UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Metrics-based comparison across languages and paradigms

Bachelorarbeit
zur Erlangung des Grades
Bachelor of Science
im Studiengang Informatik

vorgelegt von

Johannes Klöckner

Erstgutachter: Prof. Dr. Ralf Lämmel
Institut für Informatik

Zweitgutachter: M. Sc. Martin Leinberger
Institut für Informatik

Koblenz, im März 2015

Erklärung

„Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Kurzfassung

Der Vergleich von Software-Code bildet einen wichtigen Bestandteil in der Software-Entwicklung, da er einen objektiven Vergleich von Programm-Code ermöglicht. Seit den Anfängen in den 1960er Jahren wurden Metriken entwickelt, die auf unterschiedliche Gruppen von Sprachen und Paradigmen angewandt werden konnten. Das 101companies project bietet eine Sammlung von Implementationen der gleichen Aufgabe in unterschiedlichen Sprachen. In dieser Arbeit sollen ausgewählte Metriken auf Grundlage dieses Systems verglichen werden, um Vor- und Nachteile sowie Probleme der Metriken zu bestimmen.

Abstract

The comparison of software code is an important area in software engineering, since it allows to compare the software code in an objective manner. Since the beginning of metrics in the 1960s, there exists a number of software metrics that could be applied to different groups of software languages and paradigms by performing this task in an automatic way. In this thesis, the 101companies project, as a software chrestomathy with implementations of the same feature in different languages, is used as basis for comparison of the metrics to detect their advantages and disadvantages.

Danksagung

Mein Dank gilt Herrn Prof. Lämmel für die Möglichkeit, die Arbeit zu verfassen. Außerdem danke ich ihm und Herrn Leinberger für die Unterstützung während der Bearbeitung und für die wertvollen Hinweise bei aufgekommenen Fragen.

Contents

1 Introduction

1.1 Motivation

In different domains in computer science situations arise where different concepts and the corresponding code should be compared. This could help improving some factors such as to increase the efficiency regarding the time and costs to enhance the maintainability or to analyse the reusability.

It is important to consider these factors since many projects exceed the estimated time and costs. In reference to a study (The Standish Chaos Report, 2004), only 29% of software projects fulfil their targeted objective including the expected functionality, the budget, and the time. There is also the value of $55 billion which is growing out of cancelled projects every year [9] Even if it is unlikely that none of the developed code parts of the cancelled software projects can be reused in other projects, there is a great potential of reducing the loss of capital input by better predictions of the deployed resources. Of course, there are also reasons for this problem, that will not be solved by metrics. Examples for applying metrics are quality assurance, performance, debugging (prediction of defects (post release and prior to release)), management, and cost prediction.

Among others, the complexity (e.g. McCabe metric), the volume (e.g. LOC) or the modularity could be taken as a criterion of classification. Especially for larger projects, it can be helpful to use metrics which can analyze software code across different languages and paradigms when the quality of a module has to be compared to another module with the same functionality but in another language. To define a metric that is applicable in as many cases (programming languages and paradigms and also subsets of them) as possible is the goal to achieve an automated evaluation of software code. Referred to that, the advantages and disadvantages of this process should be identified.

1.2 101companies

We can utilize the 101companies project[1] (or just '101') as a basis for comparison and testing, "as it provides a suitable chrestomathy, i.e., a collection of systems exercising different languages, technologies, and styles, while being comparable in terms of implemented features." [1] The contributions, which are small software systems in the context of 101, are related to different features which are applied to a structure that is intended to be intuitively and readily understandable and extensive enough for diverse features at the same time.

[1] `http://101companies.org/`

1.3 Related Work

The comparison of the metrics in this thesis is based on the paper [1] , that uses the metric NCLOC to compare implementations of features from the "101companies project" [2].

In this thesis, the comparison of the implementations should be expanded by further metrics. Examples for comparison of implementations with the aid of the Halstead metric are stated in [3]. The validation and comparison of metrics in general is described in [4] and [5]. The description of a procedure to evaluate a metric especially for object-oriented programming languages is stated in the paper [5]. This thesis applies some of the criteria for evaluating the implementations across different languages with the special consideration of language-independent metrics.

1.4 Background

The first attempts to measure the programming productivity and effort were in the mid-1960's by metrics based on Lines of Code. At this point in software engineering, the different activities of measurement, which result in numbers, were described by 'software metrics' as a collective term and should help "predict software resource requirements and software quality". [4]

In 1976, the first books explicitly on the topic of software metrics were published. The first application of software metrics for predicting the software quality was published by Akiyama in 1971 [6] when he developed a model which was based on the number of defects per KLOC (the module defect density). Behind these approaches, there was the idea that the size (that was measured by LOC) is directly depending on the quality and effort. [4]

A higher interest in metrics for complexity and functional size arose in the mid-1970's when the diversity of programming languages increased. In this context, measures which had the intention to be independent of the used language were developed by Halstead [7] and Albrecht [8] (function points). [4] Many of the metrics which followed later were particularly based on these metrics.

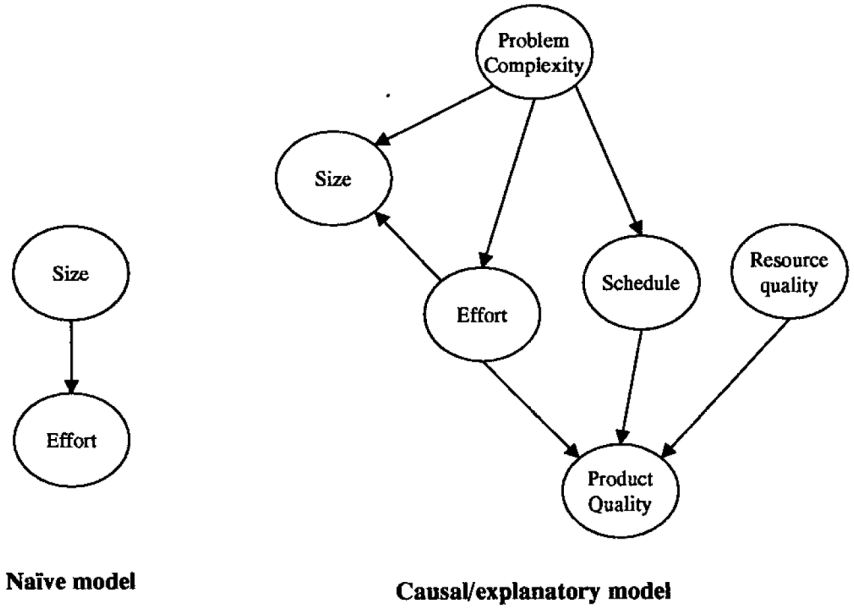

Figure 1: Comparison of two approaches to deriving a model [4]

2 General discussion

2.1 Evaluation of a metric

In this section the validation of a new metric is described with the aid of the Kaner and Bond's Evaluation Framework [10] that is based on the IEEE software metric criteria). [11]

Referring to Kaner and Bond, some points have to be checked within the framework [10]:

1. "What is the purpose of the measure?"

