Code Quality Analysis Project

December 6, 2016

Contributors

Wesley Ellington 45487212

Seung Ki Lee 35460312

Overview

For this project, we were tasked with creating a tool for the static analysis of code quality. To do so, we first had a to answer the questions of what good code was, and how we could identify it in a practical application. After a great amount of research and discussion, we have designed a unique code analyzer that employs five different metrics, some of which are industry standards, some of our own design. It was very important to us that our metrics be meaningful to the user, therefore, they are all processed separately and considered different in nature from one to the next.

There are many code analysis tools in existence today, so creating some basic metrics was of no issue to us as we began. However, we wanted to take this opportunity to look into some unexplored territory to see if there are areas of analysis that could be leveraged by a user of our program. We know that these two metrics stray from the norm of what is used by most companies today, but hope to learn a great deal from them.

In this document, we will look at our program design, its implementation, our metrics and the considerations we made in their creation, as well as the lessons we learned along the way. Additionally, we will look at some examples of our analyzer's output in relation to some code bases to gain some insight into the meaning behind the numbers we report. Hopefully, this will provide an inside look at our tool and what went into making it a reality.

Implementation and Design

When designing any large coding project, there are a great deal of factors that must be assessed before the first line of code is written. If well executed, the product of a process beginning will be far more valuable to the user, as well as have opportunity for growth in the future. By implementing an Object-Oriented design, we were able to save a great deal of hassle with testing and maintainability.

File Discovery and Directory Searching

To begin, we first needed a way to discover what files we needed to analyze for a project. To do this, we created a class called a directoryIterator, that when created with a specified file path, will generate a list of all .cpp, .hpp, .c, and .h files within all subdirectories. This made it easy for us to later on create parsed output from the files to analyze for our metric. Additionally, this class would open and close each file that it saw as valid, ensuring that all of the files it located were readable and would not cause issues for processes along the road.

Naturally, we could not accomplish this task without the help of the C++ standard library, using the dirent.h header to analyze the desired directory and subdirectories as we descended through them. Upon discovery, each filename would be added as an absolute path to a vector of strings stored within the directoryIterator class. This way, an instance of the class can be used by other functions to see all of the files in question.

Metric Base Class

One of our main concerns in this task was the expandability and portability of the code we created. With this in mind, decided to leverage the polymorphic capacities of the C++ language to our advantage through inheritance. We created an abstract base class served as a placeholder and drew from it to create multiple subclasses, one for each metric we chose. The base class shall contains three main attributes, a double that will represent the score of the code on that given metric, an int that represents a weighting value for use in creating a compound score, and a vector of well formatted strings to give verbose output should it be requested. All of these make it possible for simple and concise calculation of score, as well as well formated and consistent test output. Each metric has unique member functions for creating its results as well as access to the list of files discovered by the directoryIterator.

By keeping this functionality in mind, this code system is easily expandable and adding new tests require a one to two line edit to the main body of the code.

