Exploring Software Measures to Assess Program Comprehension

Janet Feigenspan*, Sven Apel†, Jörg Liebig†, Christian Kästner‡,

*University of Magdeburg, Germany †University of Passau, Germany ‡Philipps University Marburg, Germany

***Abstract*—Software measures are often used to assess program comprehension, although their applicability is discussed controversially. Often, their application is based on plausibility arguments, which, however, is not sufficient to decide whether software measures are good predictors for program comprehension. Our goal is to evaluate whether and how software measures and program comprehension correlate. To this end, we carefully designed an experiment. We used four different measures that are often used to judge the quality of source code: complexity, lines of code, concern attributes, and concern operations. We measured how subjects understood two comparable software systems that differ in their implementation, such that one implementation promised considerable benefits in terms of better software measures. We did not observe a difference in program comprehension of our subjects as the software measures suggested it. To explore how software measures and program comprehension could correlate, we used several variants of computing the software measures. This brought them closer to our observed result, however, not as close as to confirm a relationship between software measures and program comprehension. Having failed to establish a relationship, we present our findings as an open issue to the community and initiate a discussion on the role of software measures as comprehensibility predictors.**

***Keywords*-software measures, program comprehension**

I. Introduction

Software quality assessment is an important task in software engineering, because it can reduce maintenance effort, a main cost factor in software development [26]. A typical way to assess quality facets, such as program comprehension, is to use software measures. Software measures are computed based on properties of source code. The kind of property that influences software quality is based on a number of plausibility assumptions. For example, the more branching statements, such as *if* and *for*, a method has, the more difficult it is to understand, because more possible execution paths exists [28]. Or, the more operations a *concern* (i.e., code fragments that semantically belong together) defines, the more difficult it is to understand (concern operations [13]). Measures are a convenient and cheap way to assess software quality, because tools[1] can simply collect the numbers.

The simple extraction is one reason why software measures are widely accepted and used, despite warnings and empirical evidence about their drawbacks [3]. For example, there is a lot of work, in which different facets of aspect-oriented programming with AspectJ are compared to object-oriented programming with Java [5], [14], [18], [19], [25], [29] (see Table I for an overview of software measures and facets evaluated in recent publications at high-ranked conferences). This body of work is carefully designed and useful, because it systematically evaluates the claimed benefits of aspect-oriented programming. However, conclusions in favor of or against aspect-oriented programming are solely based on plausibility.

Our objective is to evaluate whether a relationship between software measures and program comprehension can be confirmed empirically. We argue that if software measures are suitable as comprehensibility predictors, then we should find a relationship in a controlled experiment, in which subjects work with source code. If we cannot find such a relationship, this would indicate that a correlation between software measures and program comprehension is not as trivial to describe or may not even exist.

To evaluate whether there is such a relationship, we conducted a controlled experiment, in which trained subjects worked with MobileMedia, a software for manipulating multimedia data on mobile devices [14]. We selected MobileMedia for two reasons: First, it is used in numerous studies, often with software measures, for several purposes (e.g., [2], [9], [14], [17], [29], [31]). This makes it a common system to compare results and to which we can relate the results of our experiment. Second, MobileMedia exists in two comparable versions with considerably different software measures. The difference between both versions was the target of the aforementioned prior studies. Hence, the authors ensured that both versions were carefully designed and code reviewed and that differences in software measures actually exist.

The experiment in a nutshell: To measure program comprehension, we designed maintenance tasks, in which subjects should locate the cause of a bug. If subjects submitted a solution, we can assume that a comprehension process took place [8]. From the answers of subjects, we analyzed the correctness and time of a bug fix. We expected a correlation between subject performance and software measures. However, the results of our experiment do not indicate such a relationship. To further explore our data for a possible relationship, we took into account the behavior of subjects during the experiment to compute software measures (e.g., the response time). However, we still could not detect a relationship. In the remainder of this paper, we give a short introduction to software measures and program comprehension and describe the experiment and our analysis in detail.

II. Software Measures

There are numerous software measures that target program comprehension. For better overview, we divide them into

[1]For example, SourceMonitor http://www.campwoodsw.com/sourcemonitor.html or ConcernMorph [16]

Reference	Conference	Goal	Software Measures
Bryant et al. [5]	AOSD '06	Composability, Modularization of Pattern Interactions	CBC, CDC, CDLOC, CDO, DIT, LCOO, LOC, NOA, WOC, Interaction Analysis
Figueiredo et al. [14]	ICSE '08	Changeability, Modularization, Dependencies	CBC, CDC, CDLOC, CDO, DIT, LCOO, VS, LOC, NOA, WOC, Added and Changed Elements, Interaction Analysis
Garcia et al. [18]	AOSD '05	Modularization	CBC, CDC, CDLOC, CDO, DIT, LCOO, LOC, NOA, WOC
Greenwood et al. [19]	ECOOP '07	Stability in the Face of Changes	CBC, CDC, CDLOC, CDO, DIT, LCOO, VS, LOC, NOA, WOC, Added and Changed Elements, Interaction Analysis
Kulesza et al. [25]	ICSM '06	Modularization, Maintainability	CBC, CDC, CDLOC, CDO, DIT, LCOO, VS, LOC, NOA, WOC
Molesini et al. [29]	WICSA '08	Stability in the Face of Changes	Added and Changed Elements

