Object-oriented metrics suite for Quality Software Development and Maintenance

Anirudh Sharma, Masters of Computer Science, NUIM, Ireland
(email: Anirudh.Sharma.2015@mumail.ie)

***Abstract*— A software metric is a quantitative measure of a degree to which a software system or process possesses some property. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments. The project aims to give the values for the code characteristics which can be used during and post development phase. The fluctuation of these values will analyze the performance and object oriented approach of the code.**

I. Introduction

Understanding how a software system is structured, and what are the different object oriented characteristics it is been used is crucial for software comprehension. It allows developers to understand an implemented system and reason about how non-functional requirements are addressed. The information about these elements could also help us in the maintenance of the software and also give us an idea about how development is evolving. Yet, many systems lack any architectural documentation or any information about different object oriented characteristics usage, or it is often outdated due to software evolution. In current practice, the process of maintaining software or monitoring its different characteristics relies primarily on developer knowledge. Towards this goal, I analyze the usefulness of adopting different code-level characteristics to reflect the importance of these characteristics in the maintenance and monitoring of the software development. These code level characteristics is been taken from one of my selected research paper which is "Revealing the Relationship between Architectural Elements and Source Code Characteristics [1]".

II. Description Of The Problem

As the software evolves and becomes complex it becomes hard to keep track of its different object oriented characteristics by looking at the source code. Now as these characteristics could play a very important role while doing refactoring of different components and also in the maintenance phase of the software. Therefore, there should be some way of looking at these characteristics in a condensed manner.

III. Experiment Description

To get these code level characteristics, in this experiment I use the Asm java library to directly look at the source code of the given piece of software. In this way the developers no more have to depend upon scanning through the source code to know about code level characteristics.

A. Characteristics

These characteristics have been taken from one of my selected paper "Revealing the Relationship between Architectural Elements and Source Code Characteristics" for this module.

Two kinds of characteristics used were "Direct Characteristics" and "Indirect Characteristics".

TABLE I
DIRECT CHARACTERISTICS

Number of Static Attributes (NSA) is the count of the static attributes in a class. It provides information related to the role that elements play in the architecture, as static attributes commonly appear in classes that state definitions.
Number of Children (NSC) is the count of a class's direct subclasses. This characteristic represents information about the structural organization of the system and focuses on quantifying the number of classes that directly depend on the class under measurement. Consequently, elements with high NSC value have more generic roles in the system than those with low NSC.
Number of Overridden Methods (NRM) is the number of redefined methods when the evaluated class is a child class. This metric measures how much a child class differentiates itself from its parent class. It provides structural information of how specific an element behaves.
Number of Attributes (NOA) is the number of attributes of a class. In software applications, data (or domain) classes tend to have more attributes than other classes. Consequently, this characteristic indicates the architecture role of an element, as it occurs in data classes.
Number of Methods (NOM) is the number of methods in a class. Counting the number of methods gives insights about the quantity of processing a class handles. For example, elements in a business module of a layered architecture have higher NOM values than elements of other modules.
Number of Static Methods (NSM) counts the number of

static methods of a class. Static methods occur in classes that state definition, similar to NSA. It is usually related to business rules that are specific to the domain.

Number of Getters and Setters (NGS) is the sum of getters and setters of a class, more specifically, the sum of methods whose name starts with get or set. Data classes have more readable and writeable attributes than other classes, and thus have more getters and setters. Consequently, a high number of getters and setters also indicates the architecture role of an element.

Total Lines of Code (TLOC) is the sum of the lines of code in the class, excluding blank and comment lines. As architectural elements that handle processing may have complex algorithms, they have more lines of code than others. In addition, data elements have fewer lines of code, because their methods are mostly getters and setters.

As described in the paper "Revealing the Relationship between Architectural Elements and Source Code Characteristics" by Vanius Zapalowski, Ingrid Nunes and Daltro Jones Nunes.

TABLE II
INDIRECT CHARACTERISTICS

Class Name Words (CW) are the words present in a class name, which in this paper correspond to the words present in an architectural element. This information provides an understanding of the application domain, because its syntax makes the code legible. Additionally, similarities in the element names indicate to which architectural module a class belongs. For example, the suffix of its name relates the responsibility of a particular class (e.g. the suffix BusinessService). Evaluating all class name words of an application brings too much variability and singularity, so we consider up to the five most frequent words in the application.

Superclass Usage (SC) is the list of all superclasses, from the direct superclass to the Object class, of an architectural element. As with CW, this characteristic retrieves information related to the application domain. In addition, it also provides structural knowledge, as classes performing similar tasks inherit methods from the same superclasses. As above and for the same reason, we consider a characteristic each of up to the five most frequent superclasses of the system.

Interface Usage (IU) is the list of interfaces that an architectural element implements. Similarly to SC, this characteristic relates common tasks of architectural elements. It also considers up to the five most frequent interfaces used in the application. Note that the CW, SC and IU characteristics depend on developers using appropriate names.

