Program Complexity Metrics and Programmer Opinions

Bernhard Katzmarski, Rainer Koschke
Fachbereich Mathematik und Informatik
University of Bremen
Bremen, Germany
{bkatzm,koschke}@tzi.de

***Abstract*—Various program complexity measures have been proposed to assess maintainability. Only relatively few empirical studies have been conducted to back up these assessments through empirical evidence. Researchers have mostly conducted controlled experiments or correlated metrics with indirect maintainability indicators such as defects or change frequency.**

This paper uses a different approach. We investigate whether metrics agree with complexity as perceived by programmers. We show that, first, programmers' opinions are quite similar and, second, only few metrics and in only few cases reproduce complexity rankings similar to human raters. Data-flow metrics seem to better match the viewpoint of programmers than control-flow metrics, but even they are only loosely correlated. Moreover we show that a foolish metric has similar or sometimes even better correlation than other evaluated metrics, which raises the question how meaningful the other metrics really are.

In addition to these results, we introduce an approach and associated statistical measures for such multi-rater investigations. Our approach can be used as a model for similar studies.

***Index Terms* — control-flow metrics, data-flow metrics, program complexity**

I. Introduction

Halstead [1] as well as McCabe [2] were among the first researchers claiming that software metrics can be used to assess program complexity. Since then many different metrics have been proposed to measure complexity. Only few of these proposals have been evaluated empirically, mostly, by correlating metrics with indirect measures of complexity such as defects or change frequency or with other metrics believed to predict complexity, or by way of controlled experiments or field observations.

The problem of indirect measures such as defects is that causality is difficult to assure because there may be other, confounding variables influencing the results, such as the effort spent on testing, time pressure, or programming skills. Controlled experiments offer better control of influencing variables, yet they are relatively expensive and for this reason often small, which limits their validity.

In this paper, we explore the idea of using programmer assessments of code complexity gathered through questionnaires. Questionnaires were first introduced by Francis Galton (1822 – 1911), who is considered father of behavioral statistics. It is said he visited a livestock fair in 1906 where an ox was on display, whose weight was to be estimated by the visitors. None of the nearly 800 participants got the correct answer, yet the mean of all estimates was accurate to a fraction of one percent. This anecdote has lead to the notion of *wisdom of crowds* and *crowd sourcing*.

That is not to say that we claim the so-called *wisdom of crowds* emerges also when it comes to assessing program complexity. Rather we view questionnaires as a research instrument complementary to other instruments. While others have focused mostly on other research instruments, we would like to explore whether and how questionnaires can be used. Furthermore, subjective assessment by questionnaires is at least useful to generate operational hypotheses, which can later be assessed by controlled experiments.

Contributions. Our new contributions are two-fold. First, we investigate the question whether control and data-flow metrics can be used to assess program complexity as gathered from developer opinions. Second, our research approach can be used as a model for investigations based on questionnaires. We discuss and show how methods and statistics from behavioral sciences may be adapted to program-understanding contexts.

Overview. The remainder of this paper is organized as follows. Section II describes related research. Our approach chosen is described in Section III and its observations are reported in Section IV. Threats to validity are discussed in Section V. Section VI, finally, concludes.

II. Related Research

The focus of our research is the question whether program aspects measured by software metrics can be used to determine program complexity from the perspective of programmers. This section describes related research in this area, namely, the metrics investigated in this study, means of determining program complexity, and program aspects influencing perceived complexity.

A. Metrics

There is an abundance of software metrics. In this section, we describe only those evaluated in our study. Our focus is on intraprocedural control and data-flow metrics.

For control-flow metrics, we consider McCabe's Cyclomatic Complexity (*CyclCompl*) [2], that is, the number of predicates plus one, and *Npath* [3], that is, the number of acyclic execution paths.

978-1-4673-1216-5/12/$31.00

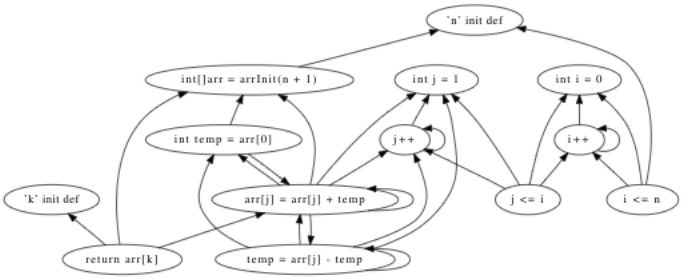

Fig. 1: Def-Use Graph for `bico` from [4]

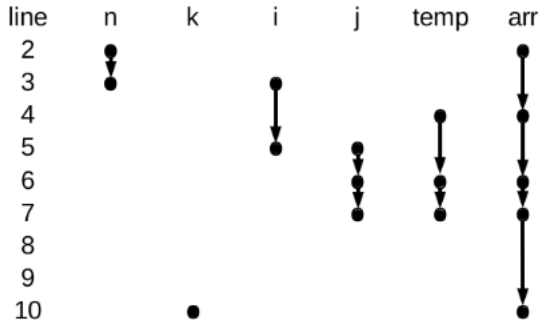

Fig. 2: Lifespan Example

Listing 1: Bico Sample

```
1  int bico(int n, int k) {
2    int[] arr = arrInit(n+1);
3    for (int i = 0; i <= n; i++) {
4      int temp = arr[0];
5      for (int j = 1; j < i; j++) {
6        arr[j] = arr[j] + temp;
7        temp = arr[j] - temp;
8      }
9    }
10   return arr[k];
11 }
```

As data-flow metrics, we choose Dep-Degree (*DepDeg*) [4] and *Lifespan* [5]. They are explained using listing 1, which is taken from Beyer et al. [4].

Dep-degree is a data-flow metric proposed by Beyer and Fararooy [4]. The metric is based on the def-use graph whose nodes represent statements and whose edges relate variable definitions and uses. Dep-degree is defined as the number of edges in this graph. Figure 1 shows the def-use graph for listing 1. The number of edges is 28, that is, *DepDeg* $= 28$ for method `bico`.