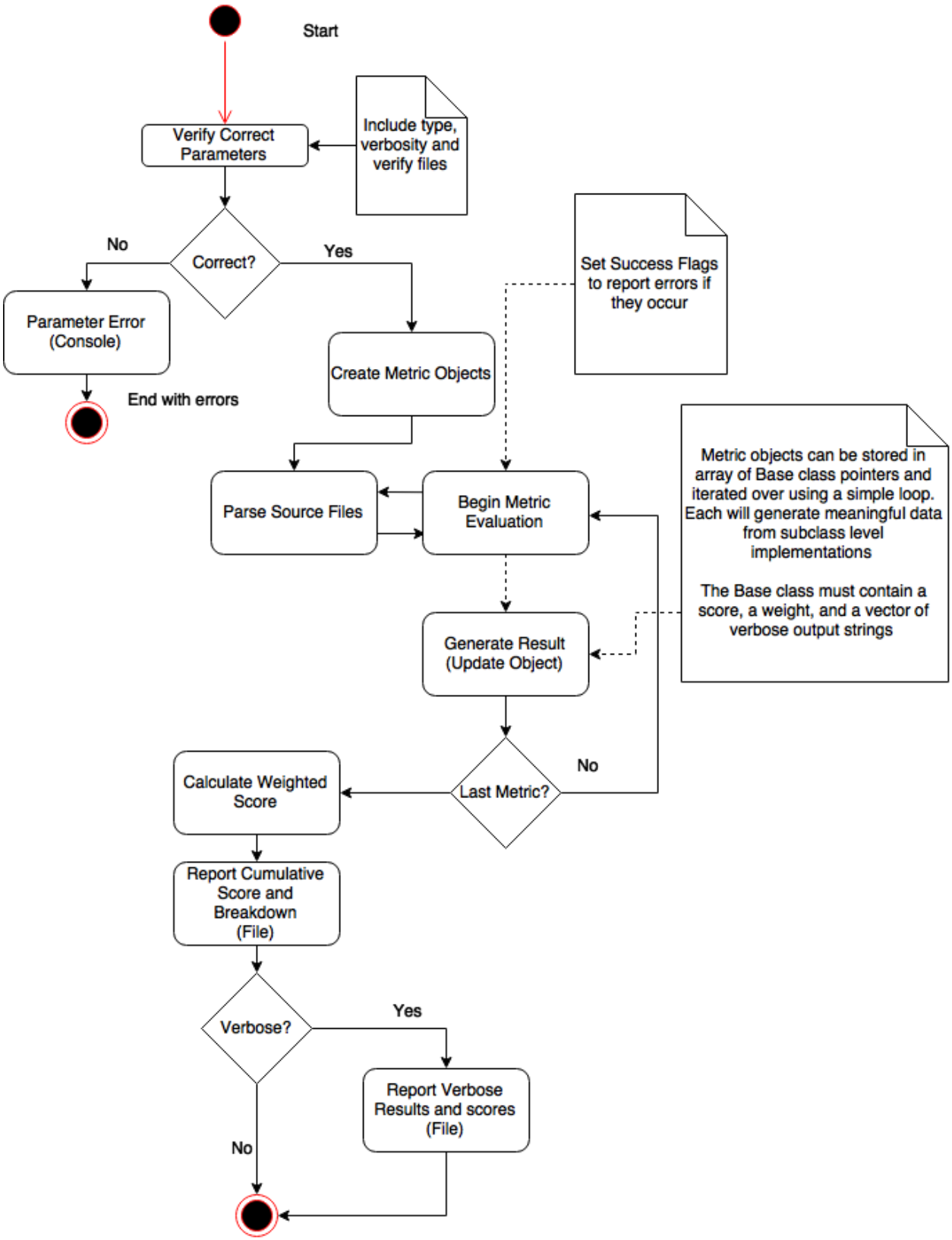
Parsing

After looking into available options for parsing C++ code, we decided not to use an external library. Due to the restraints of the options we looked into, there was no way to include any of the parsing engines that we found in a meaningful manner to obtain data we could use. Although tools such as Elsa (one of the strongest contenders) are good for high level analysis, the backend requirements of building the project made it difficult to make any sort of implementation a reality. Additionally, we looked into exploiting the abstract syntax tree generated by CLang, but were met with similar issues, not to mention this would require running multiple compilers side by side.

In our final design, we handle parsing on a metric by metric basis. That is to say, each metric parses the code base on its own using either a char by char search or some application of REGEX. This was done for two reasons. First, it enables highly specific syntax matching for each sought after pattern, and second, removes any dependence on a centralize data format. This way, we can adapt the test to the files and hone our parsing for each test, rather than try to read in the entire code base worth of data all at once.

Basic Control Flow and Approach

The order of events in this program is fairly straightforward. Initially, it checks the command line arguments it was issued to ensure proper execution. Next, a directoryIterator object is instantiated, creating a list of all files that need to be parsed and checking that all are readable. Next, each metric object is created and executed, generating the score for each as is created as well as logging its verbose output information. Finally, all of the metrics are combined to create a compound score using the value and scoring weight assigned to each metric. This score is output to a file and depending on whether or not the verbose flag was used, output the verbose records as well.


Start
Verify Correct Parameters
Include type, verbosity and verify files
Correct?
No
Yes
Parameter Error (Console)
End with errors
Set Success Flags to report errors if they occur
Create Metric Objects
Parse Source Files
Begin Metric Evaluation
Metric objects can be stored in array of Base class pointers and iterated over using a simple loop. Each will generate meaningful data from subclass level implementations
The Base class must contain a score, a weight, and a vector of verbose output strings
Generate Result (Update Object)
Last Metric?
No
Calculate Weighted Score
Report Cumulative Score and Breakdown (File)
Verbose?
Yes
Report Verbose Results and scores (File)
No

Metrics

There are many types of analysis we can run, but for the sake of this project, we focused on two main areas The first classification of metric pertains to the readability of the code in question and the style used to create it. This is important to us as it lends the codebase to better maintainability and makes it accessible to other users. Here, we look at the non-compilation-impactful aspects such as white space,variable and function naming, and the ratio of lines of comments to lines of code. Together, these present us with an idea of how readable the code under review is.

