Final Project Report

CSE 2341

Jacob Hillman
Will Kastner

The Metrics

1. **Initialized Variable Metric & Variable Name Metric**

The Initialed Variable metric determines whether a variable has been initialized to a value upon declaration. Uninitialized variables are a common source of issues in code. An object is randomly assigned a value if one is not deliberately given to it, which can lead to strange problems that are difficult to debug.

The Variable Name metric examines the variable names of a project and how well these names follow the naming conventions in terms of variable length and variables without numbers. We chose this metric because it involves objectifying something subjective and reading source code can become cumbersome if variable names are not distinguishable and informative.

We implemented these metrics under the same class in order to avoid parsing the file for variables multiple times. The class only checks for variables of primitive types, which are held in a vector that is initialized in the constructor. The runMetric function has three buffers that will store the type, name, and initialized value of each variable that is located. It also keeps track of whether or not the variable was initialized when declared or if it is a parameter. It stores all variables as Variable objects in an AVL Tree (described in architecture section). The parsing functions check for numerous cases of variable declarations, such as if the variable is declared within a function header, if it is a function declaration in the function definition, and if the variable is found in a for loop (which is ignored assuming it is a temporary variable such as *i*). The function also accounts for multiple variables that are declared in the same declaration, array variables, and pointers declared in the following ways: int* ptr, int * ptr, int *ptr (as well as double pointers).

The scoring for the Initialized Variable metric is very straightforward. It is simply the number of uninitialized variables divided by the number of variables declared multiplied by 100. Since this may produce very high scores for smaller files that contain less variables, the overall score for the directory is weighted based on how large each file is, and therefore, the higher scores of small files are diluted by scores of the larger files.

The scoring for the Variable Name is calculated based on whether or not the variables fall within a "reasonable length" range of a variable name. The range was decided to be between 8 and 20 characters long based on reading a number of articles about variable naming best practices. The program calculates the average length of variables in a file and counts the number of variables with lengths below, within,

and above the length range. The score for this metric begins at zero. To score the naming conventions of the file, magnitude amount is determined by taking the percentage of a single variable to the total number of variables and multiplying it by 50. Every time a variable containing numbers, a short variable, or a long variable is found the magnitude amount is added to the file score. Additionally, if the average variable length is outside of the length range 15 points are added to the score. For both metrics, if there are no variables found in the file, the score is set to -1 and ignored.

2. In-Line Comment Metric

In-Line Comment metric examines the length of comments that are found at the end of a line (after some functional code that is written). If included at all, the length of these comments is very important towards the aesthetic of a document. Long end-of-line comments are a signifier for an inexperienced coder.

This metric takes in each line of code and searches for the "//" symbol, and counts the characters that follow the symbol. The ideal average for these characters was calculated to be 34 characters. This number comes from running the metric on over 1,500 files from a total of 1,187 contributors. The standard deviation on the mean was calculated to be 8, so averages of between 26 and 42 are ideal. The greater the average varies from this range, the higher the score from this metric.

3. Indentation Metric

Indentation metric checks to see if the indentation of the file follows common indentation practice. Correct indentation is extremely important for readability of a file, and is a characteristic of professionalism. We choose this metric because it provides concise and useful feedback for the user.

Primarily, this metric works by checking for '{' and '}' characters. Usually, these characters signify that the next line of code needs to be either one level greater or one level less of indentation, respectfully. This metric is primarily made up of a recursive function called CheckIndentation. Whenever it is determined that the code following a specific point should be one level of indentation greater, the function is called. An integer keeps track of the level of indentation that is appropriate at any given time. This number is multiplied by four to obtain the total amount of whitespace that should be found at the beginning of a line (with the "\t" character counting as four whitespaces).

There are also special cases that this metric tests for, such as single line loops or if-statements that do not require braces, and the declarations such as a "private" and "public", which do not follow standard indentation practice. These exceptions are usually marked by the absence of a semicolon at the end of the line. This metric takes away all comments and whitespace for the end of a line, and checks to see if the line contains a semicolon. If it does not, the metric now knows that this a special case.

