1 Importance of Metrics in X3Profiler

As programs continue to increase in complexity and size, maintainability issues will become a major problem for most software projects. A software development team that closely follows tried and true engineering practices will probably succeed in delivering a correctly functioning product. As any good software engineer is aware, after delivering the product, the job is not done yet. As long as users continue to run the software, maintenance and revisions will be a necessary part of keeping that software alive and well. If we do not pay close attention to how our software product is developing, and act accordingly, then this product will quickly become a nightmare to maintain.

We, as software developers, are not helpless against this problem though. Much research has been done to develop tools and procedures for keeping software under control at all times. Pre-established metrics are a valuable tool when dealing with source code complexity. Unfortunately, though, metrics are often not used properly throughout the development life cycle. Developers either never take the measurements, or they take the measurements but dont do anything with them. In either case, everyone loses when the program becomes an untamable beast with rabies.

When we take a source code measurement, we are taking a snapshot of the programs internal state. We are viewing our work from an objective point of view, which is something most programmers cannot do with their own work. While it is true that metrics are not absolute indicators of trouble, when applied intelligently, they do sort out potential problematic areas in the source code. Programmers should spend some extra time analyzing these results and applying any changes if they believe those changes are needed.

There are two big decisions to make with regards to metrics. The first big decision is, of course, which metrics to apply. There are many pre-established metrics out there, most of them with out of the box programs that will apply them to the source code. Programmers, however, should be careful not to fall into the trap of believing that simpler metrics are less useful than more complex ones. The second big decision is when to apply such metrics. Normally, it is best to apply such metrics on a regular basis, even when the program has not undergone much change. In X3Profiler we have chosen to apply several metrics to our program, most of them simple and easy to understand, on a regular basis.

2 Metrics Used in X3Profiler

The first metric we apply when measuring our software is called Number of Lines. This metric might seem to simple to be useful, yet it quickly generates a rough estimate of how big our project is. This metric is generated by parsing our source code files and counting how many lines of text we have in them. With this reading we can derive other statistics of our source code such as how fast are we progressing. There is one big problem with this, though. As Bill Gates once said, Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weights. In a way, what Bill Gates meant was that Number of Lines does not take into account other aspects like good programming practices into the picture. Thus, while Number of Lines gives an insightful look at how much source code we have written, it does not really gauge how much progress has actually been accomplished, it is simply a very rough

estimate.

The second metric we apply is called The McCabe Complexity Metric (MCC). This metric is designed to measure the number of linearly independent paths through a program. Put simply, this means how many branches can a given program go through. For example, if a program has three if statements, then there are eight possible paths that this program could possibly go through. Extrapolating a bit, this metric measures how complicated the logic is in a given program since the more branches a program has, the more states it can reach. Therefore, a high reading in this metric indicates that the program is hard to comprehend and maintain.

The main reason we use the MCC metric in our project is to determine how complex our testing should be. Ideally, if we have eight possible paths of execution, we should definitely test each path thoroughly to ensure reliability. We all understand that testing is an unfortunate impossibility, and here is where MCC can help out significantly. We can concentrate our testing efforts on those modules that receive a high MCC reading since those are modules likely to have errors in them. We can refactor such modules so they are simpler to comprehend and less likely to have errors in them. Testing becomes so much easier by listening to the MCC.

The third metric we apply is called the Halstead Complexity Measurement (HCM). Similar to the MCC, this metric attempts to determine which modules are complex and error prone, thus inherently evil. As usual, this metric is created by parsing the source code files, yet it differs from the other metrics in that the operators and operands now determine the complexity of the program. The reason for which HCM measures complexity this way is that data can only be changed by operators and operands, and errors are likely to occur where data changes frequently. Thus, just as with the MCC, this metric helps out when simplifying code for testing and maintenance purposes.

The fourth metric we apply is called the Module and Function Size metric (MFS). This metric is a watered down combination of Number of Lines and MCC. It generates its value by counting how long a function is by itself. The importance being that most functions that are long can be split into several sub functions that will allow for greater maintainability and reusability since they are easier to understand. Why have three different metrics for one common goal? Based on the fact that none of these metrics are absolute, we want to look for modules to refactor from as many angles as possible.

The fifth metric we apply is called the Number of Classes (NOC). All it really does is keep track of how many classes we have in our program. In a way, this metric reading sort of hints at how decomposed our program is since the more classes we have, the more chances of reuse we get. The whole point of this is to make sure no one class is trying to do too much by itself. This not only aids in reuse, but it is a great aid in improving maintenance and testing as having each class by itself localizes potential problems to that class alone.