CBC: Coupling between Components, CDC: Concern Diffusion over Components, CDLOC: Concern Diffusion over LOC, CDO: Concern Diffusion over Operations, DIT: Depth of Inheritance Tree, LCOO: Lack of Cohesion in Operations, LOC: Lines of Code, NOA: Number of Attributes, VS: Vocabulary Size, WOC: Weighted Operations per Component

Table I
OVERVIEW OF SOFTWARE MEASURES USED IN SETTINGS WITH ASPECTJ AND JAVA PROGRAMS.

measures that describe complexity, program size, and separation of concerns. The intention of this section is not to give an exact definition of the measures or a complete overview of software measures. Our goal is to give an impression of software measures, their intention, and why their measured property of source code appears plausible.

Complexity Measures: Various measures exist that capture complexity facets of a software system. For example, McConnell defines complexity as the number of branching statements, such as *if*, *switch*, or *for*, of a method [28]. Source code with a low complexity value is preferable, because it has less possible execution paths a developer has to work with.

Size Measures: Size measures assess the size of a software system. The simplest measure is *lines of code (LOC)*, which represents the number of lines a program consists of [20]. This measure was developed to assess the size of fixed-format assembly languages, in which not much variability for implementing source code existed. Even for contemporary, more flexible programming languages, it seems plausible that the more lines a program has, the more difficult it is to understand, because a programmer has to consider more code.

Concern Measures: A *concern* is "anything a stakeholder may want to consider as conceptual unit [...]" [35]. *Separation of concerns* is a key principle in software engineering [32]. If a concern is not encapsulated in a module, a programmer has to trace it across the entire system (e.g., for modification), which is tedious and error prone.

Concern attributes (CA) and *concern operations (CO)* represent, respectively, the number of attributes and operations assigned to a concern [16]. The more attributes and operations a concern contains, the more facets a developer has to consider in order to understand a concern, which should make it more difficult to understand.

There are a lot of other measures that assess complexity, size, or separation of concerns of a software system.[2]. Each measure sounds like a plausible indicator for program comprehension. However, when taking a closer look at program comprehension, the plausibility fades: Program comprehension does not only depend on source-code properties, but also on the person who is working with source code. Experienced developers often use *top-down* comprehension [4], [38], which means that they are stating and refining hypotheses about the general purpose of a program, based on their knowledge. Unexperienced developers cannot use their knowledge, so they analyze the source code statement by statement and group statements to semantic chunks, which they compose to a hypothesis about the general purpose of a program. This is referred to as *bottom-up* comprehension [33], [37]. Typically, a developer uses both; top-down comprehension where possible and bottom-up comprehension where necessary. This is described by integrated models [39]. Hence, program comprehension is a complex process, and the relationship to software measures may not be as trivial as it seems at first sight.

III. EXPERIMENT

We give a detailed description of our experiment in this section. Further information on the experiment such as questionnaires, tasks, and computation of software measures are available on our project's website at http://fosd.de/exp_swm to enable other researchers to replicate the experiment.

A. Objective

The objective of our experiment is to evaluate the relationship between software measures and program comprehension. We decided to include four software measures that we introduced in Section II: the complexity value defined by McConnell as a representative of complexity measure and lines of code as a representative of size measure. Since separation of concerns is one important influence factor on program comprehension, we include both concern measures concern attributes and concern operations.

As stated previously, there are points in favor of software measures (plausibility, easy to apply) and points against them (program comprehension is very complex, prior empirical results [3]). Since we can argue (and others have)

[2]See [13], [20] for a more exhaustive overview

both in favor of and against a relationship between software measures and program comprehension, we do not state a specific research hypothesis in one direction or the other. Instead, we define a research question:

RQ: Is there a relationship between software measures and program comprehension?

B. Material

As material, we use MobileMedia, a medium-sized software product for the manipulation of multi-media data on mobile devices, which was implemented by Figueiredo et al. with the support of post-graduate students [14]. MobileMedia was developed in eight releases, of which we took the last, because it was most suitable for our research goal: The implemented concerns have considerably different software measures, which should make it easier for us to observe a difference in program comprehension of our subjects. We explicitly encourage the reader to compare both versions of MobileMedia and experience the differences first hand.