   It has to be considered who intends to utilize the metric and how much should be invested to ensure its validity.

2. "What is the scope of the measure?"

   Does the metric affect only one team or even multiple parts of a large project or company, which would probably mean a higher number of confounding variables concerning the system?

3. "What attribute of the software is being measured?"

   The idea of the measurement has to be defined clearly.

4. "What is the natural scale of the attribute we are trying to measure?"

A suitable type of scale must be chosen for different types of measures like program length or programmer skill.

5. "What is the natural variability of the attribute?"

   How big are the expected differences of the values resulting from the metric, that can be ignored, because they are too small to lead to another statement?

6. "What measuring instrument do we use to perform the measurement?"

   For example, 'counting', 'comparing', or 'timing' can be used as instruments to obtain the values.

7. "What is the natural scale for this metric?"

   In some cases, the scales of the metric and the attribute can differ, particularly if the metric can not be applied automatically, but evaluated by an expert, for example in case of comparing and ranking the testing efforts ("thoroughness of testing").

8. "What is the natural variability of readings from this instrument?"

   In which case does the metric give a wrong result concerning the original purposes?

9. "What is the relationship of the attribute to the metric value?"

   Is the metric measuring the attribute (the original purpose)?

10. "What are the natural and foreseeable side effects of using this instrument?"

    What impacts on the attribute will arise, if the developers change their behavior for improving the measured result? Is it possible that the underlying attribute is getting worse in that case?

2.2 Limitations / Problems

Among some developers, there exists a "strong tendence to display over-optimism and over-confidence" [9], which means that "simplistic measurements may cause more harm than good," if "data [...] is shown because it's easy to gather and display". [12] But positive effects of software metrics on "the developers' productivity and well being" [11] are mentioned as arguments to contradict the statement above.

The problem arises from deriving a model from a statistical connection without the consideration of causal relationship because it's possible that not all the data, which is necessary to make a decision, has been collected (see Figure 1) with the result that the "regression models often lead to misunderstanding about cause and effect." [4] Since these "models lack any causal

structure", they "cannot be used effectively to provide decision support for risk assessment."

2.3 Different points of view and the corresponding metrics

Management, Administration:

Software managers and administrators could ask questions as the following which relates to a cost model [4]:

- "For a specification of this complexity, and given these limited resources, how likely am I to achieve a product of suitable quality?"
- "How much can I scale down the resources if I am prepared to put up with a product of specified lesser quality?"
- "The model predicts that I need 4 people over 2 years to build a system of this kind of size. But I only have funding for 3 people over one year. If I cannot sacrifice quality, how good does the staff has to be to build the systems with the limited resources?"

Although the described metrics in this thesis aren't able to give a direct answer to the questions, they can help to develop a model that is influenced by the metrics.

Developer:

Relating to the problems occurring by the subdivision of modules during the testing phase, the developer can make a better decision about whether to divide a module in an early state of the development process by estimating the increase in complexity with the help of a metric.

Another problem concerns the focus of the developers on the components and their interaction in the testing phase. A metric could help to quantify the integration need of an individual component, so that a developer can distinguish the importance of testing how the components work together. (cyclomatic complexity) [13]

3 Metrics

The following metrics are chosen for comparison because they are relevant in evaluating the complexity or effort. [14] [15]

3.1 LOC

As described in the motivation for metrics, the manager of software projects had "the assumption that product 'size' measures should drive any predictive models." [4] Lines of Code (LOC or KLOC for thousands of lines of code) was the first key metric that was used to measure the program volume. Furthermore, 'LOC per programmer month' "was, and still is, used routinely as the basis for measuring programmer productivity" so that it "was assumed to be a key driver for the effort and cost of developing software." [4] For example, Putnam and Boehm used LOC (or similar metrics) in early resource prediction models as a key factor in a formula like 'Effort=f(LOC)'.

In the late 1960's, indirect measurements of LOC like 'defects per KLOC) should predict the program quality. [4]

Possible problems in usage of the metric are given in the code examples (Listing 1 and 2). These two examples consist of the same number of lines of code. Although the two methods are completely different concerning the complexity and effort, the values for both examples are equal.

```
1 void print() {
2     System.out.println();
3     System.out.println();
4     System.out.println();
5     System.out.println();
6     System.out.println();
7     System.out.println();
8     System.out.println();
9 }
```

Listing 1: "LOC 1"

```
1 static int factorial(int n) {
2     if (n > 1) {
3         return n * factorial(n-1);
4     } else if (n == 1 || n == 0) {
5         return 1;
6     } else {
7         throw new RuntimeException("No valid value");
8     }
9 }
```

Listing 2: "LOC 2"

3.2 Cyclomatic complexity / McCabe metric

Along with three other software metrics, that "were defined by Halstead, Albrecht, and DeMarco"[13], cyclomatic complexity, which was defined by McCabe, was one of "four basic theories" that "have been the source of the majority of the research conducted on software metrics". [13]

By the McCabe metric, it is possible to measure the number of paths through a program. [13]

Because of the possibility of an infinite number of paths caused by a loop ("if the program has a backward branch"), the metric "is built on the number of basis paths through the program." [13]

By use of graph theory, the Cyclomatic complexity v(G) gets its value in deriving from a flowgraph. A more practical approach is to determine the number of decision statements in a program which leads to the formula [13]:

$$\text{v(G)} = \text{number of decision statements} + 1$$

If the decision statements (which possibly contain compound conditions) were always counted as 1, there could be a different number for a program with the same semantic but a different syntax . To avoid this case, cyclomatic complexity splits every decision statement into several parts that are counted separately. If cyclomatic complexity didn't recognize these differences in program complexity because of compound predicates, the values for `"IF A=B AND C=D THEN"` (1 `IF` statement + 1 $\rightarrow$ **2**) and `"IF A=B and IF C=D THEN"` ($\rightarrow$ **3**) wouldn't be equal. McCabe proposed in his paper [13] an upper limit of 10 for program complexity because the programs would be less manageable and testable in case of greater complexity.