Lifespan is another data-flow metric descibed by Elshoff [5]. The idea is that many accesses to a variable and long distances between them increase complexity. A variable *occurs* in line if its name is used in this line (except for its pure declaration). A *span* of a variable is the number of lines of code between one textual occurrence and its immediate lexically subsequent textual occurrence. *Lifespan* is calculated as the average over all spans of all variables of a method. Let V be the set of variables occurring in a method. If a variable $v \in V$ has o_v occurrences, it has $m_v = o_v - 1$ spans. Then, the sum of all spans of all variables is $S = \sum_{v \in V} \sum_{i=1}^{m_v} span_i(v)$ and *Lifespan* is S divided by the number of spans as follows: $S / \sum_{v \in V} m_v$.

As an example, consider Figure 2. Variable n occurs in line 2 and 3. Therefore we have two occurences and $span_1(\texttt{n}) = 1$. Variable *temp* occurs in line 4, 6 and 7; therefore we have three occurences and two spans of $span_1(\texttt{temp}) = 2$ and $span_2(\texttt{temp}) = 1$, respectively. Overall, we have ten spans as follows:

variable	spans
`arr`	2, 2, 1, 3
`n`	1
`i`	2
`temp`	2, 1
`j`	1, 1
S	16

Therefore, *Lifespan* for `bico` is calculated as $16/10 = 1.6$.

In addition to that, we consider size metrics, namely, lines of code (*LOC*), Halstead's difficulty (*HalDiff*) [1], Cognitive Functional Size (*CFS*) [6] and *Simple Readability* (*SR*) [7]. Cognitive Functional Size (*CFS*) developed by Shao and Wang [6] belongs to a family of so called cognitive complexity measures developed in the field of cognitive informatics. The authors claim that they measure the difficulty of comprehension. *Cfs* itself is based on basic control structures (BCS). A BCS is a pattern in the control flow graph such as sequence, branch, iteration, recursion, or function call. Based on empirical studies, the authors propose different weights for complexity for each type of BCS.

BCS may be in a linear layout or embedded in others. The total cognitive weight of a software component, W_c, is defined as the sum of the cognitive weights of its q linear blocks composed of individual BCSs. Since each block may consist of m layers of nesting BCSs, and each layer of n linear BCSs, the total cognitive weight, W_c, can be calculated by $W_c = \sum_{j=1}^{q} [\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i)]$ where $W_c(j,k,i)$ is the recursive weight of the i'th BCS nested in the k'th layer of the j'th linear block. *CFS* is then defined by $CFS = (N_i + N_o) \cdot W_c$ where N_i is the number of inputs and N_o the number of outputs.

Consider listing 1. We have a block consisting of a sequence (weight: 1) and an iteration (weight: 3) that has another iteration (weight: 3) embedded. We derive $W_c = 1 + (3 \cdot 3) = 10$ (note that only one sequence is counted per component). The numbers of inputs are $N_i = 2$, $N_o = 1$; therefore *CFS* for the `bico` example is $CFS = (2 + 1) \cdot 10 = 30$.

Posnett et al. developed a metric for readability of source code, called *Simple Readability* (*SR*) [7]. It is statistically derived from human-based ratings gathered in a study by Buse and Weimer [8]. They define $SR = \frac{1}{1+e^{-z}}$ where z is defined

as: $z = 8,87 - 0,033 \cdot V + 0,40 \cdot LOC - 1,5 \cdot Entropy$ and V is Halstead's volume and *Entropy* is a measure of entropy calculated from the counts of terms (tokens or bytes). *SR* for the `bico` example is 0.0035316.

Although their model has better performance than the original model by Buse and Weimer according to the human ratings collected in Buse and Weimer's study, it is hardly applicable on different data sets because it was trained on a data set with very small metric values. For our selected set of methods, there is a high negative correlation between *SR* and *LOC* ($\rho = -0.82$). We will further investigate it in our study in Section III. Unfortunately, the original model by Buse and Weimer could not be used in our evaluation because it always evaluates to 0, when it comes to larger code fragments.

B. Research Instruments for Assessing Program Complexity

Program complexity relates to the difficulty to understand a computer program. Different research instruments have been used in empirical research to investigate factors affecting program complexity. This section describes these instruments and explains why we choose questionnaires as our means.

Boysen was among the first ones proposing a concrete objective measure for program understanding [9], [10], which he took from cognitive sciences on reading. He proposed to use the reaction time required to answer a question relating to a program. The measure is instrumented by showing a program to a developer and then asking whether a certain statement holds. The longer the reaction time, the more complex the program is. In essence, what Boysen proposed is a controlled experiment using time for task completion as a measure. Virtually all controlled experiments on program comprehension measure the time for a task (in addition to its correctness). They vary mostly in the selection criteria for experimental subjects and objects, and types of tasks. Reaction time and all other measures used in controlled experiments are objective to the degree of how representative the people, programs, and the types of tasks or questions are. The advantage of experiments is that they provide a way to objectively measure and observe a cause-effect relation. A drawback of controlled experiments is that they are relatively expensive to conduct and generally do not scale well. That is, they are also often biased by relatively small samples in software engineering. Furthermore, experiments are often limited by the rigid setting an experiment requires. For instance, the outcome may only be valid for the particular task that was given to experimental subjects.

Other researchers have used field observations to assess program comprehension aspects [11]–[13]. In field studies, researchers observe how programmers understand programs in their normal environment and tasks. Observational studies are less controlled than real experiments, which is both an advantage (unforeseen things may be observed in a real setting) and disadvantage (because variables influencing the observation are not controlled). If the observation is not automated (as for instance, in analyzing automatically recorded interaction history [14]), field observations require the presence of the observer and are, hence, also costly.