Scoring for this metric is very simple. The ideal number of incorrectly-indented lines can easily be determined to be zero. Therefor, a file is penalized for every instance of incorrect indentation. If verbose output is requested, the metric outputs the line numbers for every violation in the file.

4. Coupling Metric

Coupling metric determines the number of user-defined classes used by each file. According to Microsoft, "studies have shown that an upper limit-value of 9 is the most efficient." If a file contains 4 or less user-defined classes, the score is zero. If it contains between 5 and 9, than the score is five times the number of classes as it is approaching the upper bound. Lastly, the file is heavily penalized if the number of classes exceeds 9, with an additional 20 points added. A file should limit the number of user-defined classes that it uses because these classes limit the reusability of a file. It is difficult to use a file with many user-defined classes in a context outside of the specific project that is being tested.

The user-defined classes are determined in a function within the DirectoryParser class. When the parser goes through a directory to determine file names, it also checks within the files to see if they implement a new class. If so, the class name is stored in a Vector. Within the metric object, each line of the file is cross-referenced with every class in the vector. If a declaration of one of these objects is found, an integer numClasses is incremented.

The Architecture

The program is designed to run through a class named DirectoryParser. The purpose of the DirectoryParser class is to open up all of the C++ source code files in a given directory and store the file information in a HashTable. The HasthTable object in the DirectoryParser class uses the path names (which were also stored in a vector) as keys and File objects as values.

The File class is designed to hold any useful information from a file in the directory. This class stores the path, name, number of lines, score, and metric information. Additionally, it has functions to run through all of the metrics and combine the individual score into an a "file score." The metric information is stored in a vector of MetricInfo objects (described below). The MetricInfo vector is set through a function in the DirectoryParser class.

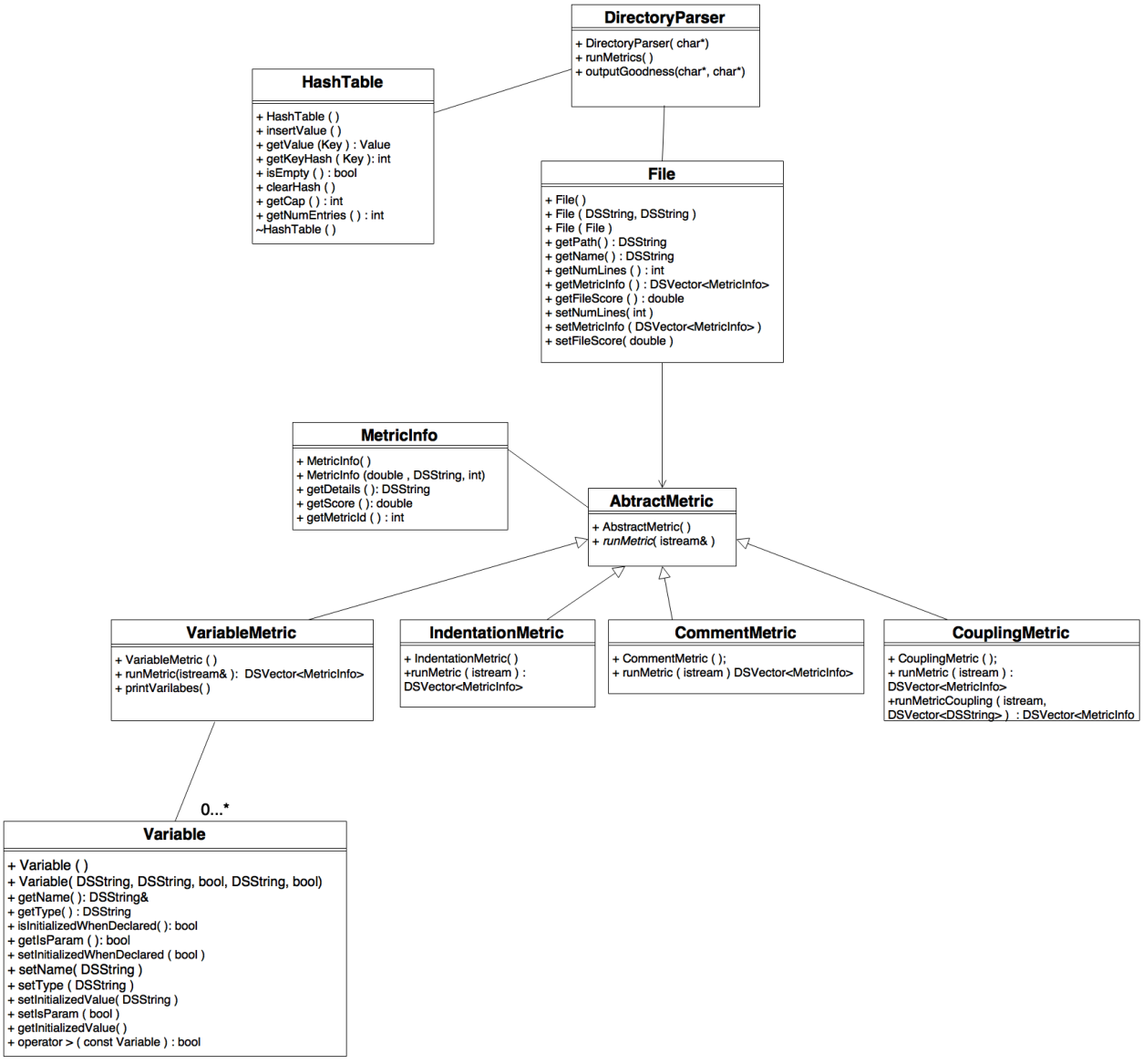
The MetricInfo class is designed to hold the information for each metric individually for a given file. The object has a score, a string that represents the details of the score, and an ID that represents which Metric the object represents.