Second, we turn to the more impactful, but more complicated aspects of the code. Here, we have developed a method for approximating the quality of the comments we find by running basic analysis on the english words used in the author's comments. Finally, we have developed a new type of metric, similar to a hybrid of Cyclomatic Complexity and Halstead Complexity that attempts to approximate the degree of fan out one can expect from a best case run of the code base.

Comment to Lines of Code Ratio

This is one of the most common metrics used in industry today. Although simple in concept, it allows the user to see what ratio he or she is maintaining in their code. In our experimentation, we discovered that most large projects maintain a ratio of about .1, or one comment per ten lines of code. With this in mind, we normalize the calculated score ratio against this value and return a scaled version of this value as the score for this metric. In our verbose output, we record the filename, number of lines sampled, and number of comments found in each record. This will allow users to easily see which files they should work on when trying to increase comment completeness.

Comment Quality

Something that we felt was often neglected by code analyzers was the quality of the comments that were being looked at. Just knowing the number of comments is not enough, we attempted to better understand the impact of each comment by looking at its linguistic makeup - the usage of part of speech.
What we looked for in a comment was succinctness and fullness of the comments. It should not require a lot of context to understand except for the code it is commenting on, and easy to understand. To quantify this, we first sampled comments from various codes that meets those criteria, and found out the ratio of each part of speech.

The process we employed was to parse the file for comments, and pull every words from given comment base, sure to drop punctuation and formatting. Once parsed, the words are added to list where they are matched against specific type of conditionals and list of words in avl tree to determine its part of speech. This means that modifying words (adjectives and adverbs), nouns, and verbs can be separated out of analysis. By weighing the ratios of different types of words compared to one another, we can see that the author made an attempt to make meaningful comments that are informative to the reader. This means that programs with stub comments consisting of poorly constructed English will not score as highly. Verbose output for this metric consists of the ratios of word classifications relative to one another for each file it finds. Lastly, we took into account the lines of code between each comments. If comments are in block and code between comments gets too big, the comment may not be doing its job of clarifying the code.

The simulation showed most of succinct and full comments has noun ratio between 50% and 70%. If less than 50% of noun to total words ratio suggests that comments in the code are too long, and most likely not giving information regarding the code base itself. Lower ratios were often detected in copyright declarations at the top of the file due to increasing amount of verbs, modifiers and fillers from usage of full sentences. If more than 70% of comments are nouns, this suggests fragmented words and phrases most often written to describe basic logic of a code that could be easily derived from the code itself. Also most of these comments were hard to understand without context, and often related to describing the function of poorly named variables in code.
Also, succinct and full comments had verb ratio between 15% and 35%. Lower ratio is related to higher percentage of noun usage which relates to context-specific and trivial comments, and higher ratio suggests complicated sentences. Often, ratio out of this bound suggests short comments. For instance, through simulation on multiple codebases we found that the ratio of verb often exceeds 50% or is below that comments of total word less than thirty. Modifiers marked 10% to 20% of entire comment base, and conjunction and prepositions marked less than 10% in all tested codes. Higher ratio of these reduces succinctness and informativeness of the comment.

Variable and Function Naming

Like the previous metric, we wanted to look at the quality of the variable and function names used in the code base. Unfortunately, since these keywords cannot be tokenized as easily, it is very difficult to try to establish the linguistic merit of these terms. We can however, look at the size of the declarations to establish a best case meaning for them. That is to say that it can be assumed that “good” names come in a range of acceptable

lengths and follow certain letter patterns. By analyzing length and character composition, we can determine with a fair amount of certainty. From a great deal of research on industry standard and experimental data on what we consider to be good names, we calculated a ranges that most reasonable variable and function names fall in. This rewards the user for using well conditioned names that are meaningful and are not just shorthand placeholders.

The Length of name less than 5 characters are often uninformative. string data, int i, int size… these names requires context to understand what they do. Instead, string string_data, int loopCounter, int vectorSize gives much more information. On the other hand, too long of a name will decrease the legibility as well. Void calculateVariableAndFunctionNameQualityAndOutputScore() is very informative, but legibility-wise it is an eyesore. Lastly, name with all lowercases are often abbreviated also unclear. void calc is hard to understand. String checkAName or string check_a_name has much more legibility.