Another means to assess program complexity is to use statistical correlation between factors and indirect measures – or surrogates – of complexity. One such indirect measures, for instance, is program defectiveness. The reasoning here is that if a program is complex, it should tend to have more defects. There is a very large body of research devoted to predicting defects based on software metrics and other measured factors that uses this type of research instrument. The independent variables to be correlated with defects include code complexity measures [15], code-change metrics [16], object-oriented metrics [17], dependencies [18], [19], or organizational factors [20]. Typically, a binary categorization into *defective* and *not defective* is predicted. Only a few studies so far have predicted defect densities: Knab et al. [21] and Ostrand et al. [22] predict defect densities and Nagappan and Ball build regression models to directly predict defect densities [23], [24]. For a more comprehensive summary of this line of research, we refer the reader to a recent survey by Mende [25].

The problem with such indirect measures is that they do not control the influence of the programmer's skill, exact circumstances, and the test effort. For instance, a very complex piece of code may have been developed by highly skilled developers and be thoroughly inspected and tested by others. That is why it has a very low defect rate despite its complexity.

In this paper, we choose a questionnaire as a research instrument – that is, a series of questions and other prompts for the purpose of gathering information from respondents – in which we ask programmers for their opinions about the complexity of a set of programs.

The advantages of questionnaires are that they require relatively little effort for participants, do not necessarily need the presence of researchers, and, hence, can reach a much larger group of people. As opposed to other research instruments such as controlled experiments, questionnaires may scale well. Their disadvantage is that they are subjective and their measures are often only at an ordinal scale.

The work in this area that comes methodologically closest to ours is an investigation of readability by Buse and Weimer [8]. They studied aspects of program readability using a web-based survey that collected 12,000 opinions from 120 students at the University of Virginia. Each student had to rate 100 small code snippets (on average 7.7 LOC, including comments) on an absolute scale from 1 to 5 for readability. The aspects of readability were many detailed issues such as layout, the occurrences of certain token types, length of identifiers, etc. Their goal was to build a model to predict readability by using machine learning techniques. Our work, on the other hand, focuses on control and data-flow metrics and uses paired comparisons, which makes it possible to detect inconsistent ratings of a single rater. We argue that asking a person to rate 100 snippets on an absolute scale bears the risk of collecting unreliable data because of effects of fatigue or learning and there is no measure used to uncover inconsistency. As it turns

out in our study, human raters tend to make inconsistent judgments even for few exemplars to be compared pairwise.

C. Investigated Complexity Aspects

Kemerer gave a comprehensive survey on metrics suggested to assess program complexity for the body of research up to the year 1995 [26]. The research of that time mostly focused on aspects of size [27]–[30], structured programming [31], modularity [32], and coupling [33]–[35]. The studies on size suggested that smaller modules have a disproportionately high share of defects; those on modularity indicate that more modular programs are in fact easier to modify, and the studies on coupling found that highly cohesive but loosely coupled modules were less likely to require modification, and use of global variables increases the chances for defects and changes.

Researchers of that time also looked into relations among different types of metrics. Lind et al. [36] found a high correlation between the more complex metrics and size measured as lines of code (LOC) and LOC had the highest positive correlation with development effort in their study. A study by Gill et al. [37] suggested that normalizing McCabe's *Cyclomatic Complexity* by LOC ("complexity density") is a more useful explanator of software maintenance productivity.

Since then, aspects specific to more recent programming paradigms such as object-orientation [17], [38], [39] or aspect-orientation (e.g., [40]) have been investigated.

Research has also extended into other aspects of programming including comment style [41], identifier styles [42]–[44], identifier density within slices [45], bad smells [46]–[50], and the use of design patterns [51].

III. Study Design

We chose a questionnaire as a research instrument for the reasons stated in Sections I and II-B. This section describes the necessary steps in conducting research on software characteristics based on questionnaires with multiple raters. The approach described here may serve as a model for conducting similar research. Because of limited space, we cannot describe all aspects and alternatives, but at least we mention issues and give references for further reading.

The necessary steps are as follows:

1) formulate an operational hypothesis that may be answered by questionnaires
2) determine the target audience of the questionnaire
3) design the questionnaire
4) test and refine the questionnaire in pilot studies
5) distribute the questionnaire among participants; collect the results
6) determine consistency for each rater; manage inconsistencies of individual raters (e.g., filter them out)
7) determine concordance, that is, the agreement among all raters
8) test the hypothesis against the raters' responses and determine the significance of this test

a) Hypothesis: The assumption of existing research on metrics is that they are capable of predicting cognitive complexity in a programmer's attempt to understand a program. Since we chose to compare against human ratings, the operational hypothesis investigated in our study is as follows:

> The ranking produced by a metric based on intraprocedural control and/or data flow at the method level correlates positively with rankings produced by programmers for perceived program complexity.

The hypothesis relates to methods because the metrics we wanted to assess are based on intraprocedural control and data flow. Methods are also a meaningful chunk for programmers.

b) Target Audience: Our target audience are programmers with sufficient knowledge in Java. We chose Java because we expected a larger reachable audience proficient in that language. Other than that, we used convenience sampling to define our target audience.

c) Design of the Questionnaire: One central aspect for the design and evaluation is the nature of the data asked. Typically, the data are at a nominal or ordinal scale when it comes to human-based ratings. In our case, we asked programmers which Java methods are perceived more complex. That is, we asked participants of the questionnaire to state their opinion whether the complexities of two methods are less ($<$), greater ($>$) or equal ($=$), that is, the scale of the gathered data is ordinal. We can hardly expect programmers to assign an absolute complexity grade to methods (a ratio scale). Even asking for a meaningful numeric difference in the complexities of Java methods (an interval scale) can hardly be justified (e.g., "method a's complexity is 5 complexity units more complex than method b").

There are different alternatives in a questionnaire for ranking objects at an ordinal scale depending on the number of objects to be ranked. If the number of objects is low, one can ask programmers to order them all at once. If the number of objects is high, one could use the method of successive intervals. Here, programmers are asked to put objects into buckets of similar characteristics. For instance, one could specify categories such as *very complex*, *complex*, *neutral*, etc. for program complexity. All objects in the same bucket would be treated as of equal rank, which would lead to ties. This choice must be carefully considered because specific statistics must be used if there are too many ties.