After the DirectoryParser class fills the hash table, the runMetrics function goes through each file in the HashTable and processes all five metrics for that file. The runMetrics function creates a vector of MetricInfo objects and an array of AbstractMetric pointers. The AbstractMetric class is the base class of all four metric classes. These metric classes inherit the runMetrics pure virtual function from AbstractMetric. Each pointer of AbstractMetric is initialized to an object of each Metric class (VariableMetric, IndentationMetric, CommentMetric, CouplingMetric) whose implementation was described in detail at the beginning of the report. Then, the runMetric function is called for each pointer, and it returns a vector of MetricInfo objects, which is added to the vector created in the DirectoryParser function. Once the vector in the DirectoryParser runMetrics function is filled with information from each metric, the MetricInfo vector is copied to the File object.

Finally, the outputGoodness function in the DirectoryParser class determines whether to print the brief output or verbose output and then collects the scores from all of the File objects. Then, the program will print out the highest 10 scores by accessing the File objects one last time.

Additionally, the VariableMetric class uses the Variable class to keep track of any useful information necessary for the variables in its AVL Tree. A Variable object contains the name, type, and initialized value of a variable. It also holds boolean variables that recognize whether it is a parameter and if it was initialized when declared.

A UML diagram of the architecture is shown below to provide more detail.



** Only the public functions are shown for each class **

Annotated Analysis

Directory Tested:
Sprint 4 - Jacob Hillman
Number of source code files: 17

```
The overall score of the directory containing 15 files was 20.63.

Below are the details for the highest 10 file scores:

------------------------------------------------------------------------
For file "DSString.h", the file score was 49.8. The details are shown below:
------------------------------------------------------------------------
There were 2 variables uninitialized out of 2 variables declared.
    Metric 1 Score: 100

The average length of a variable is 3.5 and the number of variables
containing numbers is 0. The total number of variables in the file is 4.
    Metric 2 Score: 65

There was no incorrect indentation in the file.
    Metric 3 Score: 0

The Average number of characters for in-line comments is 58.8 out of 5 comment(s).
    Metric 4 Score: 83

There are 1 user-defined classes used in this file.
    Metric 5 Score: 0
```

The overall score determined for this directory is 20.63. This is the result of a **weighted average** based on the number of lines in a file compared to the total lines of all files. The more lines of code a file has, the more it's individual score counts towards the average. The ten files with the highest scores are listed along with a breakdown of the score of each individual metric in this **verbose output.** If this were a brief output, only the top line containing the overall score would be printed.