We checked to see if the length of the name is of good length (between 5 ~ 22) and if there were any abbreviations (more than three consecutive lowercase consonants) for scoring.

Function and Line Size

This metric is concerned purely with of the formatting of the code. It reads in a file and ensures that all lines are less than eighty characters long, making deductions for each line that is not in accordance. This discourages the user from using lines that are unnecessarily long and rewards them for breaking them up into more digestible pieces. It then makes a second pass, determining the number of lines in each function.

Delatation

In the interest of discovering something about the design of the code itself, we have invented a new metric related to the function structure of the scrutinized code. Delatation, derived from the shape of a river delta, is a metric that tries to describe the fan out of functions if they were to be drawn as a tree, where each function was a node and each child was a function called within the parent. This is a best case metric, as it used average values across the code base and the total number of function calls to approximate the “area” the tree would take up. To better understand this, it is easiest to think of the function tree as a triangle that can be approximately described by the logarithm of the total number of functions declared (height) multiplied by the logarithm of the average number of in function calls in each function squared, all divided by two. This

models a simple unit subtree consisting of one function and all of its calls, scaled to represent the total size of the codebase.

This metric rewards the programmer for creating large number of separate functions and balancing that number against the number of calls made in each function. Extremely skinny triangles, that is programs that have a great deal of functions, but low call count per function, typically indicate an excessive amount of functions created, that may make the code difficult to understand due to high stack depth. Wide and short triangles, or code that has a high average call per function cound, are indicative of code with long functions that may be needlessly complex, and thus could benefit from being broken down into smaller functions. The highest scores come from code that maintains a good balance between these two numbers, giving the largest “area” for a given codebase functionality.

In testing our software, we found that most codebases had a dilatation value of around fifty, so we used this to normalize the dilatation value when creating scores. Another issue we ran into was improper parsing of the function calls within the function. In many cases, strange syntax would lead to a fault in our parsing method leading to a high relative error in our code. To combat this, the errors are counted up and used to scale back the weight assigned to the metric in final reporting. This way, high error count code does not negatively affect the final score of the code as much.

Annotated Examples

We ran our code on three different code base: First, Seung Ki Lee's sprint 01. Second, proxygen codebase from the demo. And third, folder comprised of over two hundred c/cpp files.

For the first code base, we anticipated high scores on delatation, code format, and lower score on comment ratio and comment quality with average score on variable and function name. Given that this was academic assignment, we assumed comments and naming conventions to be acceptable. With long functions and code design where main was doing most of the work, we assumed the delatation and code format scores would reflect it.

Surprisingly, the overall score came out to be 16, with variable and function naming and delatation being the highest scores. As the analyzer was ran against our own code base, and since we were already following the convention ourselves, comment quality and comment ratio showed significantly low range. As expected the delatation showed the highest score. However as the codebase itself was small, the absolute number of bad conventions were smaller and resulted in lower score. Our scoring system which rewards small number of deducting factors worked in favor of small sized file.

For the second code base, we anticipated lower score on comment ratio and quality as the codebase is from facebook, and has more contributors. Overall, we assumed the score on proxygen folder would be much lower than our project.

Overall score was surprisingly 52. With higher score on delatation and variable and function name, this showed that even industry level codes have problem with bad conventions. We noticed that comment quality had higher range due to fragments of comments and a lot of files had no comments at all, which increased the scores. A lot of files had very small number of comments resulting in abnormal ratios for comment quality.

Lastly, on a code base with 200+ files the assumption was that it would have lower score than proxygen. The files were well organized and from looking through few of the files, they were well commented with decent ratio. The overall score turned out to be 10, proving our assumption right.

Conclusion

Our goal in this project was to create a useful tool in a robust and creative new way. We implemented an Object-Oriented solution that allowed us to create five metrics of varying scope. We learned a great deal about what makes code good, and what we can do as programmers to analyze other codebases. Using the five metrics we have designed, we hope that our tool will provide insight into the quality of any codebase it is used on to help the author of any codebase work to a better solution.

We would like to take this opportunity to thank Professor Fontenot as well as the Data Structures TA staff for helping us to master the material we needed to complete this project. Together, our work has taught us a great deal about software and making good code, as well as the techniques, algorithms, and data structures needed to create our analysis tool.