We chose to use a paired comparison instead of sorting all objects at once because we believe that it is cognitively easier to make a decision for two concrete examples than between multiple objects at the same time. The disadvantage is that the paired comparison may produce inconsistent rankings, that is, pairs of comparisons contradict each other because of circularity or nontransitivity. We will delve into this aspect below in more detail. On the other hand, uncovering such inconsistent rankings is also a chance to assess how reliable the judgment of a rater is. If a rater is forced to specify a total order considering all objects at the same time, chances are that

he or she makes arbitrary decisions that are only artificially consistent.

A participant was shown two methods, A and B, at a time and had to select one of four possible answers: $A > B$ when A is more complex than B, $A < B$ when B is more complex than A, $A = B$ when A and B are equally complex, or *Don't know* if the participant was unable to make a judgment. The guiding questions for this decision were stated as:

- In which method would you expect more errors?
- Where would you hesitate more to make any change?
- If you have time for reviewing only one of the two methods, which one requires a review more urgently from your point of view?

In addition to the rating, we asked participants for their confidence in their ratings using four categories: *strongly confident*, *weakly confident*, *weakly unsure*, *very unsure*. The option *strongly confident* was the default. We believed that participants' confidence is generally high and this default increases their convenience.

We wanted to use realistic methods and, hence, selected methods from two open-source projects, namely, *JEdit* and *Hibernate* (cf. Table I).

TABLE I: Selected Projects (LOC Measured with *sloccount*)

Project	LOC	Revision/Version
JEdit	105,353	19931
Hibernate-Core	296,395	4.0.0.Final

Our aim was to compare the control and data-flow metrics described in Section II-A. They do not take into account more advanced programming features such as threads, try-catch-blocks, synchronized-blocks, or anonymous classes, which possibly have an influence on program complexity. For this reason, we excluded methods with these features from our study.

For each project, we created one group of methods based on control flow (CF) assessed by *CyclCompl* and another one based on data flow (DF) assessed by *DepDeg*. The methods within a group were selected to have sufficient differences for the respective metric used to form the group. Additionally, we made sure that Spearman's correlation between *CyclCompl* and *DepDeg* was nearly 0. Altogether, we have four groups, namely: *JEdit-CF*, *JEdit-DF*, *Hibernate-CF*, and *Hibernate-DF*, each containing five methods.

We were not aware of methods with the same semantics but different metrics (also known as semantic clones) in the two selected systems. That is, we are comparing methods with different semantics. The alternative would have been to create methods ourselves with the same specification but alternative implementations. However, creating such artificial methods ourselves could introduce a bias. Instead we opted for more "natural" methods. Without any context, it is hardly possible for the participants to guess the purpose of the methods. That is why we argue that the comparison was purely structural. We think that semantics has hardly played a role beyond the operational semantics of low-level programming constructs.

Participants were assigned randomly to one project. The only constraint we had on the random assignment was that the number of respondents for each project was balanced. Each participant had to compare the methods of two groups within that project, one group formed for *CyclCompl* and one formed for *DepDeg*. Each group had $m = 5$ methods. A rater had to make $\binom{m}{2} = 10$ comparisons for all methods within the same group. Altogether, a participant had to make 20 comparisons. To mitigate the influence of learning effects, all methods in one group were shown to the participant before he or she started the paired comparison.

We removed all comments and normalized the code using a pretty printer. The code was presented with syntax highlighting because that is common in nowadays IDEs.

d) Pilot studies: Pilot studies aim at testing the questionnaire. We ran three subsequent pilot studies until the final questionnaire stabilized. The pilot studies aimed at the amount of time needed for each participant (which we wanted to limit to 20 min because shorter times increase the chances of completed questionnaires), comprehensibility of task assignments, and other things. As a feedback from these pilots, we lowered the number of comparisons and added means to express certainties of the judgment and to make comments. Initially, we also used identifier obfuscation to limit the influence of varying identifier quality among methods. It turned out that obfuscated identifiers made it very difficult to memorize variables for participants in their short-time memory. For this reason, we used the original identifiers of the programs for the real run of the study, where we tried to select methods with similar identifier quality.

e) Questionnaire Distribution: We ran a web-based survey to get opinions from programmers. To collect opinions online has the advantage that participitation is independent of time and location. The survey was carried out from 22 December 2011 until 31 January 2012 and was advertised by personal e-mails to colleagues and on Facebook and Twitter. Serval colleagues further distributed the announcement to their students and colleagues. That is, participants are primarily from academic institutions, basically students and professors. Demographic statistics are given in Section IV-A.

f) Consistency Analysis: Participants can give inconsistent answers by chance or on purpose. Therefore, it makes sense to have a consistency test to exclude all inconsistent opinions. The way of asking is important for a consistency analysis. When using direct rankings of n objects given all objects at the same time, respondents are forced to generate consistent rankings, We chose a paired comparison. Inconsistencies in a paired comparison arise from non-transitive decisions (i.e., $a < b \wedge b < c \wedge a \geq c$) and circular decisions (i.e., $a < b \wedge b < c \wedge c < a$).

Kendall provides a consistency check based on circular triads [52]. Unfortunately, his method addresses only the operators $>$ and $<$; it cannot be used if $=$ is a valid decision. For this reason, we instead represent a rater's decision as a graph – named *single-rater graph* – and check this graph for

transitivity and absence of cycles. In total, each rater makes $\binom{m}{2}$ paired comparisons if there are m objects. The objects are represented by m nodes in the graph and all $\binom{m}{2}$ paired comparisons are represented as directed edges. There is an edge from a to b if the decision was $a > b$. If the decision was $a = b$, there will be an edge from a to b labeled by =. All nodes connected by edges labeled by = are collapsed into one node. Cycles in the collapsed graph indicate inconsistent ratings.

g) Concordance Analysis: Consistency relates to a single rater whereas concordance relates to inter-rater agreement. Thurston proposed a method called *Law of Comparative Judgment* [53] to determine concordance. Unfortunately, it is applicable only when objects cannot be rated equal. If equality is a valid decision, Fleiss's Kappa coefficient [54] can be used instead. The Kappa coefficient κ lies in [0,1] and indicates the level of agreement among raters where 1 indicates total agreement among all raters.