Finally, the sixth metric we apply is called Percent of Comments (POC). This metric basically computes the ratio of lines of comments to lines of code. POC is extremely useful when generating documentation for our software product; if we notice a low POC number (probably somewhere around the low ten percents) that means we need to document more what we are doing for future use. On the other hand, if we notice a high POC number (probably somewhere around the high thirty percents) that means we need to cut down on our documentation. In the end, this metric warns us of having either few comments or far too many comments in our code, both of these

being bad for the project.

As we can see, metrics are very helpful in pinpointing abnormalities in the source code. They hint at the programmers when some module is too long, too complex or simply not well documented. Programmers need these kinds of tools because while they might think the module they wrote is not hard to understand at all, it might not be the same for the new programmer that will get stuck maintaining it in years to come. As with anything else, metrics are guidelines meant to suggest problem areas in the source code. As long as the programmers take the measurements and analyze their results, software maintainability and reuse can be improved significantly.

3 Analysis of Measurements

Note: We have analyzed the metric readings for the DataTypes.py and x3gui.py files since these are the ones that we mostly worked on. The other files were either static (like x3p.py which calls everything else) or were automatically generated by QT (like x3gui.ui.h)

Under the "Previous Iteration" folder (and under the Construction phase) you will find 3 folders with metric data taken during this semester. Now, the very first metrics data was generated for the Metrics Homework as part of the class, as such, we have not included it here (we are interested in how our project developed since then.)

The following are dates at which the data was generated and any outstanding reasons for their measurements, if any:

02/17/2005 Homework 2, we needed to start generating data.

03/01/2005 2 weeks since first set was generated.

03/30/2005 4 weeks since second set was generated. Project fully under way at this point, though metrics were not taken biweekly because we were having problems with our graphing library.

04/15/2005 2 weeks since third set was generated, file writing was being attempted with PYX.

04/28/2005 CS Days approaching, finalized code. Final Results on our metrics.

Overall, as the semester went by, these are the most general trends that we found in our program:

1. The most shocking numbers came back from the McCabe Complexity Metric. This was most notable in our x3Gui file. Throughout this file, the lowest number we got for some functions was a 2, which is quite acceptable, yet we got several values of 4 and 6 for our more important functions (like graphOperations.) In a way, these values did not change much throughout our development since most of these functions were not long, but since they had a lot interaction with QT libraries (this is the GUI file after all) we got high level readings. In order to compensate for this high MCC value, we spent a great deal of time manually testing the GUI and the output it produced.

2. One metric that does impact us (at least the last reading) was the % of comments we had in our main file DataTypes.py. Now, throughout most of our development we kept the % of comments around a 10% to 15% range, which is quite good. Our biggest spike in commenting came when we attempted to do the graphing through file reading. Since we took some shortcuts around that time we commented those in the code itself. Now, near the end of the semester we found an easier way to accomplish the graphing without resorting to file writing (which caused cache errors with the graph not being updated.) We ended up using more built in Python functions (which we felt really didn't require much commenting) and thus both our lines of code and % of comments dropped a bit. Is this something severely bad? No. We have generated a ton of other documentation already to supplement what we do with a why we do it.

3. For the Halstead Metric, again, x3gui.py came out upfront being a clear winner. It is interesting, though, to notice how this file ends up with another high reading. If we look at how x3gui.py is generated, we will see the reason for it. x3gui.py is tied up to x3gui.ui.h (which is *automatically* generated by QT designer) and it involves *many many* QT function calls to accomplish even the simplest thing. Again, just as with case 1 above, we are not too concerned about this metric reading in the sense that while it is high, this file is jointly created with the help of the TrollTech QT Designer Interface, so the programmer is not really alone when dealing with these files.

4. The number of classes increased only at major revisions of our code. We decided to create a graph class for each possible graph format (this proved to be far easier and modular for PYXusage.) Other than that, this reading stayed fairly stable since once we didn't change our architecture around after it was finished last semester.

5. The number of lines grew steadily as we moved ahead through the semester. This is, of course, perfectly normal as more features get integrated as we near the handing in deadline. Now, as you will see in these metrics reading, some files did not change drastically. This, however, does not mean that heavy work underwent through each one. If you take a look at the CVS.log file as well, many submissions included some refactoring of our main files.

Overall, the Source Code files can still be improved upon (this is usually true always, though.) Now, based on both the metrics, on our analysis of the metric readings and our own experience with developing the code for this project, we can safely say that a new programmer entering the development of X3Profiler will be up and running (maybe fixing bugs or adding features) in *tops* one week. While some readings are high, our code is not very large itself (in numbers of lines) and we do have good documentation for it.