Visualization

In order to visualize the complexity of a program, we could construct a flowgraph with nodes and arcs. The nodes represent one part of the code with a sequential flow (of the instructions) and the arcs correlate with the branches in the program. After we have determined the number of vertices (n), edges (e), and connected components (p), the value for cyclomatic complexity can be calculated by $v(E) = e - n + p$.

In the example the value would be 5:

$$\begin{aligned} v(E) &= 15 - 11 + 1 \\ &= 5 \end{aligned}$$

$$\text{for } e = 15, n = 11, \text{ and } p = 1$$

The value corresponds to the "maximum number of linearly independent paths through the program" , which is originated by the mathematical properties. [13]

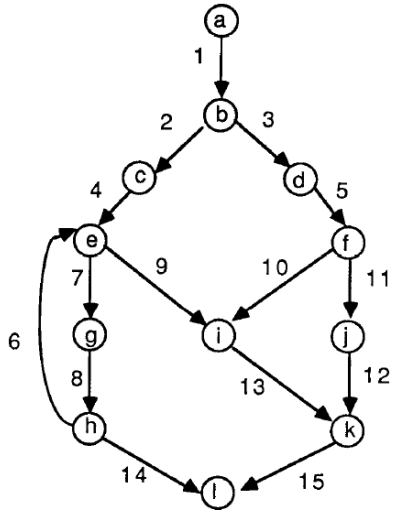

Figure 2: Example of a flowgraph (cyclomatic complexity)[13]

Another form of visualization of the structure can be realized by a design tree and a design subtree. The design tree is based on the hierarchical structure of the program. A design subtree represents one subset of the whole tree with the modules which are executed with given data input. The connection of modules within the design tree can be transformed into a flowgraph. Sometimes it is necessary to reduce the module's flowgraph. If there is a path that "does not influence the interrelationship between design modules", the reduction can be performed with the aid of reduction rules. The resulting cyclomatic complexity *iv(G)* of the reduced graph G is named **module design complexity**. To calculate the design complexity, the module design complexity of any component has to be summed up [13]:

$$S_0 = \sum_{i \in D} iv(G_i)$$

("where D is the set of descendants of M unioned with M" [13])

Even if none of the studies were able to give an absolute threshold for quality software, there are studies, that found a significance of the value of complexity with these results:

- An improvement of the program results in a reduced value of cyclomatic complexity.
- Another study has compared the average error rate of two groups of modules with different values for cyclomatic complexity. The group with a value of 10 or less had an error rate of 4.6 while the error rate of the other group (with a value of more than 10) was 5.6 . (The groups in the analysed system with 276 modules had approximately the same size.)
- A third study describes a connection of the value of cyclomatic complexity and the performance of programmers on comprehension, modification and debugging tasks.
- "The correlation between cyclomatic complexity and the number of errors was above 0.90". (The data basis was given by "the empirical error data collected on the UNIX operating system.")

[13]

3.3 Chidamber and Kemerer

There is a number of object-oriented design metrics (introduced by Chidamber and Kemerer in [16]) which are discussed in a study [5] "as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators".

Properties of the study

- Data collection of eight medium-sized information management systems/projects
- Identical requirements for the resulting systems
- Programming language: C++
- Development: sequential life cycle model

Testing of the metrics

The metrics were tested in a study with a software system developed by students on the basis of the sequential software engineering life-cycle model. To ensure that the idea and requirements would be comprehended clearly by the students, a software project should be developed from a domain, which is not too complex for students without a detailed briefing. Therefore a management information tool was taken in the video rental business that analyses customers and video databases. During the development process, the detected faults were documented

and later associated with the final classes at the end of the implementation phase. Additionally, a team of experienced developers reviewed the classes in the testing phase after the complete implementation of the system.

Proposed attributes of the metric:

1. Can predict class fault-proneness in an early state of the software development
2. Beat the "traditional" code metrics in terms of earlier predictions during the life cycle

Limitations in using the metrics

Since the discussed metrics depend on the used language , they had to be adjusted in certain points to apply them .

List of the metrics of Chidamber and Kemerer, which are often applied together in software projects:

3.3.1 Weighted Methods per Class (WMC)

To calculate the value of WMC for a class, the complexity of each method of this class has to be summed up. In order that the metric wouldn't be limited regarding the applicability, the metric for calculating the complexity of the method is not defined.

3.3.2 Depth of Inheritance Tree of a class (DIT)

The dependencies regarding the inheritance of a class can be described in a graph. Depending on the used language and the analysed program, the graph can either be a tree or another graph structure. The maximum depth of this graph defines the value of DIT.

3.3.3 Number Of Children of a Class (NOC)

This metric is similar to DIT, but it regards only one level below the class in the graph, so the directly inherited classes are counted.

3.3.4 Coupling Between Object classes (CBO)

The metric CBO is used to count the classes that are coupled with the analysed class. A class is described as coupled to another if it uses the member functions of the other class.

3.3.5 Response For a Class (RFC)

To look at the impact of an object of the analysed class, the metric RFC counts the maximum number of methods that can be executed by invoking them directly by this object.

3.3.6 Lack of Cohesion on Methods (LCOM)

In this case, the cohesion is measured by counting the functions with shared variables. The resulting value of a class is the number of pairs of functions (of this class) without shared variables minus the number of pairs of functions with shared variables. To ensure these negative values do not appear in this metric, the value is set to zero even if there are only functions with shared variables.

3.3.7 Critical discussion of the metrics

By reason of the detected flat inheritance hierarchy (DIT) and the low number of children of a class (NOC), these metrics couldn't give a significant relation to the fault-proneness. [5]

3.4 Halstead

The Halstead metric [7] is a complexity metric that distinguishes between two classifiers: "**operators**" and "**operands**". A keyword that relates to an algorithmic action is counted as an operator. An operand is defined as a keyword that represents data. The basis for the metric is given by the following numbers:

$$n_1 = \text{number of unique operators}$$
$$n_2 = \text{number of unique operands}$$
$$N_1 = \text{total occurrences of operators}$$
$$N_2 = \text{total occurrences of operands}$$

The sum of all unique tokens is defined as the vocabulary of the program:

$$n = n_1 + n_2$$

The program length is calculated by the total number of tokens:

$$N = N_1 + N_2$$

The following measures are based on n and N:

- Volume: $V = N \times log_2 n$
- Difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
- Effort: $E = D \times V$