Fleiss's Kappa is defined for values on a nominal scale, but is nevertheless applicable in our setting, which has ordinal data. We can view a paired comparison between two objects, a and b, as a composite ordered pair (a, b) that is put into one of three categories labeled $<$, $>$, or $=$.

If there is little concordance among the human raters, it is difficult to assess metrics with this human oracle. In such cases, one might be able to find partitions of raters with higher concordance and one could investigate their characteristics that explain lower concordance with other groups (for instance, level of experience).

h) Test Hypothesis: The next step is to compare the metric's ranking against the consistent human rankings. Different options are available to form an oracle for this matching. One can derive a unified judgment from the agreeing individual ratings or one can compare the metric ranking against each single rater and compute a statistical agreement between the individual human and metric rankings.

For the latter comparison, it can be checked for every edge in a *single-rater graph* whether the direction is in accordance with the direction the respective metric provides. For m objects compared by n raters, there are $\binom{m}{2} \cdot n$ paired comparisons that can be matched against each metric.

One problem here is that metrics and human rankings are at different types of scale (interval vs. ordinal scale). At ordinal scale, the difference between two objects is meaningless as opposed to an interval scale. This circumstance poses a problem in particular for equality in a human ranking. For instance, if a human decided that two objects, a and b, are equally complex and McCabe's complexity for a is 1234 and for b it is 1235, then $a < b$ holds from the perspective of this metric, but the difference is likely negligible from a human rater's point of view. One could define a threshold ϵ and consider all metrics $|a - b| < \epsilon$ to be equal, but it is unclear how to select and justify an appropriate value for ϵ. For this reason, we decided to ignore equal ratings in the single-rater evaluation. We note that this decision is to the advantage of a metric because none of the gathered values for the methods considered in our study are equal. That is, from the perspective of a metric, all methods are different, but some human raters may have decided that they are alike.

To quantify how well metric values match programmers' opinions, we calculate a matching index for every single-rater graph. Let $G = (V, E, s, t)$ be a single-rater graph of one particular rater, where s is a function $E \to V$ that yields the source of an edge and t yields its target and invariant $s(e) > t(e)$ holds according to the rater. Let $a_m : V \to \Re$ be a function that assigns values of metric m to each node. The matching index M_m is calculated as $M_m = \frac{|M_e|}{|E|}$, with $M_e = \{e | e \in E \wedge a_m(s(e)) > a_m(t(e))\}$. This measurement of the agreement with a single rater can be extended to multiple raters by taking the average over all single-rater graphs.

Another approach for measuring the agreement of a metric ranking with human rankings is to take the majority opinion into account. This can be done by inferring a *multi-rater graph* from all *single-rater graphs*. An edge in a *majority-multi-rater graph* exists between two nodes if the (simple) majority of all raters share this opinion. The problem here is that the least neccessary fraction of votes for a simple majority is 33 % plus one vote since we have three possible votes $(<, =, >)$. We therefore take also a second more appropriate approach – using a *distinct-majority-multi-rater graph* – into account. In this graph, there is an edge only if the majority vote is supported by sufficient agreement. To measure agreement, we use an intermediate measure by Fleiss. Fleiss's test works for m objects (here: pairs of methods) put into k categories (here: the relations $<, =, >$) by n raters. For the i'th object, the agreement is calculated as the number of agreeing pairs in proportion to all possible pairs: $P_i = \frac{1}{n(n-1)}(\sum_{j=1}^{k} n_{ij}^2 - n)$ where j is a category and n_{ij} is the number of participants who have put object i in category j. This formula indicates the expected percentage of full agreement of two randomly selected raters for one single object. We used two different thresholds for Fleiss's P_i: 0.5 and 0.6 in the study.

The agreement between both *majority-multi-rater graph* and *distinct-majority-multi-rater graph* and a metric's ranking can be determined as for single-rater matching using M_m from above for a metric m. Matched edges in propotion to all possible edges can be interpreted as how well the metric matches the agreed upon ranking of multiple raters.

IV. Results

This section describes the results. We first describe general data on the survey and then validate our hypothesis.

A. Meta Data

We collected opinions from 206 participants who filled out the questionnaire completely. Additional 126 participants read the instructions and left without any rating and additional 53 did some but not all (7 ratings on average). However, the survey was anonymous and so it is not clear whether some of the users who quit came back and participated later.

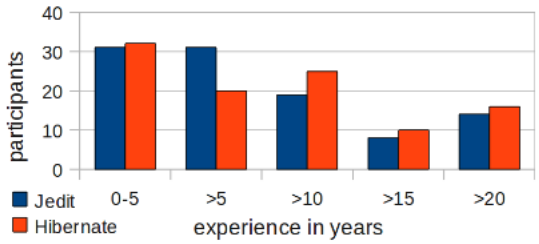

Fig. 3: Participants Experience

Finishing participants needed 12.6 minutes on average for all 20 ratings. Age and level of programming experience are similarly distributed in both groups of projects (cf. Figure 3). The level of significance (also known as critical p-value) in our study here and below is uniformly $p = 0.05$. At that level, there are no significant differences in time needed and experience between the different groups.

Participants are significantly less confident in project *Hibernate* than in *JEdit*(p=0.00032). Altogether, participants in *JEdit* put significantly more confidence in their single ratings (p=0.012).

