Lyle School of Engineer
Computer Science
Cyber Security Department

GoodEnough Report

Chi Tran – Patrick Lin – Kevin Thompson

1. Architecture

The software is written around a primary loop in GoodEnough.cpp, which, after establishing a valid input and output file from the command line, loops through each file matching the extensions indicating that it is part of a C++ codebase and running it through 5 analysis. The metrics are subclasses of a superclass Metric. It then outputs the results to the specified file.

2. Metrics Overview

- ### *Code Reuse*

Repeated lines of similar code can be a sign of poor design; ideally a procedure that needs to be done more than once can be looped or put into a function. Of course, some repetition is impossible to avoid.

My implementation scans one line of code at a time, trimming whitespace (to identify otherwise-identical lines of code), and uses a hash table to store the lines of code and identify if they have been seen before, counting the number of times they have been seen.

The score is determined by adding two points for each time a line of code is repeated at least once.

- *Complexity*

Determine the complexity of the code by seeing how deep we go past the first {} in each function. If functions are too deep then the code may be too complicated.

Implementation scans the code one character at a time, looking for "{" to indicate the start of a section of code, and counting the nested number of "{" until it returns to a 0 balance. It records the depth of each function identified this way.

The score is determined by the function (averageDepthOfFunction - 1.5) * 20. That is, our ideal average is 1.5.

- *Function Size*

Determine the average size of functions; functions that are very long are likely not efficient.
The implementation scans the code 1 character at a time, looking for the entry point to functions and then counting the number of lines until the function has ended.

The score is determined by the function (averageLength - 30 * 1.2); that is, our ideal is 30 lines of code per function. If the average is less than 30, there is no score penality.

- *Lines of Code*

I count the literal lines of code per physical line of code. Many different operations on a single line leads to increased complexity, decreased readability and mistakes.

The implementation goes through the codebase character by character, looking for comments (to ignore anything inside of them), then looking for '}' or ';' characters to indicate a literal line of code, counting the number of them per physical line (which ends at a newline).

The score is determined with the formula (averageLines - 1.05) * 500; that is, ideal is 1.05 per physical line of code.

- *Variables*

A metric judging variable and function names by their deviation from various standards, or if there is a consistent standard used.

The implementation looks for variable/function names by looking for words indicating a type (e.g. “bool”, “char”, “int”) and then analyses those names. It checks to see if they are in a CamelCase format, a delimited_format, or if they significantly deviate from any standard (e.g. AVariableNameThatISWEIRD). It also counts the length of the variable/funciton names.

If CamelCase or delimited_format is primarily used, there is no score penalty. If they are both used relatively often, there is a penalty of 2 points for each percentage point you are too close (the worst being 50%/50%. If there are nonsense variables, you are penalized. If the average character length of functions/variables is over 10, you are penalized 5 points for each full number over 10.

3. Metrics Implementation

- *Code Reuse*

One line of code at a time is input by getline(), and then remove_if (from <algorithm>) is used to remove all spaces from that line. I then check if this line is in the hashtable; if not, we add it with a count of 1, and if they are, we increment the counter.

The score is determined by adding two points for each time a line of code is repeated at least once.

- *Complexity*

“Complexity” works by checking how many brackets are opened inside of each function; that is to say, the more “levels” deep the logic goes, the more complex the function is. “Complexity” does not check if the code compiles correctly (that would be the compiler’s job) – “Complexity” only checks to see how deep the logic goes in each individual function.

To make the program recursively get all the files in the directory, instead of using Boost, we used opendir() from dirent.h, which is part of the C standard. We used dirent.h to use opendir(), which gives us a list of all the files in the directory specified. And then, we used stat() to check if each file a file, or a directory. If it is a file, the program will check for the correct extensions .cpp/.hpp/.h/.c, and then scan them with the metrics. If it is a directory, it is put onto a queue of strings, and while that queue is not empty it used a goto to return to the top of the function and open the next directory.
We avoided recursion in the scan function because we wanted to maintain the metrics; originally it was going to be a recursive function

- *Function Size*

The code is scanned one character at a time with .get() off of the input stream. If the character is a '{', we increment depth; if it is '\n' we increment length, and if it is '}' we decrement depth. If we determine we are outside of a function in this fashion, we add up the total number of lines and add it to the average.