To discuss the expressiveness of the Halstead metric, two code examples (Listing 3 and 4) are given with the following values:

- Example 3
  - Difficulty : 2.37
  - Effort : 269
- Example 4
  - Difficulty : 1.16
  - Effort : 39

The two code examples provide the same functionality but very different values. While the invocations of the methods are nested in code example 4, new variables were declared in example 3 to store the values without further usage. Although the metric doesn't declare to measure the functionality, so that it is unlikely for the values to be equal, it can be argued that the difficulty of example 3 is not twice as big as the one of example 4 because the difficulty of a nested structure could be higher than stated.

```
1  int adjustSalary(int currentSalary) {
2      int salary1 = currentSalary;
3      int newSalary1 = doAdjustment1(salary1);
4
5      int salary2 = newSalary1;
6      int newSalary2 = doAdjustment2(salary2);
7
8      int salary3 = newSalary2;
9      int newSalary3 = doAdjustment3(salary3);
10
11     return newSalary3;
12 }
```

Listing 3: Example of Halstead metric

```
1 int adjustSalary(int currentSalary) {
2     return doAdjustment3(doAdjustment2(doAdjustment1(
          currentSalary)));
3 }
```

Listing 4: Example of Halstead metric

3.5 Henry and Kafura

The metric of Henry and Kafura [17] is related to the information flow of a software module. The input and output together with the use of global data structures is measured to evaluate the complexity of the modules. The following two components are defined:

fan-in of procedure A:

(number of local flows into procedure A) + (number of data structures as an input for A)

fan-out of procedure A:

(number of local flows from procedure A) + (number of data structures that were updated)

With factoring in the length into the formula (for procedure A), the complexity is calculated by

$$\text{complexity}(A) = \text{length}(A) * (\text{fan-in}(A) + \text{fan-out}(A))^2.$$

A problem in applying this metric is shown within Listing 5, where the variables of one method are passed to another method without changing the values. For this module, the value takes all three variables into account so that the value is higher than what would be expected.

```
1  public class Henry_Kafura {
2
3      public static void passVariables(int a, int b, int c
           ) {
4          System.out.println();
5          doSomething(a, b, c);
6          System.out.println();
7      }
8
9      public static void doSomething(int a, int b, int c)
           {
10         System.out.println();
11     }
12
13 }
```

Listing 5: Example of Henry & Kafura

4 Comparison of metrics

4.1 Criteria for comparison of metrics

1. Applicability to as many languages and paradigms as possible.

   If necessary: limitations on certain groups.

2. Consideration of the utilization of (un)typed languages and the use of certain kinds of input, i.e. API, external data structures.

3. Applicability regarding the (different) size of projects and program code:

   Applicable to different sizes of projects?

4. Type of metric:

   Which attributes does the metric include, i.e. which statement concerning the types of metrics can be made?

   Does the metric really measures the intended attributes? (In which cases?)

4.2 Application of the criteria to the metrics

4.2.1 LOC / NCLOC

Referred to criterion 1 and 3, no limitations are necessary (it can be used for every text document), but there is almost no consideration of syntax or semantic. Only alterations of LOC (e.g. NCLOC) regard parts of the text file (such as comments). The disadvantage of this metric is that it takes no typing or external data structures into account.

4.2.2 Halstead

Regarding criterion 1, the metric has a wide-ranging applicability as it can be used for every language. The only prerequisite is to define the operators and operands for the specific language.

The number of operators and operands are used to calculate the volume, difficulty, effort and necessary time. (criterion 4) In this thesis, especially the difficulty and the effort are relevant because they have a direct relation to the complexity.

4.2.3 Henry & Kafura

The metric is applicable (criterion 1) to all procedures with detectable input and output, so the metric can be used for a wide range of languages. It has no regard of typing as the parts of the code with no relation to input or output are only involved by the length of the code (referred to LOC, section

3.1). However, the use of external data such as files has an effect on the value because the metric also counts global data structures. (criterion 2) Regarding the size (criterion 3), no limitations are necessary, since it only takes the scope of one class (or method) into account.

The feature (4) of the metric is the measurement of the effort for implementation, testing, and maintenance.

4.2.4 Metrics of Chidamber and Kemerer

Since the metrics of Chidamber and Kemerer were developed for OO programming languages (criterion 1), it can only be applied to a limited group of languages that uses a structure with classes. For criterion 2 CBO and RFC are helpful. If a class uses an API, it is coupled to one part of the API , which is reflected in the metric 'CBO'. Additionally, 'RFC' is also beneficial at this point, since it counts the directly invoked methods so that the invoked methods of the API can be represented in the value of the metric. As it depends on a group of classes (criterion 3), it can't determine the value of a single class like LOC or McCabe to compare it to another class, but it's still expressive for little projects even if it should be applied to bigger projects for results that are more precise. [5] The different types (criterion 4) of complexity (as described in 3.3) give a prediction of the fault-proneness. [16]

5 Evaluation of metrics (101 context)

To compare the different metrics, the following bar charts are taken into account to show the application of metrics on the basis of the 101repo. Together with the description of the values, references to the criteria in section 4 are given. If a value for a metric is omitted within the diagram, it is either for the reason of missing applicability to this contribution or a better overview for comparison.

5.1 NCLOC

Except for jdom, the java contributions have almost equivalent values. The slightly lower value of 'javaStatic' comes from the different structure (the functions 'total' and 'cut' are not placed into the classes of the 'model'-folder (Company, Department, Employee)), which shows that this metric does not consider enough the different structures within the implementations. (crit. 4). In the pyjson contribution, the input is given by a json-file, that will be scanned for the relevant keywords such as 'salary' or 'departments'. This example shows that the usage of an external data structure (pyjson) or the DOM structure (jdom) itself doesn't influence the value. But, it can be argued that the detection of the keywords has a lower complexity for the class. (criterion 2)

5.2 McCabe

In contrast to LOC (NCLOC) in section 5.1, the value for pyjson is relatively high, which relates to the 'if' conditions to detect the keywords (employees, departments, manager). This case shows that the metrics result in completely different values, even though they are both intended to indicate the complexity. (criterion 1) Additionally, it is shown that the complexity that relates to the data input indirectly results in a relatively high value even if the program itself is relatively small. (criterion 3)

In the jdom contribution, the keyword is restricted to 'salary' with the result that the lower complexity leads to a low value, too. (criterion 1)

5.3 Henry & Kafura

The values of pyjson and jdom are similar, but the additional variables of pyjson to manage the files increase the value. Since only the direct input through the parameters to the methods is counted, the DOM structure can't be taken into account, which results in the extremly low value (the low quantity of classes with small functions and few parameters). (crit. 2) Since the other java-contributions contain more classes and methods, these contributions amount to notably high values, although the differences in the code are smaller.