B. Data Analysis

i) Consistency: Out of 412 *single-rater graphs*, 204 had to be removed due to inconsistency.

j) Concordance: For all groups, we calculate Fleiss' Kappa's coefficient as an indicator of agreement between participants. κ lies between 0.21-0.40 (*JEdit-CF*: $\kappa = 0.254$, *JEdit-DF*: $\kappa = 0.311$, *Hibernate-CF*: $\kappa = 0.335$, *Hibernate-DF*: $\kappa = 0.221$). These values seem to be relatively small at first glance. We note, however, that the larger the number of categories, the greater the potential for disagreement, with the result that Fleiss's Kappa will be lower with many categories than with few [55]. Furthermore, if we had 80 % of the participants agreeing on category $<$ and 20 % agreeing on category $>$ for each pair of methods, the value of Fleiss's Kappa would be 0.314, although intuitively we would believe the agreement is considerable. According to Landis [56], the values for Fleiss's Kappa we observed can be interpreted as *fair agreement*.

Our null hypothesis is that there is no agreement among raters ($\kappa = 0$). For all groups but *Hibernate-DF*, the null hypothesis must be rejected at $p = 0.05$ according to the significance test for Fleiss's Kappa.

k) Metrics evaluation: For all metrics we calculate the matching index M_m for all *single-rater graphs* to see how metrics are in accordance with programmer opinions.

As a baseline to get an idea on how to interpret the numeric accordance M_m, we added two other metrics to our evaluation: the occurrences of the character = and a senseless metric. The number of occurrences of the character = (*NOES*) counts all textual occurrences of = within a method. Intuitively, there should be a correlation between *NOES* and other data-flow metrics as this character is used for the assignment operator. Indeed there is a higher correlation of *NOES* with *DepDeg* (0.6) than with *CyclCompl* (0.48) in our data set. The detection of this character is purely text-based. Because = is also part of the relational operators ==, <=, and >=, the metric does not truly count assignment operators. Yet, at least it is not influenced by comments and literals because comments were excluded and = did not occur in any of the string literals in our selected methods. This metric is much simpler to gather than any data-flow metric.

Beyond that, we added a meaningless metric *Foo* calculated as the integer representation of the MD5 hash code of a method's source codes modulo 399. This metric is meaningless and should not have any correlation with program complexity. It fails only Property 5 (monotonicity $\forall P \forall Q(|P| \leq |P;Q| \wedge |Q| \leq |P;Q|)$ where $|P|$ denotes the complexity of program P) and Property 8 (if P is a renaming of Q, then $|P| = |Q|$) of the nine properties suggested by Weyuker for complexity measures [57]. It is interesting to compare the accordance of such a meaningless metric to other metrics.

Figure 4 shows a boxplot of all calculated matching indices M_m. Calculated means are as follows:

CyclCompl	*Npath*	*DepDeg*	*Lifespan*	*HalDiff*
0.6167	0.5791	0.6962	0.7433	0.7169
LOC	*CFS*	*NOES*	*Foo*	*SR*
0.6854	0.6541	0.7428	0.7243	0.2755

There is no significant difference between the control-flow metrics *CyclCompl* and *Npath* ($p = 0.106$). Data-flow metric *Lifespan* has significantly better accordance to programmer opinions than *DepDeg* ($p = 0.027$). Both data-flow metrics have a significant better accordance than control-flow metrics. *Foo*'s accordance is better than *CyclCompl* and *Npath* and similar to *DepDeg*, *Lifespan* and *HalDiff*. *NOES* is better than *CyclCompl*, *Npath* and *DepDeg* and similar to *Lifespan* and *HalDiff*. The matching of *SR* is very low. The reason might be that we influenced its basis by removing comments and normalizing code. Nevertheless, it seems this type of readability metric can hardly be used when it comes to larger code fragments.

In addition to that, the metric matchings for *majority-multi-rater graphs* as well as *distinct-majority-rater graphs* are shown in Table II. We used two different thresholds 0.5 and 0.6 of Fleiss's P_i for *distinct-majority-multi-rater graphs*. Although 0.5 seems to be very low, the propability is already higher than the expected value 0.33 (for 3 categories). With threshold 0.6, there are only 18 of 40 ratings selected overall, which is less than 50 %. However, there is little difference between the simple majority and the distinct ones (supported by Fleiss's P_i). Therefore any of these can be used to assess the ranking quality.

We note that metrics different from *CyclCompl* and *DepDeg* correlate with *CyclCompl* or *DepDeg* in some cases. For instance, *HalDiff* and *DepDeg* correlate in case of *JEdit-CF*. In such cases, we cannot tell whether *DepDeg* is better suited than *HalDiff*. We can always compare *DepDeg* to *CyclCompl*, however, because the methods were selected so that these

TABLE II: Multi-Rater Graph Accordance with Metrics

	Type	FunctionSets	*CyclCompl*	*Npath*	*DepDeg*	*Lifespan*	*HalDiff*	*LOC*	*CFS*	*NOES*	*Foo*	*SR*
JEdit-CF	MAJORITY	10/10	0.70	0.70	0.60	0.80	0.60	0.80	0.60	0.90	0.50	0.10
	DISTINCT. 0.5	7/10	0.86	0.86	0.57	0.86	0.57	1.00	0.71	1.00	0.57	0.25
	DISTINCT. 0.6	3/10	1.00	1.00	0.33	1.00	0.33	1.00	0.67	1.00	1.00	0.00
JEdit-DF	MAJORITY	10/10	0.90	0.70	0.60	1.00	0.70	0.70	0.80	0.90	1.00	0.40
	DISTINCT. 0.5	7/10	0.86	0.86	0.86	1.00	1.00	0.57	0.71	1.00	1.00	0.43
	DISTINCT. 0.6	6/10	0.83	0.83	0.83	1.00	1.00	0.67	0.83	1.00	1.00	0.00
Hibernate-CF	MAJORITY	10/10	0.50	0.30	0.80	0.90	0.80	0.70	0.70	0.80	0.90	0.10
	DISTINCT. 0.5	9/10	0.56	0.33	0.78	0.89	0.78	0.78	0.78	0.89	0.89	0.00
	DISTINCT. 0.6	6/10	0.50	0.17	1.00	1.00	1.00	0.83	0.83	0.83	1.00	0.00
Hibernate-DF	MAJORITY	10/10	0.60	0.50	0.80	0.60	0.80	0.70	0.70	0.50	0.50	0.00
	DISTINCT. 0.5	5/10	0.40	0.80	1.00	0.40	1.00	0.80	0.60	0.80	0.80	0.00
	DISTINCT. 0.6	3/10	0.33	1.00	1.00	0.33	1.00	1.00	0.67	1.00	1.00	0.00
Overall	MAJORITY	40/40	0.68	0.55	0.70	0.83	0.73	0.73	0.70	0.78	0.73	0.17
	DISTINCT. 0.5	28/40	0.68	0.68	0.79	0.82	0.82	0.79	0.71	0.93	0.82	0.15
	DISTINCT. 0.6	18/40	0.67	0.67	0.83	0.89	0.89	0.83	0.78	0.94	1.00	0.00

