Part 1.

1. Quality trends analysis with McCabe IQ tool
2. Measurements and interpretation
   1. Average Cyclomatic Complexity [v(G)] and Essential Complexity [ev(G)]:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Summary | M1 | M2 | M3 | M4 | M5 |
| v(G) | 17.8 | 36.6 | 38.6 | 38.8 | 40 |
| ev(G) | 12 | 24 | 25.6 | 26.2 | 27.2 |

* 1. Charts and interpretation

From analyzing the charts

Part 2.

1. McCabe Analysis:

As seen in the charts above, the analysis of the average cyclomatic and essential complexities shows that there is a trend in the growth of these complexities. We can see that the complexity values become bigger at every subsequent milestone and that it tends to grow a lot faster during the first couple of milestones compare to the rest. This

II. Logiscope

1.

ANALYZABILITY = cl\_wmc + cl\_comf + in\_bases + cu\_cdused

* cl\_wmc = sum of static complexities of the class methods = SUM (ct\_vg)
* cl\_comf = ratio between number of lines of comments in the module and the total number of lines ( cl\_comm / cl\_line).
* in\_bases = number of classes from which the class inherits directly or not if multiple inheritance is not used the value of in\_bases is equal to the value to in\_depth (in\_depth = maximum length of a chain of inheritance starting from the current class)
* cd\_cdused = number of classes used directly by the current class

CHANGEABILITY = cl\_stat + cl\_func + cl\_data

* cl\_stat = Number of executable statements in all methods and initialization code of a class.
* cl\_func = Total number of methods declared inside the class declaration.
* cl\_data = Total number of attributes declared inside the class declaration.

STABILITY = cl\_data\_publ + cu\_cdusers + in\_noc + cl\_func\_publ

* cl\_data\_publ = Number of attributes declared in the public section or in the public interface of a Java class.
* cu\_cdusers = Number of classes which use directly the current class.
* in\_noc = Number of classes which directly inherit from the current class.
* cl\_func\_publ = Number of methods declared in the public section.

TESTABILITY = cl\_wmc + cl\_func + cu\_cdused

* cl\_wmc = sum of static complexities of the class methods ( SUM (ct\_vg)).
* cl\_func = Total number of methods declared inside the class declaration.
* cu\_cdused = Number of classes used directly by the current class.

2.

i.

**Milestone 1:**

EXCELLENT: 80.0% GOOD: 20.0% FAIR: 0.0% POOR: 0.0%

**Milestone 2:**

EXCELLENT: 71.43% GOOD: 28.57% FAIR: 0.0% POOR: 0.0%

**Milestone 3:**

EXCELLENT: 71.43% GOOD: 28.57% FAIR: 0.0% POOR: 0.0%

**Milestone 4:**

EXCELLENT: 71.43% GOOD: 28.57% FAIR: 0.0% POOR: 0.0%

**Milestone 5:**

EXCELLENT: 28.57% GOOD: 57.14% FAIR: 14.29% POOR: 0.0%

As it is evidently clear, the milestone with the worst quality code (based off of its maintainability), by deduction, is milestone 5 since it has the lowest percentage of maintainability with 28.57% of the code being considered excellent, 57.14% being considered good, and it is also the only milestone with code considered fair with 14.29%, thus strengthening the affirmation that it is in fact the least maintainable milestone by Logiscope’s standards of calculation.

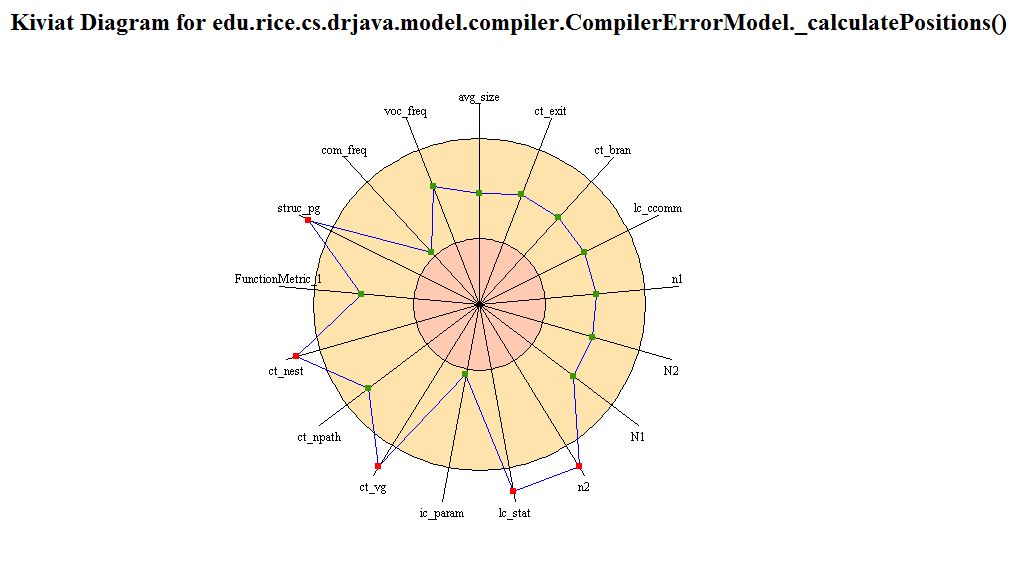
ii.