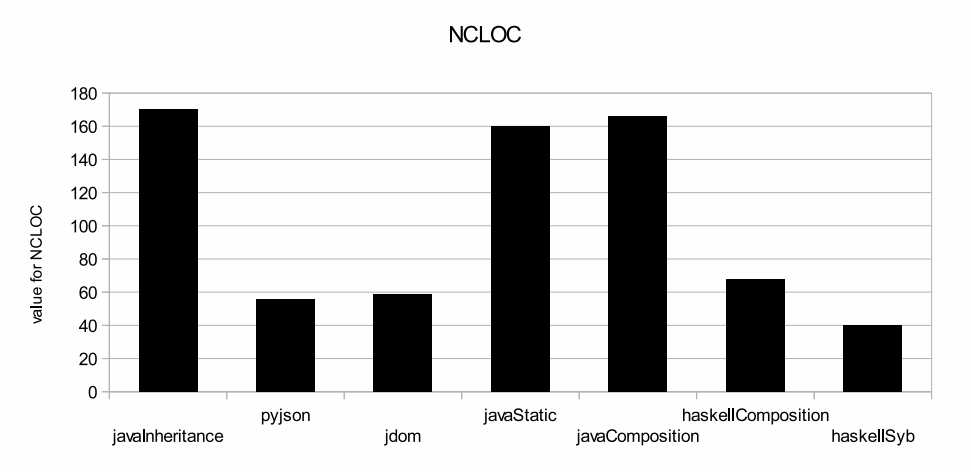

Figure 3: Values of NCLOC for 101 contributions

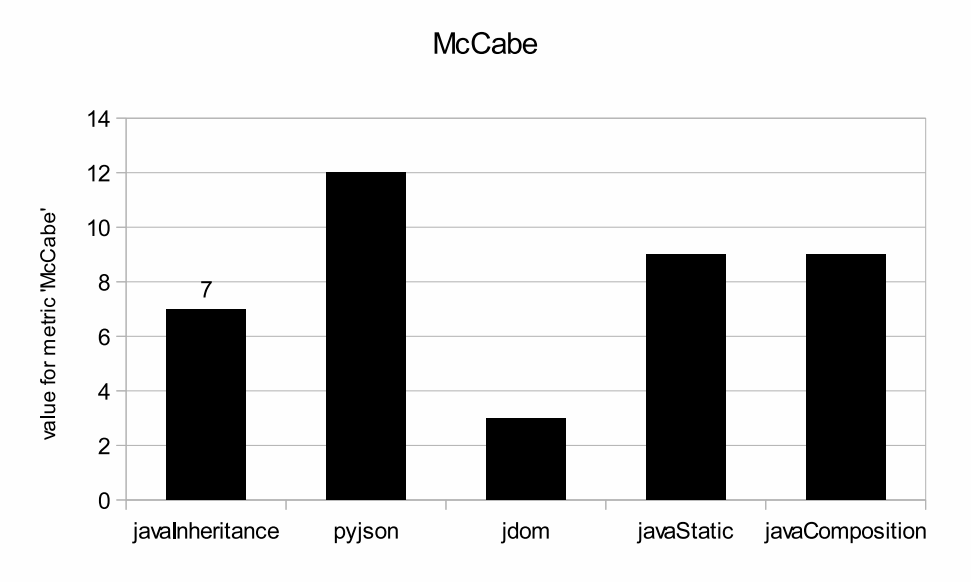

Figure 4: Results of applying the McCabe metric to 101 contributions

The measurement was performed while ignoring comments, brackets, and package declarations.

5.4 Weighted Methods per Class (WMC)

This metric is based on the average of each class, which means that the value for the pyjson contribution is outstandingly high due to the low number of classes (with a similar complexity). The other values are almost equally big because the code is spreaded among more classes. This case with a high variance also shows the problem for criterion 3 because the metric can't produce expressive values for these small contributions which could lead to a misunderstanding.

5.5 Coupling Between Object classes (CBO)

The noticeable high value of the javaStatic contribution can be explained by the static functions (to evaluate 'Company', 'Department', 'Employee') within the classes 'Cut' and 'Total', which is a reasonable relation to the intended attribute (criterion 4), since the methods are more distributed across the classes. Referred to criterion 2, the relatively high value of jdom seems to correlate with the intended attribute. For the pyjson contribution, the problem of the external file arises with this criterion.

5.6 Response For a Class (RFC)

Since the pyjson and jdom contributions are connected with an external structure (criterion 2), they don't use a higher number of methods as in the javaComposition contribution, which leads to the low values.

5.7 Halstead metric

The high value of the difficulty of the haskellComposition contribution is plausible relating to the compact structure and can be explained by the relatively high number of unique operators (see figure 12). The javaInheritance and javaStatic contributions have almost equal value, which correlates to their similar usage of names for variables.

The two values of the javaInheritance and the javaStatic contributions could be reasoned by the high number of methods and variables. Reasonable at this point is also the lower value of the haskellComposition contribution that relates to the mentioned compact structure. In both diagrams, the jdom contribution shows the problem in criterion 2.