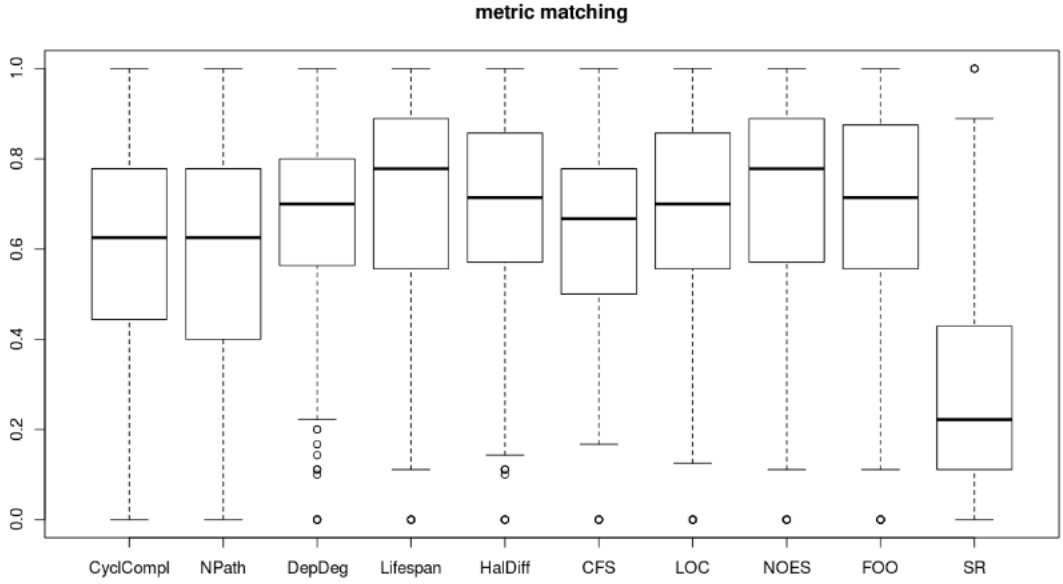

Fig. 4: Single-Rater Graphs Metric Matching

two never correlate. The results suggest that *DepDeg* reflects programmers' opinions slightly better than *CyclCompl*.

It is interesting to see that the very simple data-flow approximation *NOES* performs similarly to the other evaluated data-flow metrics that are much more expensive to compute. Likewise, the fact that the meaningless metric *Foo* has such a high accordance questions the accordance of the other metrics.

V. Threats to Validity

We have already noted that questionnaires are subjective. This section discusses additional threats to validity.

We used convenience sampling to gather participants by sending out e-mails to colleagues – the vast majority working in academia. Many of them distributed the questionnaire to their students. Consequently, most participants are from academia. Professional programmers might judge differently. However, many of the participants had substantial programming experience.

Although we collected a high number of ratings, only half of them are consistent. This high degree of inconsistency may partly arise from problems in the experimental design and implementation. For instance, one participant complained he did not find a way to go back and correct his false rating. It may be the case that inconsistent ratings come from these accidentally false ratings. Despite removing inconsistent ratings from the study, we still had more than 200 ratings.

Our results are based on methods implemented in Java. Although we excluded language features special to Java, it is not quite clear how results apply to other programming languages. Furthermore, methods were small (12-51 lines of code) to limit the influence of size and we ignored interprocedural flows and object-oriented design aspects, which may have a greater impact on comprehension. That is, we evaluated only a limited

number of metrics. We made all data available[1] so that other researchers can try other metrics.

We used only two open-source projects (JEdit, Hibernate). Selecting other projects especially other application domains might bring different results.

For *Lifespan* and *CFS*, there was no reference implementation available, so we had to implement ourselves. We may have misinterpreted details.

We did not specify any rating criteria, which gives more room for different interpretations. On the other hand, predetermining aspects may introduce a bias by the experimenter.

VI. Conclusion

This work presents an empirical evaluation of metrics based on a questionnaire. We provide a model how methods from behavioral science can be applied to gather information about program comprehension.

We collected opinions from over 200 programmers and matched their rankings with metric values. Although, we did not predetermine any rating criteria, there seems to be a fair agreement among programmers on program complexity. Data-flow metrics seem to better conform to programmers' opinions than control-flow metrics. Nevertheless, no metric fully agrees with the intuitive understanding of complexity. A very simple data-flow metric *NOES*, which counts only assignments provides results similar to other data-flow metrics and requires less effort to compute. The comparatively well performance of the meaningless metric *Foo* raises the question whether any of the metrics reflects opinions at all.

As future research, we intend to investigate whether the subjective assessment of programmers can actually be confirmed by more objective measures in controlled experiments.

Acknowledgment

The authors would like to thank all participants of the survey. Furthermore we thank Dirk Beyer for providing us with his implementation of *DepDeg* [58]. This work was supported by research grants of the *Deutsche Forschungsgemeinschaft* (DFG).

References

[1] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[2] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.

[3] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Commun. ACM*, vol. 31, pp. 188–200, Feb. 1988.

[4] D. Beyer and A. Fararooy, "A simple and effective measure for complex low-level dependencies," in *International Conference on Program Comprehension*, 2010, pp. 80–83.

[5] J. Elshoff, "An analysis of some commercial PL/I programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 2, pp. 113–120, Jun. 1976.

[6] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69 –74, Apr. 2003.