Depth of Inheritance Tree (DIT) is the distance from the topmost class (i.e., the class from which all other classes derive — Object in Java). The DIT values start in zero to the top-most class and, for each level of class inheritance, the DIT value increases by one unit. Therefore, this characteristic extracts the level of specialization of an element, complementing the information given by SC.

Weighted Methods per Class (WMC) is the sum of the McCabe Complexity of all methods of a class. The complexity is based on the number of paths linearly independent in the source code decision tree. This characteristic measures the amount of processing handled by an element in order to group them by their quantity of processing.

Mean Methods Complexity (MMC) is the average McCabe Complexity of the architectural element's methods. This characteristic differentiates elements handling much complexity in fewer methods, such as Utils elements, from elements with many methods handling large complexity, usually Model or Data elements.

Mean Method Size (MLOC) is the average number of lines of code of the class's methods, excluding blank and comment lines. MLOC can help to identify elements that share a common behavior, e.g., high MLOC indicates that a class has much logic inside it, consequently it may be a business element.

As described in the paper "Revealing the Relationship between Architectural Elements and Source Code Characteristics" by Vanius Zapalowski, Ingrid Nunes and Daltro Jones Nunes.

The above mentioned characteristics could be summarized in a matrix form and can thus can be shown as in table:3.

TABLE III
SET OF SELECTED CHARACTERISRICS CHARACTERISTICS

Name	Type	Source
Depth of Inheritance Tree	Numerical	Chimdaber and Kramer [2]
Class Name Words	Binary	Antequil and Lethbridge [3]
Superclass Usage	Binary	
Interfaces Usage	Binary	
Number of Attributes	Numerical	Henderson-Sellers [4]
Number of Children	Numerical	Chimdaber and Kramer [2]
Number of Getters and Setters	Numerical	
Number of Methods	Numerical	Henderson-Sellers [4]
Number of Overridden Methods	Numerical	Henderson-Sellers [4]
Number of Static Attributes	Numerical	Henderson-Sellers [4]
Number of Static Methods	Numerical	Henderson-Sellers [4]
Total Lines of Code	Numerical	Henderson-Sellers [4]
Class Methods Mean Size	Numerical	Henderson-Sellers [4]
Weighted Method per Class	Numerical	Chimdaber and Kramer [2]
Mean Methods Complexity	Numerical	Chimdaber and Kramer [2]

As described in the paper "Revealing the Relationship between Architectural Elements and Source Code Characteristics" by Vanius Zapalowski, Ingrid Nunes and Daltro Jones Nunes.

IV. USE OF TECHNIQUES

In order to calculate the above mentioned characteristics I made the use of code given in the "complexityexample" example given in the module.

A. Description of Use

As a general overview of the code, it has following classes, ClassComplexityCounter.java, ClassFileOpener.java, IntfOvrdnMth.java, MainBytecodeAnalyser.java, MethodComplexityCounter.java, NumberOfChildre.java and OverridenMethods.java.

The MainBytecodeAnalyser.java is the class in which the "main" function is defined. To this class the bytecode which we want to analyze is passed as the arguments. The bytecode could be contained by just a simple .class file or a directory containing .class files or a jar file of any code. The MainBytecodeAnalyser.java in turn calls the ClassFileOpener.java to check whether the passed argument is a file, directory or a jar file. The ClassFileOpener.java contains the code for dealing with these different kinds of input.

The ClassComplexityCounter.java is the class which extends abstract class "ClassVisitor". This contains all the methods which are invoked on each .class file the program encountered. All these methods in this class are invoked only when some particular type of bytecode is encountered. For example the "visitMethod" of this class is only called when some method is reached in the byte code of the given .class file by the ClassReader.

The following are specific techniques which I used in order to calculate the matrix shown in table3.

Number of Attributes: I calculated this using a counter which gets incremented each time a "visitField" is invoked in the ClassComplexityCounter.java.

Number of Children: In order to calculate this I made an arrayList in which I pushed the object of NumberOfChildre.java class. This class has two fields which corresponds to className and superclass of the class visited. Then I traversed in the arrayList looking for java/lang/Object as a super class (as this would be the base class), after such entry is found I compared its className with the remaining elements of the arrayList and if this className is a super class for any other class then the counter is being incremented for this characteristic.

SuperClass usage and Interface Usage: For these two characteristics, the given type was binary but there was a problem in deriving as how to represent these characteristics in binary form. Therefore, there characteristics have been calculated as Boolean values, that is if the visited class is extending any class then the super class usage is set as 1 otherwise 0. And similarly for interface usage, if any visited class is implementing any interface then the interface usage value is set as 1 otherwise 0.

Number of methods: For calculation of this characteristic's value, whenever in a class a visitMethod method is invoked, then the counter for this gets incremented.