There are several benefits in using MobileMedia:

- MobileMedia was implemented in two versions, one in AspectJ (referred to as AspectJ version), the other (referred to as Java version) with Java ME and Antenna, a preprocessor for Java. Both versions were code reviewed, such that the same coding conventions have been applied to both versions. Moreover, exhaustive tests were conducted to assure that both versions are comparable. Because of the efforts of the developers (see [14] for more details), two comparable versions exist.
- Both MobileMedia versions considerably differ with respect to software measures. In Table II, we present software measures for the MobileMedia versions we worked with in our experiment (complexity and lines of code were computed by hand and with SourceMonitor. Concern attributes (CA) and concern operations (CO) by hand and with ConcernMorph [16]). We can see large differences between the two versions per concern, which are caused by the fact that in the AspectJ version, source code is separated into more, but smaller modules. If measures indeed describe program comprehension, then this large difference in software measures indicates a large difference in comprehensibility. For example, concern *Video* in the Java version has a 5 times higher complexity value, 86 % more lines of code, and about 4 times more attributes and operations than in the AspectJ version. Hence, measures suggest that the AspectJ version should be significantly better with respect to program comprehension than the Java version.
- Numerous researchers used MobileMedia in their studies [2], [5], [9], [10], [14], [17], [18], [19], [25], [29], [31]. Hence, there are a lot of results by other researches, which allow us to relate our work to them and vice versa. Consequently, the restriction to MobileMedia is not a drawback, but rather a benefit, because we contribute to the knowledge base regarding MobileMedia. Furthermore, MobileMedia is a good starting point for generalizing results to other software systems, because numerous different facets have been evaluated thoroughly.
- MobileMedia was developed as a software product line. A user can generate different variants of MobileMedia by selecting the desired concerns (e.g., one variant with concerns *CountViews* and *Favourites*, another variant without both concerns) [6]. To this end, the implementation of a product line must ensure that there is a mapping of concerns to the corresponding code units.

Concern	Version	Complexity	LOC	CA	CO
CountViews	AspectJ	0.97	319	2	21
	Java	3.39	1 268	27	41
PhotoAlbum	AspectJ	1.29	257	5	23
	Java	2.15	1 771	49	73
Favourites	AspectJ	1.70	257	3	19
	Java	3.39	1 268	27	41
Video	AspectJ	0.96	262	11	20
	Java	2.31	1 892	45	78
Music /MMAPI	AspectJ	1.05	326	12	24
	Java	2.32	2 081	63	88

Table II
OVERVIEW OF SOFTWARE MEASURES PER CONCERN.

To illustrate the commonalities and differences of both versions, we show equivalent code excerpts of each version in Fig. 1. The left part shows the AspectJ version. A *pointcut* (Lines 7 to 9) captures the execution of the method `initMenu()` in class `MediaListScreen`. When this method is executed, the *advice* (Lines 12 to 14) is executed, which adds the `sortCommand` to a menu.[3] In the right part of Fig. 1, we show the Java version that implements the same behavior as the AspectJ version, but uses #ifdef markers to map the `sortCommand` in class `MediaListScreen`.

To control the level of experience of our subject with a certain tool (e.g., call hierarchy in Eclipse), we implemented our own tool infrastructure with source-code viewing and a project-browsing component to display the source code. The tool uses Eclipse-like syntax highlighting and shows all files of the software system ordered by packages (similar to the package explorer in Eclipse). We implemented a logging functionality to track each action of our subjects during the experiment. We also used this tool for displaying the descriptions of the tasks and to capture the answers of subjects.

In addition to the source code, subjects got a feature diagram[4] of the product line on a sheet of paper and a mapping of files to concerns. Subjects were familiarized with feature diagrams before the experiment. The reason for providing both is to ensure that subjects direct their attention to those files that belong to a concern, which allows us to compare software measures and program comprehension at the concern level, not only at the level of the complete product line.

[3]See [23] for an introduction to AspectJ.

[4]A feature diagram is a graphical representation of concerns' hierarchy and their relationships [22].

```
public privileged aspect CountViewsAspect {
  // 147 additional lines of code...


  public static final  Command sortCommand = new
    Command("Sort by Views", Command.ITEM, 1);


  // pointcut declaration
  pointcut initMenu(MediaListScreen screen):
    execution(public void MediaListScreen.initMenu())
    && this (screen);

  // advice code
  after(MediaListScreen screen) : initMenu(screen) {
    screen.addCommand(sortCommand);
  }
  // 66 additional lines of code...
}
```

```
public class MediaListScreen extends List {
  // 39 additional lines of code...

  // #ifdef includeCountViews
  public static final  Command sortCommand = new
    Command("Sort by Views", Command.ITEM, 1);
  // #endif

  // 19 additional lines of code...
  public void  initMenu() {
    // 40 additional lines of code...

    // #ifdef includeCountViews
    this.addCommand(sortCommand);
    // #endif

    // 6 additional lines of code...
  }
}
```

Figure 1. Comparison of AspectJ and Java version of MobileMedia. Left: AspectJ version, showing pointcut expression and advice code. Right: Java version, showing #ifdefs to annotate code fragments.