[7] D. Posnett, A. Hindle, and P. T. Devanbu, "A simpler model of software readability," in *Working Conference on Mining Software Repositories*, 2011, pp. 73–82.

[1] http://www.informatik.uni-bremen.de/st/konferenzen.php

[8] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546 –558, july-aug. 2010.

[9] J. Boysen, "Factors affecting computer program comprehension," PhD thesis, Iowa State University, 1977.

[10] J. P. Boysen and R. F. Keller, "Measuring computer program comprehension," in *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*. ACM Press, 1980, pp. 92–102.

[11] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[12] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," *International Conference on Software Engineering*, pp. 344–353, 2007.

[13] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do industry developers comprehend software?" in *International Conference on Software Engineering*. ACM Press, 2012.

[14] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[15] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[16] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. of the 31st ICSE*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

[17] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, pp. 897–910, 2005.

[18] T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting Bugs from History*. Springer, Mar. 2008, ch. 4, pp. 69–88.

[19] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. of the 30th ICSE*. New York, NY, USA: ACM, 2008, pp. 531–540.

[20] C. Bird, N. Nagappan, P. T. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? an empirical case study of Windows Vista," in *Proc. of the 31st ICSE*, 2009, pp. 518–528.

[21] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proc. of the Workshop on Mining software repositories*. New York, NY, USA: ACM, 2006, pp. 119–125.

[22] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[23] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *International Conference on Software Engineering*, 2005, pp. 284–292.

[24] ——, "Static analysis tools as early indicators of pre-release defect density," in *International Conference on Software Engineering*. ACM, 2005, pp. 580–586.

[25] T. Mende, "On the evaluation of defect prediction models," PhD Dissertation, University of Bremen, Germany, 2011.

[26] C. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering*, vol. 1, no. 1, pp. 1–22, Dec. 1995.

[27] V. Basili and B. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[28] V. Shen, T.-J. Yu, S. Thebaut, and L. Paulsen, "Identifying error-prone software - an empirical study," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 317–323, 1985.

[29] B. Compton and C. Withrow, "Prediction and control of ada software defects," *Journal of Systems and Software*, vol. 12, no. 3, pp. 199–207, 1990.

[30] K. An, D. Gustafson, and A. Melton, "A model for software maintenance," in *International Conference on Software Maintenance*, 1987, pp. 57–62.

[31] B. A. Benander, N. Gorla, and A. C. Benander, "An empirical study of the use of the goto statement," *Journal of Systems and Software*, vol. 11, no. 3, pp. 217–223, 1990.

[32] T. D. Korson and V. K. Vaishnavi, "An empirical study of modularity on program modifiability," in *workshop on empirical studies of programmers on Empirical studies of programmers*, 1986, pp. 168–186.

[33] D. Troy and S. Zweben, "Measuring the quality of structured designs," *Journal of Systems and Software*, vol. 2, no. 2, pp. 113–120, 1981.
[34] R. Selby and V. Basili, "Error localization during software maintenance: Generating hierarchical system descriptions from the source code alone," in *International Conference on Software Maintenance*. IEEE Computer Society Press, 1988, pp. 192–197.
[35] S. Yau and P. Chang, "A metric of modifiability for software maintenance," in *International Conference on Software Maintenance*. IEEE Computer Society Press, 1988, pp. 374–381.
[36] R. Lind and K. Vairavan, "An experimental investigation of software metrics and their relationship to software development effort," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 649–653, 1989.
[37] G. Gill and C. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.
[38] J. D. et al, "The effect of inheritance on the maintainability of object-oriented software: An empirical study," in *International Conference on Software Maintenance*. IEEE Computer Society Press, 1995, pp. 20–29.
[39] T. Kamiya, S. Kusumoto, K. Inoue, and Y. Mohri, "Empirical evaluation of reuse sensitiveness of complexity metrics," *Information and Software Technology*, vol. 41, no. 5, pp. 297 – 305, 1999.
[40] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, "Exploring software measures to assess program comprehension," in *International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society Press, 2011, pp. 127–136.
[41] L. P. et al., "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 595–606, 2002.
[42] B. Sharif and J. Maletic, "An eye tracking study on camelcase and under score identifier styles," in *International Conference on Program Comprehension*. IEEE Computer Society Press, 2010, pp. 196–205.
[43] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What´s in a name? A study of identifiers," in *International Conference on Program Comprehension*. IEEE Computer Society Press, 2006, pp. 3–12.
[44] ——, "Effective identifier names for comprehension and memory," *Journal Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.
[45] J. Rilling and T. Klemola, "Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics," in *International Workshop on Program Comprehension*. IEEE Computer Society Press, 2003, pp. 115–124.
[46] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "Controlled experiment investigation of an object oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, Feb. 2003.
[47] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.
[48] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?" in *IASTED*. IASTED/ACTA Press, 2006, pp. 346–355.
[49] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of antipatterns on program comprehension," in *European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2011, pp. 181–190.
[50] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120 – 1128, 2007.
[51] S. Jeanmart, Y.-G. Gueheneuc, H. Sahraoui, and N. Habra, "Impact of the visitor pattern on program comprehension and maintenance," in *International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 69–78.
[52] M. G. Kendall, *Rank correlation methods*, 4th ed. London: Griffin, 1970.
[53] L. Thurstone, "A law of comparative judgement," *Psychological Review*, vol. 101/2, p. 273286, 1994.
[54] J. Fleiss, "Measuring nominal scale agreement among many raters," *Psychological Bulletin*, vol. 76, no. 5, pp. 378–382, 1971.
[55] J. Sim and C. C. Wright, "The kappa statistic in reliability studies: Use, interpretation, and sample size requirements," *Physical Therapy*, vol. 85, no. 3, pp. 257–268, March 2005.
[56] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. pp. 159–174, 1977.
[57] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1357–1365, 1988.
[58] D. Beyer and A. Fararooy, "Depdigger: A tool for detecting complex low-level dependencies," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, 30 2010-july 2 2010, pp. 40 –41.