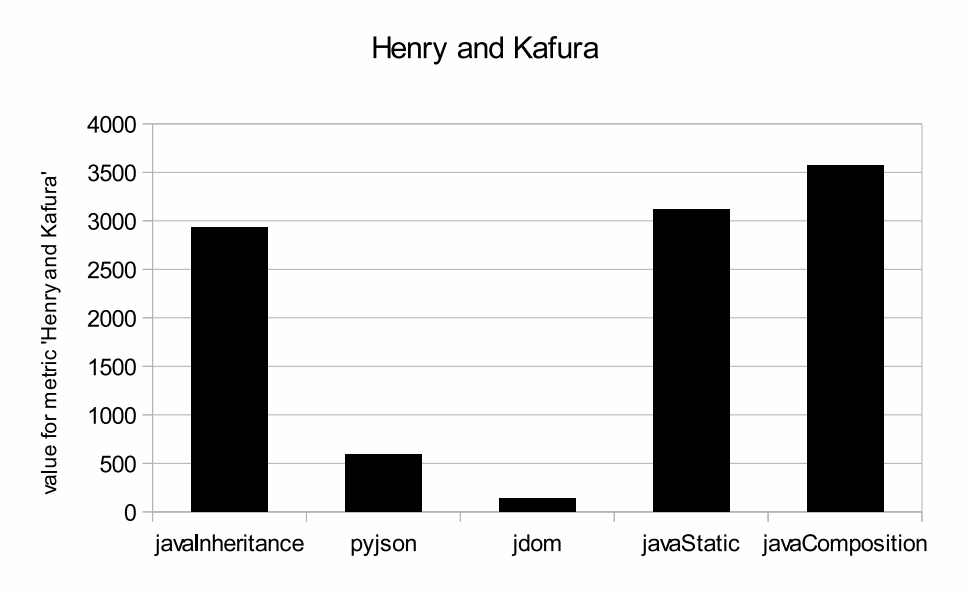

Figure 5: Results of applying the metric of Henry & Kafura to 101 contributions

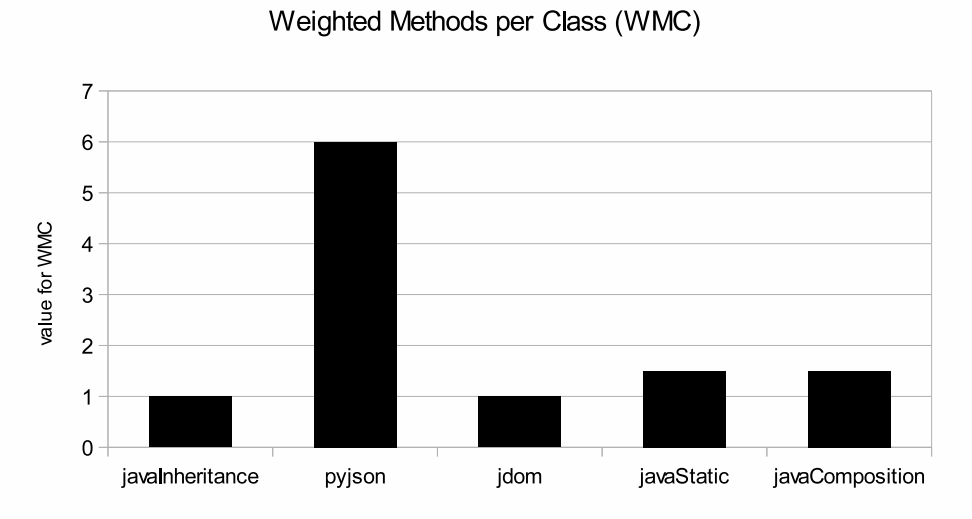

Figure 6: Results of applying the metric WMC to 101 contributions

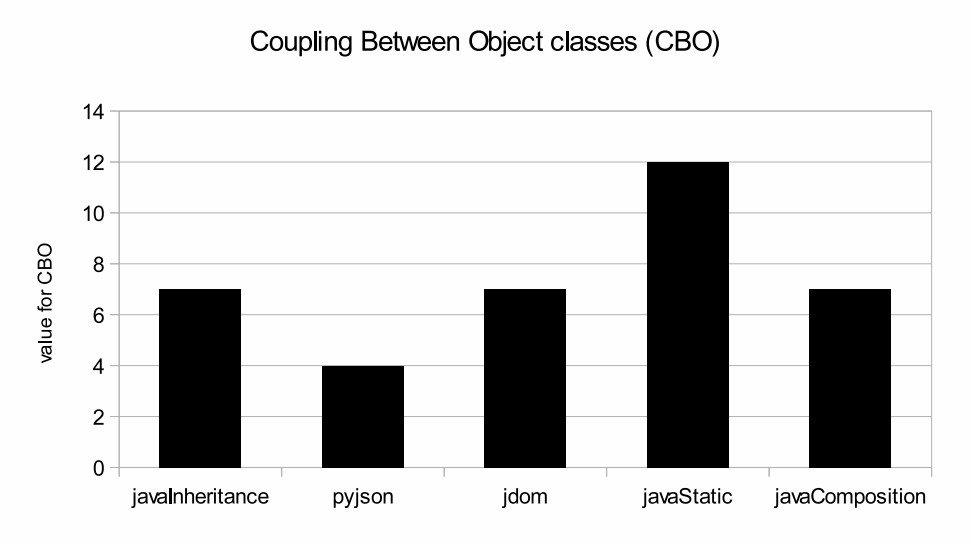

Figure 7: Results of applying the metric CBO to 101 contributions

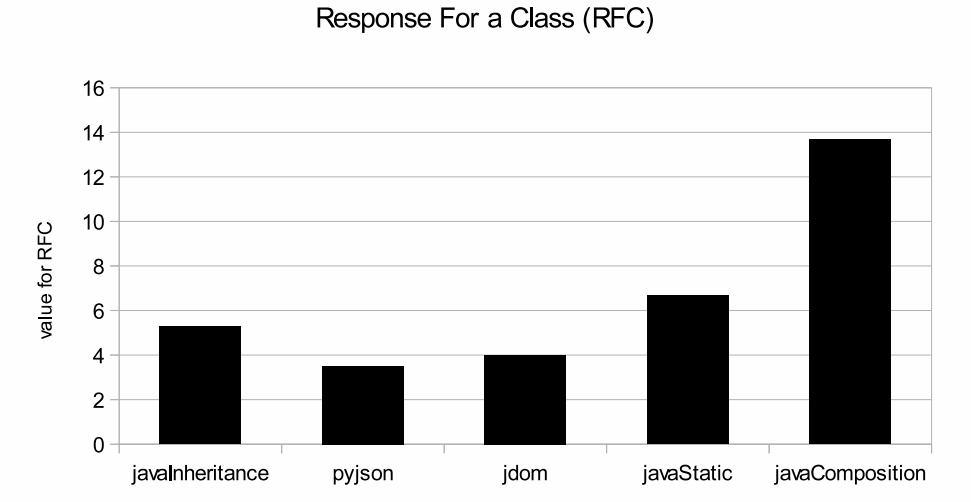

Figure 8: Results of applying the metric RFC to 101 contributions

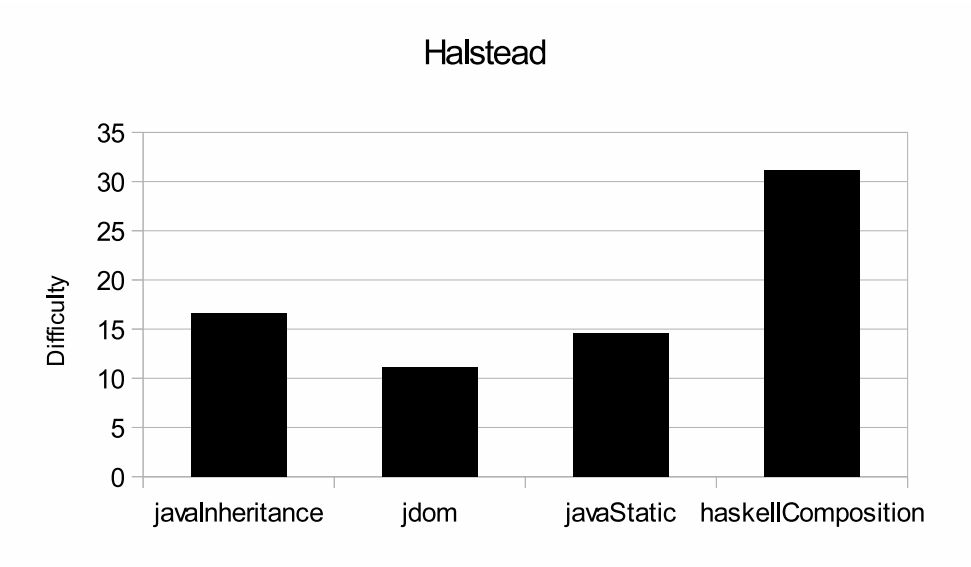

Figure 9: Halstead metric - Value of difficulty of 101 contributions

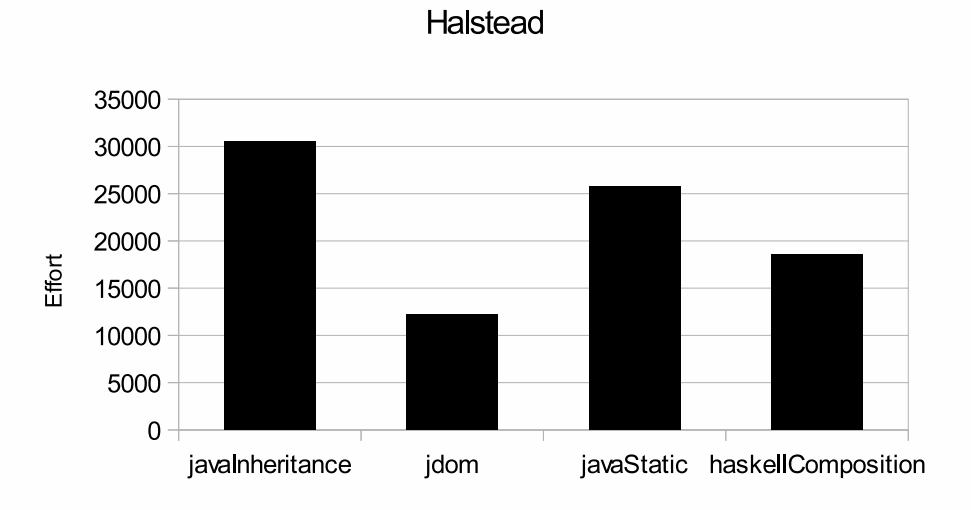

Figure 10: Halstead metric - Value of effort of 101 contributions

5.8 Groups of metrics

Based on the comparison of section 4.2 and the evaluation in the context of 101 in section 5 the following table summarizes the results. The columns represent the criteria stated in section 4.

criterion	general applicability (1)	typing/ext. data (2)	size (3)
LOC/NCLOC	✓	x	✓
McCabe	○	x	○
Halstead	✓	x	○
Henry Kafura	○	x	○
Chidamber & Kemerer	○	○	○

Table 1: Comparison of the metrics based on the criteria of section 4.2

✓= completely applicable, ○= restricted, x = no applicability

5.9 Problems

In addition to the problems that are described in section 4 there are also problems in using external data or modules.

When a metric is used in a software system with any kind of data input or API usage, it depends on the metric if all external data or program code are taken into account.

An example for this case is given in code example (Listing 6), where an external DOM file is used to specify the object structure. Instead of implementing the classes like in the 'javaComposition' contribution, the object structure is not directly covered by the code.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <xs:schema xmlns="http://www.company.softlang.org/model
       .xsd" xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified" targetNamespace="http
       ://www.company.softlang.org/model.xsd">
3
4      <xs:element name="company">
5          <xs:complexType>
6              <xs:sequence>
7                  <xs:element ref="name"/>
8                  <xs:element maxOccurs="unbounded" minOccurs
                       ="0" ref="department"/>
9              </xs:sequence>
10         </xs:complexType>
11     </xs:element>
```