Number of getters and setters: To determine the values for number of getters and setters in class, I assumed the format of getters and setters in conventional coding standards, and then I stored all the fields of the class in an array and then saved all the methods in a different array. By doing this, I compared each element of the latter array with the concatenated value (first by get and then by set) of each element of the former array. If any check results positive, then the counter for number of getters and setters in a class is being incremented.

Number of overridden methods: In order to calculate this, I partitioned the problem into two parts. First, the number of overridden methods because of super class and second the number of overridden methods because of interfaces. Now to calculate the number of overridden methods because of super class usage, I made an arrayList of objects of class OverridenMethods.java. This class has three fields, each of which corresponds to a class name, method name and the super class of the visiting class. Now, as each class can have multiple methods, therefore more than one object of this type was created during a single class visit. If the className attribute for each element in the arrayList comes under superclass attribute for any element in the arrayList, then this method attribute for latter element is compared with the method attribute of former element then if they match then the counter was incremented.

Now for number of overridden methods because of interfaces, I first checked the access attribute for the visited class to be 1537, as this is the access code for the interfaces. If the interface is encountered using this access attribute, I counted the number of methods for this interface and saved this method count and interface name in an object of class IntfOvrdnMth.java. Then I pushed this object in an arrayList. After this, whenever a class is encountered (whose access value is 33) then I compared the interface name from the elements of the arrayList which I made earlier with the interface names in the array "Interfaces"(this array has automatically been generated with the visit method). If the names match, then for the particular class encountered, then the total count of overridden methods is incremented by the number of methods associated with this interface.

Number of Static Attributes: When the visitField method is invoked and its access attribute is equals to 8 then the counter for number of static attribute is incremented.

Number of Static Methods: When the visitMethod method is invoked and its access attribute is equals to 8 then the counter for number of static methods is incremented.

Class Methods mean size: For this, I overrided the methods of abstract class MethodVisitor which could be invoked due to the different types of instructions in the method. By doing this, I calculated the number of lines of code for each method and then I added all these values and divided it by the sum of number of methods and number of getters and setters.

Total Lines of code: For this, I also counted the lines where any field was declared and then added this value to the total number of lines of all the methods calculated for class method mean size characteristic.

Mean method complexity: For calculating the complexity of individual methods, I used the same complexity which was calculated in the example "Complexityexample" given in the module and divided it by the total number of methods.

Weighted method per class: For this I added the complexity of individual methods of a class.

Depth of Inheritance tree and Class name words: I couldn't derive values for these two characteristics because of the time constraint and limited knowledge for the derivation of these two characteristics.

B. Implementation

The implementation of code could be found in the separate folder. The output of test code looks like as shown below.

```
>> Interface Name: first/pettable
 > Class Name: first/MeTestOne
 > Class Name: first/Test
 > Class Name: first/MeTestSecond
##### Characteristics Matrix: #####
[1] No. of attributes: 6
[2] No. of getters and setters: 8
[3] No. of methods: 12
[4] Interface Usage: 1
[5] Superclass Usage: 1
[6] No. of static attributes: 2
[7] No. of static methods: 3
[8] Mean method Complexity: 1.0
[9] Class methods mean size: 4.0
[10] Total Lines Of Code: 104
[11] Weighted Methods per Class: 20.0
[12] No. of Overriden Methods: 1
[13] No. of Children: 1
-----------------------------------------------
>> Interface Name: second/shareable
 > Class Name: second/Me
 > Class Name: second/MeTestOne
 > Class Name: second/Test
##### Characteristics Matrix: #####
[1] No. of attributes: 5
[2] No. of getters and setters: 3
[3] No. of methods: 10
[4] Interface Usage: 1
[5] Superclass Usage: 1
[6] No. of static attributes: 1
[7] No. of static methods: 1
[8] Mean method Complexity: 1.0
[9] Class methods mean size: 4.0
[10] Total Lines Of Code: 62
[11] Weighted Methods per Class: 13.0
[12] No. of Overriden Methods: 2
[13] No. of Children: 1
```

V. RESULTS & ANALYSIS

All these characteristics are the standard code characteristics for software development. Therefore, these characteristics have been widely used for different application performance analysis and maintenance. These characteristics have further been used for the monitoring of the system depending on the values for these characteristics. The fluctuation of these values can trigger the programmer for working on more efficient and better object oriented approach. For example, if the value of weighted method per class is increasing then it means that the maintenance cost of the class of getting higher.

VI. REFERENCES

[1] "Revealing the Relationship between Architectural Elements and Source Code Characteristics" by Vanius Zapalowski, Ingrid Nunes and Daltro Jones Nunes.

[2] 2. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. Journal of Transactions on Software Engineering, 20(6):476–493, June 1994.

[3] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. Journal of 23 Software Maintenance: Research and Practice, 11(3):201–221, May 1999.

[4] B. Henderson-Sellers. Object-oriented metrics: measures of complexity. Prentice-Hall, 1 edition, 1995.