The score is determined by the function (averageLength - 30 * 1.2); that is, our ideal is 30 lines of code per function. If the average is less than 30, there is no score penality.

- *Lines of Code*

The code is scanned one character at a time with .get() off of the input stream; the previous character is stored in another variable too. If the two characters are '//', we note that we are in a comment. If they are '/*', we know we are at the start of a block comment. If they are '*/', we are at the end of a block comment. If it's a newline character we check if we are in a comment, and if not we count it as a newline. If it is a '}' or ';', we increment our count of literal lines of code.

The score is determined with the formula (averageLines - 1.05) * 500; that is, ideal is 1.05 per physical line of code.

- *Variables*

The implementation looks for variable/function names by looking for words indicating a type (e.g. "bool", "char", "int") and then analyses those names. It checks to see if they are in a CamelCase format, a delimited_format, or if they significantly deviate from any standard (e.g. AVariableNameThatISWEIRD). It also counts the length of the variable/funciton names.

We use isupper() to check if there are uppercase letters, getline() to pull a full string at a time from the file, then rotate through it one variable at a time. We search for each full word that we find (with a whitespace after it) in our AVLTree to see if it is a variable name.

If CamelCase or delimited_format is primarily used, there is no score penalty. If they are both used relatively often, there is a penalty of 2 points for each percentage point you are too close (the worst being 50%/50%. If there are nonsense variables, you are penalized. If the average character length of functions/variables is over 10, you are penalized 5 points for each full number over 10.

4. Analysis

✓ One of previous projects

```
Literal Lines of Code Score: 2
 Average Literal Lines of Code: 1.04439
 Total Physical Lines of Code: 428
 Most Literal Lines in Physical Line of Code: 3
Code Re-Use Score: 30
 Number of reused lines: 15
 Average # of Lines of Reuse: 5.2
 Most reused line # of times: 23
 Most reused line: template<classT>
Overall Complexity Score: 30
 Average Complexity: 1.65116
 Deepest Function Depth: 5
Function Size Score: 0
 Average Length of Function: 20
 Total Function Length: 860
 Number of Functions: 43
Variable/Function Quality Score: 48
 CamelCase names: 17
 Delimited names: 4
 Non-standard names: 4
 Number of names total: 61
```

✓ caffe

Github: **https://github.com/BVLC/caffe**

- Caffe: a fast open framework for deep learning.

```
Literal Lines of Code Score: 22
 Average Literal Lines of Code: 1.09526
 Total Physical Lines of Code: 33969
 Most Literal Lines in Physical Line of Code: 6
Code Re-Use Score: 100
 Number of reused lines: 2686
 Average # of Lines of Reuse: 10.236
 Most reused line # of times: 622
 Most reused line: template<typenameDtype>
Overall Complexity Score: 30
 Average Complexity: 3.25828
 Deepest Function Depth: 14
Function Size Score: 100
 Average Length of Function: 173.486
 Total Function Length: 78589
 Number of Functions: 453
Variable/Function Quality Score: 8
 CamelCase names: 1676
 Delimited names: 1999
 Non-standard names: 485
 Number of names total: 12345
```

✓ LaiNES

Github: https://github.com/AndreaOrru/LaiNES

- Cycle-accurate NES emulator in ~1000 lines of code

```
Literal Lines of Code Score: 100
 Average Literal Lines of Code: 1.27225
 Total Physical Lines of Code: 2641
 Most Literal Lines in Physical Line of Code: 17
Code Re-Use Score: 100
 Number of reused lines: 95
 Average # of Lines of Reuse: 4.62105
 Most reused line # of times: 12
 Most reused line: private:
Overall Complexity Score: 30
 Average Complexity: 1.66351
 Deepest Function Depth: 6
Function Size Score: 0
 Average Length of Function: 25.9194
 Total Function Length: 5469
 Number of Functions: 211
Variable/Function Quality Score: 27
 CamelCase names: 146
 Delimited names: 276
 Non-standard names: 129
 Number of names total: 1399
```