class : CompilerErrorModel.java (lowest ranked class of all the milestones. Milestone 5; ranked FAIR)

After analyzing this particular class, it can be noted, that simply, it is has to read. This relates to bracket proximity, if statement separation, nested statements (nested loops, nested ifs, nested try-catch blocks, etc.), and location of variable declaration (both instance and local), as well as their initialization locations. It is an utter mess. Even the comments are all mashed in between these whitespace-less lines of code, making them almost indiscernable from the actual code. To further comment on the variables, having them so scattered about makes it not only nearly impossible to keep track when reading the code (since you’re busy trying to absorb the fact that already the method itself is massive and easy to get lost in such a jungle of supposed compilable text) but it also gives the impression that the author of such a mess literally planned nothing before hand in terms organization. It seems that whenever the author may have needed a new variable, it was declared on the spot (obviously nothing syntactically wrong about this detail, but it is quite impractical on the human eyes).

iii.

class : CompilerErrorModel.java => method : \_calculatePositions()

Outstanding parameters of the Kiviat Diagram and their meanings:

- ct\_nest; Maximum number of control structure nested levels in a function.

- ct\_vg; Cyclomatic number of the control graph of the function. In the graph theory the Cyclomatic number, V(g), gives the number of linearly independent paths in a connected graph.

- lc\_stat; Number of executable statements in the function.

- n2; Number of different operands used in the function.

- struc\_pg; Level of structuredness in the program (or lack thereof. i.e. Essential cyclomatic complexity)

So, to more specifically re-iterate (now that there is a visual depiction of the statistics commented on above in part ii.), the Logiscope software noted that, for the given, singular method, there was an outstanding number nested levels within the method (be it if statements, loops, etc.). Furthermore, the number of executable statements in the function was deemed to be too high, for good reason at that. It implies a “god-method”, all doing, all functioning, and non modular. It implies a lack of cohesion and conciseness. Also, as stated earlier, Logiscope caught on to the over abundance of operands (variables) within the method, thus confirming weak traceability, readability, and overall maintainability in that regard. Lastly, both the cyclomatic and essential complexity were notably high, which should really be the first red flag to begin with. With a notable lack of structuredness to the method and complexity, there becomes way too many test cases, and it becomes extremely difficult to track calls from other methods and nested calls, since lack of structure is rampant.

3.

i.

For milestone 1, there is not much to change since it received 80% EXCELLENT and 20% GOOD. As far as code quality standpoint, perhaps the exceptions could be handled a bit better, but it’s hard to say since I don’t know the exact reason why they weren’t and I’m sure it's a good reason, but other than that, there is not much else to say.

For milestones 2, 3, and 4, they had all received the same score (71% EXCELLENT and 29% GOOD), again, there is not too much to say except that again, declaration of multiple local variables and perhaps dealing with exceptions within their respective catch block may very well add to the readability, traceability, and overall maintainability to the code in question.

ii.

To help with variable declarations, perhaps always keep all declarations at the top of every method and have one comment describing the “category” or the relative significance of said group of variables to the code below it. This may be similar to structured requirements/standards documents, where potentially confusing terminology is defined first, before reference to said terminology is made. Also, splitting the methods into more, small, modular, cohesive methods would do good for traceability and reuseability of the class/method since now a developer can pinpoint a problem to a given method as opposed to looking through strands of nested while loops and try blocks that look like an italian dinner table (i.e. Pun on spaghetti code. It is appropriate to laugh now)

iii.

Again, main problem lies above, since there is only class that ranked as FAIR and none as POOR. And it was due to only two methods. These methods had identical issues. This being the massive amounts of operands and cyclomatic complexity and unstructuredness due to too many scattered variables and methods calls in nested if statements and loops,etc. I refuse to say this again. One class, two methods, same issues. Reference Kiviat chart mentioned previously.

Part 3.

Although the instructional slides on how to use LogiScope were a bit incomplete, it was manageable. And once the project was built, everything else was quite intuitive thanks to a pretty nice UI. Even though there is a lot of terminology, its made relatively simple by just using the metrics dictionary and making the connections. This is emphasized through the quality report pie charts and detailed Kiviat charts.

The McCabe tool was also quite simple to use and provided all the metrics needed in the detailed, generated report. Despite the UI being preferred on Logiscope, McCabe tool wins with the inclusion of the class relationship diagrams and slightly more accurate readings for the complexities.