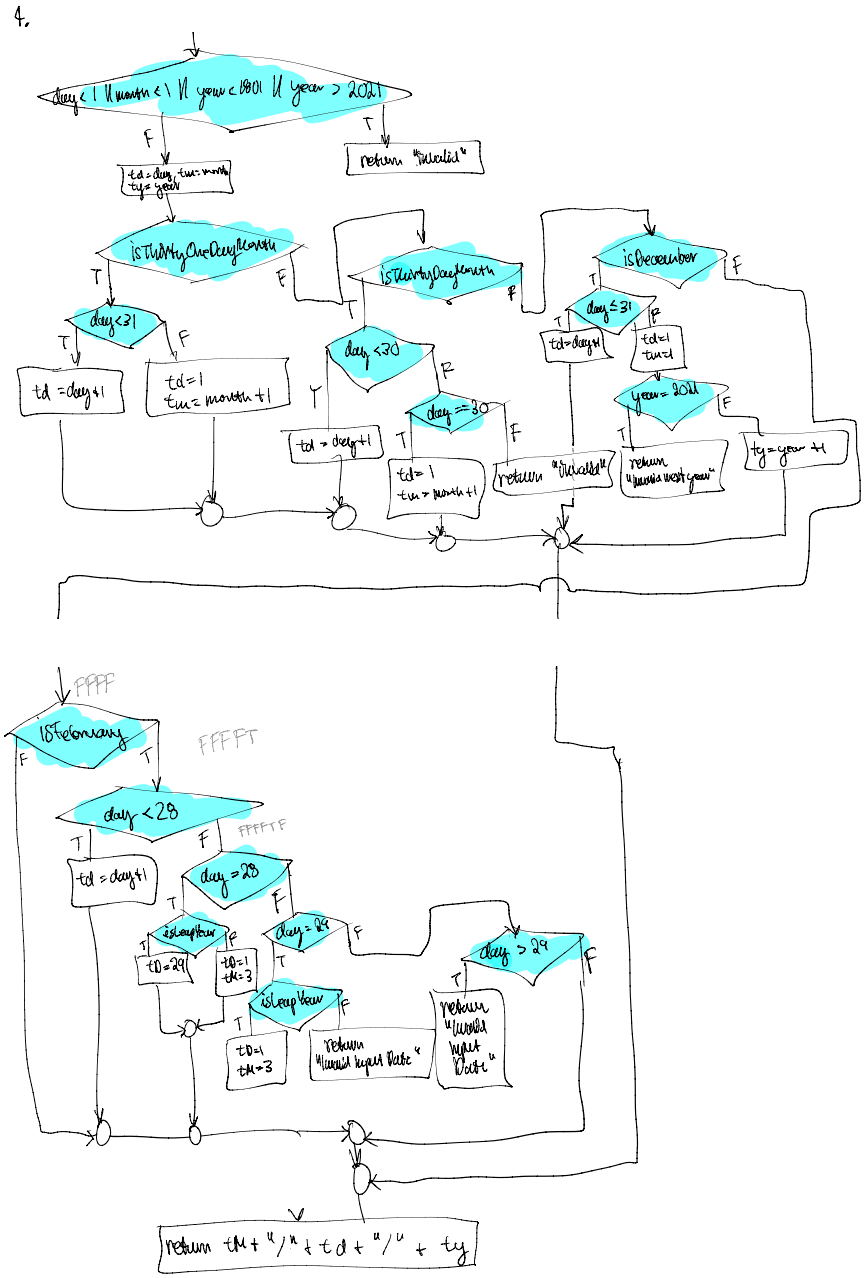

Figure 1. The control flow graph of the `NextDate` program as used during the white-box testing. All decision points are shaded blue.