Our program caught several mistakes in this first file "dsstring.h", and the output exemplifies the program's processing of a file score. The mistakes in this file are highlighted in **code example 1**.
The following mistakes were recognized by the program:

1. Both integers should be initialized to a value.
2. The variable names are short and undescriptive.
3. The inline comments are way too long.

It can also be noted that the indentation metric correctly identified correct indentation, even in the case of private member declarations.

Code Example 1 - "DSString.h"

```
private:

    char* data; // Pointer to the c-string storing the characters that make up the string
    int len; // Stores the number of letters in the word +1 for the null terminator
    int cap; // Represents the maximum number of letters a word can contain
```

Another one of the ten files described is "word.h", which is shown on the next page.

```
------------------------------------------------------------------------
For file "word.h", the file score was 45. The details are shown below:
------------------------------------------------------------------------
There were 1 variables uninitialized out of 2 variables declared.
    Metric 1 Score: 50

The average length of a variable is 2.2 and the number of variables
containing numbers is 0. The total number of variables in the file is 5.
    Metric 2 Score: 65

There was no incorrect indentation in the file.
    Metric 3 Score: 0

The Average number of characters for in-line comments is 9 out of 1 comment(s).
    Metric 4 Score: 85

There are 5 user-defined classes used in this file.
    Metric 5 Score: 25
```

In this portion of the output (**code example 2**), the program caught that String is uninitialized. Also, several of the variable names (especially in the parameters) are very short. This class uses a total of 5 user-defined classes, including Article, String and Vector (which are both user-defined in this example).

Code Example 2 - "word.h"

```
class Word
{
private:
    String name;
    Vector<Article> articles;
public:
    Word(String in)
    {
        name = in;
    }

    Word(String in, Article article)
    {
        name = in;
        addArticle(article);
    }

    bool operator> (const Word& rhs)
    {
        return name > rhs.name;
    }

    bool operator< (const Word& rhs)
    {
        return name < rhs.name;
    }

    void addArticle(Article in)
    {
        if(!articles.contains(in))
            articles.add(in);
    }
```

Directory Tested:
Data Structures Final Project - Jacob Hillman and Will Kastner
Number of source code files - 30

```
The overall score of the directory containing 26 files was 20.13.

Below are the details for the highest 10 file scores:
```

The overall score of this project is 20.13, and ten files were listed. The output for "dsvector.h" is shown below.

```
------------------------------------------------------------------------
For file "dsvector.h", the file score was 27.3528. The details are shown below:
------------------------------------------------------------------------
There were 2 variables uninitialized out of 11 variables declared.
    Metric 1 Score: 18

The average length of a variable is 4 and the number of variables
containing numbers is 0. The total number of variables in the file is 16.
    Metric 2 Score: 65

Incorrect indentation is found in line(s):
16 18 19 20 22 23 24 25 26 28 30 31 32 33 34
36 37 38 39 41 42 43 45 47 48 49 50 51

    Metric 3 Score: 26

There were no end of line comments in this file.
    Metric 4 Score: -1

There are 2 user-defined classes used in this file.
    Metric 5 Score: 0
```

In **code example 3** (on the next page), the indentation errors caught by the indentation metric can be observed. When declaring private and public members in a function, "private:" and "public:" should not be tabbed. Therefore, the declarations themselves should only have one level of indentation. In this example, lines 16-51 are said to be incorrectly indented because "private" and "public" are tabbed and the code that follows has two levels of indentation.