```
12
13      <xs:element name="department">
14          <xs:complexType>
15              <xs:sequence>
16                  <xs:element ref="name"/>
17                  <xs:element name="manager" type="employee"
                        />
18                  <xs:element maxOccurs="unbounded" minOccurs
                        ="0" ref="department"/>
19                  <xs:element maxOccurs="unbounded" minOccurs
                        ="0" name="employee" type="employee"/>
20              </xs:sequence>
21          </xs:complexType>
22      </xs:element>
23
24      <xs:complexType name="employee">
25          <xs:sequence>
26              <xs:element ref="name"/>
27              <xs:element ref="address"/>
28              <xs:element ref="salary"/>
29          </xs:sequence>
30      </xs:complexType>
31
32      <xs:element name="name" type="xs:string"/>
33      <xs:element name="address" type="xs:string"/>
34      <xs:element name="salary" type="xs:double"/>
35  </xs:schema>
```

Listing 6: JDOM from contribution 'jdom'

Some metrics are more useful to regard these external structures than others. The metric of Henry & Kafura depends on the commonly used parameters or data structures, which means that external usage would affect the value of the metric. However, the API or data input itself doesn't influence the value. In this context, two of the metrics of Chidamber & Kemerer, CBO and RFC, are useful. If a class uses an API, it is coupled to one module of the API , which is reflected in the metric 'CBO' (section 3.3.4).

Another problem at this point is the missing consideration of (un)typed languages for all metrics.

6 Conclusion

6.1 Summary

Before applying the metrics, the developer has to decide which attribute of the software project should be compared and which paradigms are used. Depending on the purpose of the measurement, one metric or a group of connected metrics can be applied.

In the process of discussing the choice of the metric that will be used for the software project, he must keep in mind that there is no perfect metric for every case in the software development.

The metric LOC can be used as basis, especially in a project with different languages when the developer takes regard of the restrictions mentioned in section 3.1. When single modules in object-oriented programming languages should be compared, the metrics of McCabe and Henry & Kafura can be helpful in an early state since they do not depend on other modules of the projects. However, if the usage of external modules like API should be considered, these metrics could give misleading results. Instead, the metric suite of Chidamber & Kemerer can give better results for a collection of modules. If different paradigms are used for the implementation, the Halstead metric can be applied. In summary, metrics can help in predicting different attributes when the used metrics are chosen carefully by considering the possible problems.

6.2 Future Work

The metric comparison in this thesis includes a selection of metrics, that could be extended in a future work. Additionally, the code basis for comparing could be expanded by more implementations or bigger projects, so that certain aspects of the metrics can be shown more clearly. Another option at this point is the inclusion of more techniques regarding external modules and typing.

References

[1] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Comparison of Feature Implementations across Languages, Technologies, and Styles. In *Proc. of IEEE CSMR-WCRE 2014*. IEEE, 2014. 5 pages.

[2] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 101companies: a community project on software technologies and software languages. In *Objects, Models, Components, Patterns*, pages 58–74. Springer, 2012.

[3] Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with liteq. In *The Semantic Web–ISWC 2014*, pages 212–227. Springer, 2014.

[4] Norman E. Fenton and Martin Neil. Software metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM. URL: `http://doi.acm.org/10.1145/336512.336588`, `doi:10.1145/336512.336588`.

[5] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.

[6] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[7] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[8] Allan J Albrecht. Measuring application development productivity. In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, volume 10, pages 83–92, 1979.

[9] Carolyn Mair and Martin Shepperd. Human judgement and software metrics: vision for the future. In *Proceedings of the 2nd international workshop on emerging trends in software metrics*, pages 81–84. ACM, 2011.

[10] Cem Kaner and Walter P Bond. Software engineering metrics: What do they measure and how do we know? *methodology*, 8:6, 2004.

[11] Alex Boughton. Software metrics. accessed 03.11.2014. URL: `http://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/boughtonalexandra.pdf`.

[12] Andrew Binstock. Integration watch: Using metrics effectively. accessed 05.02.2015. URL: http://www.sdtimes.com/link/34157.

[13] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, December 1989. URL: http://doi.acm.org/10.1145/76380.76382, doi:10.1145/76380.76382.

[14] Alan J Perlis, Frederick Sayward, and Mary Shaw. *Software metrics: an analysis and evaluation*, volume 5. MIT Press, 1981.

[15] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[16] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, Jun 1994. doi:10.1109/32.295895.

[17] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, (5):510–518, 1981.

A Appendix

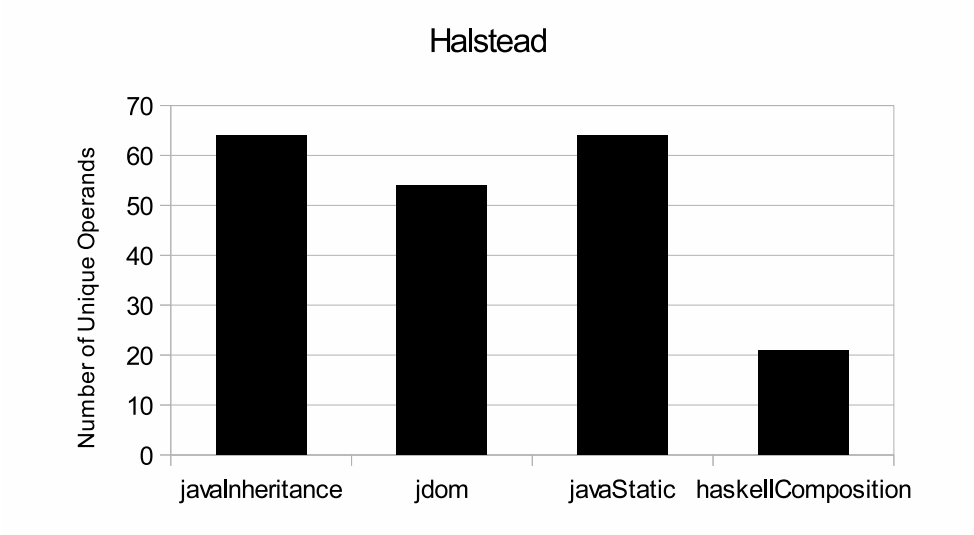

Figure 11: Halstead metric applied to 101 contributions - Number of unique operands

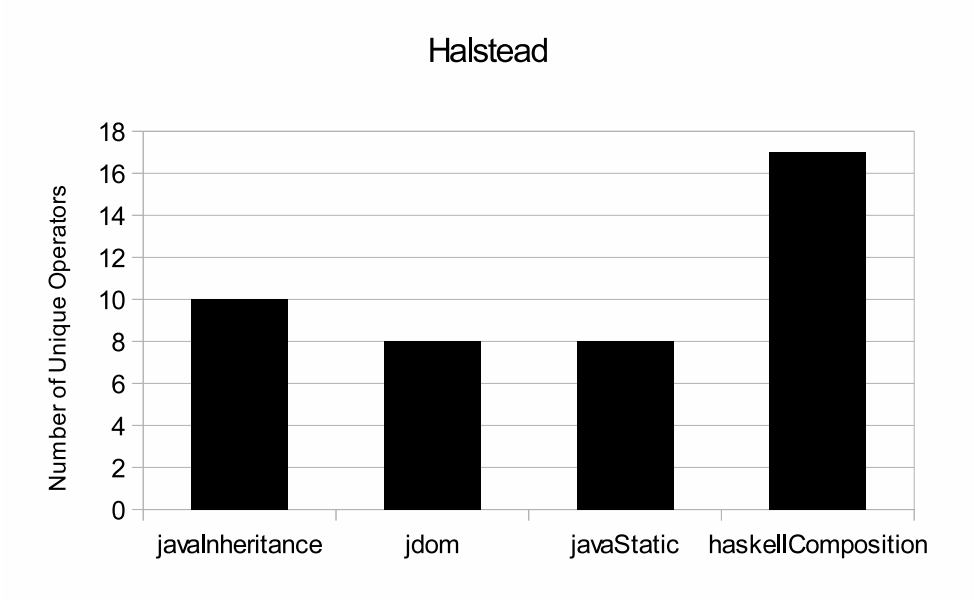

Figure 12: Halstead metric applied to 101 contributions - Number of unique operators

List of Figures

List of Tables

Listings