The following message is printed in this block of output for the end of line comments (Metric 4):

```
There were no end of line comments in this file.
    Metric 4 Score: -1
```

If a metric is determined to be ineffective in the calculation of the score for a file, then the metric returns the score as -1. If the score is -1, it is not counted towards the file's score. In this file, since there were no end of line comments in the file, the metric does not count towards the score.

Code Example 3 - "dsvector.h"

```
template<class T>
class DSVector{

  public:

      DSVector();
      DSVector(int);
      DSVector(const DSVector<T>&);

      void add(T);
      void add(T, int);
      void remove();
      void remove(int);
      T get(int);

      ~DSVector();

      int size() const;
      int capacity();
      void sort();
      bool isEmpty();
      void clear();

      T& operator[](int) const;
      DSVector<T>& operator=(const DSVector<T>&) ;
      DSVector<T>& operator+=(DSVector<T>&);
      bool operator==(DSVector<T>& );

      DSVector<T> intersect(DSVector<T>& ) const;
      DSVector<T> unite(DSVector<T>& ) const;
      DSVector<T> subtract(DSVector<T> & ) const;

  private:

      bool checkVctr(T);
      void resize();
      T* data = nullptr;
      int vsize;
      int cap;

};
```

Directory Tested:
tesseract-master - Google
Number of source code files - 627

```
The overall score of the directory containing 627 files was 26.63.

Below are the details for the highest 10 file scores:
```

The overall score of this project containing 627 files is 28.33, and the ten highest scoring files were printed to the text file.

The file "cube_reco_context.cpp", whose output is shown on the next page, exhibits a couple of key features of the program's output.

```
------------------------------------------------------------------------
For file "cube_reco_context.cpp", the file score was 52.3462. The details are shown below:
------------------------------------------------------------------------
There were 2 variables uninitialized out of 4 variables declared.
    Metric 1 Score: 50

The average length of a variable is 9.25 and the number of variables
containing numbers is 0. The total number of variables in the file is 4.
    Metric 2 Score: 12

Incorrect indentation is found in line(s):
43 44 45 46 47 48 49 50 51 52 53 54 59 60 61
64 65 66 69 70 71 74 75 76 79 80 81 84 85 86
89 90 91 99 100 117 118 119 123 124 125 128 192 196 197
202 203 204 206

    Metric 3 Score: 94

The Average number of characters for in-line comments is 25 out of 2 comment(s).
    Metric 4 Score: 5

There are 17 user-defined classes used in this file.
    Metric 5 Score: 100
```

There are two main things affecting the score of this file: indentation and coupling. It should be noted that the coupling score has been capped at 100. As seen in **code example 4**, the indentation of this file is much different than what the file is checking for. In general, larger projects tend to have a higher score for the indentation metric because they often times have very different indentation standards than our program. Our indentation metric searches for the ideal simple indentation, and these larger projects tend to use other indentation practices that are not as easy to read.

For example, in **code example 4**, the indentation beginning after the bracket in line 42 is only 2 spaces per line. Since this does not fit the traditional model of 4 spaces (or a tab) per line, the indentation metric records the line numbers and adjusts the score accordingly.

Code Example 4 - "cube_reco_context"

```
33  namespace tesseract {
34
35  /**
36   * Instantiate a CubeRecoContext object using a Tesseract object.
37   * CubeRecoContext will not take ownership of tess_obj, but will
38   * record the pointer to it and will make use of various Tesseract
39   * components (language model, flags, etc). Thus the caller should
40   * keep tess_obj alive so long as the instantiated CubeRecoContext is used.
41   */
42  CubeRecoContext::CubeRecoContext(Tesseract *tess_obj) {
43    tess_obj_ = tess_obj;
44    lang_ = "";
45    loaded_ = false;
46    lang_mod_ = NULL;
47    params_ = NULL;
48    char_classifier_ = NULL;
49    char_set_ = NULL;
50    word_size_model_ = NULL;
51    char_bigrams_ = NULL;
52    word_unigrams_ = NULL;
53    noisy_input_ = false;
54    size_normalization_ = false;
55  }
```