Since the opinion of subjects regarding the experiment they participate in can influence their performance, we measured their opinion [21]. We administered a paper-based questionnaire after the experiment, in which we asked how difficult subjects found the tasks, how motivated they were to solve the tasks, and whether they think they would have performed differently with the other version of MobileMedia. Although subjects worked only with one version, they were familiar with both, AspectJ and Java, so we assume they can imagine how working with the other version would have been.

C. Subjects

Subjects were graduate students at the University of Passau. They were enrolled in the course *Contemporary Programming Paradigms (German: Moderne Programmierparadigmen)*, in which advanced programming techniques, such as including AspectJ and software product-line implementation with preprocessors, were taught and practiced. All subjects were aware that they are participating in an experiment and that their performance does not affect their grade for the course.

We split our sample in two groups: One group worked with the AspectJ version of MobileMedia (AspectJ group), the other group with the Java version (Java group). We decided not to let both groups work with both versions, because the experiment would have lasted considerably longer, which would have been too exhausting for our subjects. Additionally, subjects could have learned from the version they started with in the experiment, so we could not be sure how confounded our result would be. Furthermore, as explained above, both versions of MobileMedia were designed to be comparable. Hence, our setting allows us to draw sound conclusions without stressing our subjects too much.

To form two comparable groups, we measured programming experience, which is a major confounding variable for program comprehension. We carefully designed a questionnaire, which we administered before the experiment. In the questionnaire, we asked subjects to estimate their experience with several programming languages and paradigms on a five-point Likert scale [27], as well as the size of projects they have worked with. A low value in the questionnaire (minimum: 5) indicates no programming experience; the higher the value is, the more programming experience a subject has (high value: 60, the scale is open ended). The mean programming experience of the AspectJ group is 41.9 (standard deviation: 10.6), and of the Java group 40.8 (standard deviation 10.5). We had 21 subjects, of which one was female; she was in the Java group.

To account for the possibly more complex nature of AspectJ, we did not hide bugs in highly syntax-specific code. Since AspectJ is an extension to Java, its syntax is based on Java syntax, with additional elements, such as *pointcuts* (roughly similar to pattern matching) and *advice* (roughly similar to Java syntax). We introduced the bugs only to advice code and made sure that the claimed benefits of aspect-oriented programming for program comprehension, such as separation of concerns, could still be measured. However, since subjects did not have to implement any source code or understand complex pointcut declarations, a thorough understanding of AspectJ syntax is not necessary in our experiment.

D. Tasks

Program comprehension is an internal cognitive process that we cannot observe directly [24]. Instead, we have to find appropriate measures, such as bug fixing tasks [8], [12]. For our experiment, we created six maintenance tasks. In each task, we gave subjects a bug description as a user might provide it. Since we evaluate the relationship of software measures on the concern level, we supported subjects to work with source code implementing a concern: First, for each bug description, we provided the concern, in which the bug occurred. Second, we opened for each task all files that belong to the according concern. However, subjects could open all other files of MobileMedia, if they thought it was necessary (e.g., to trace a method call). For solving the tasks, subjects

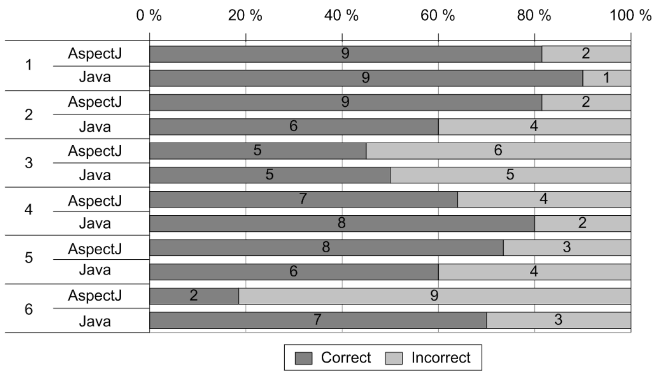

Figure 2. Frequencies of correct solutions.

should locate the position at which the bug occurs (file and line), state why the bug occurs, and suggest a solution. We used all information to decide whether a bug was identified correctly. Additionally, we measured the time subjects needed to solve a task (referred to as response time). For all tasks, we carefully introduced bugs into the source code.

To better understand of the nature of bugs, we describe the first bug in detail. The bug description subjects got stated:

> When creating/converting media, the counter how often a medium was shown, is always set to 0.
> The bug occurs, when concern *CountViews* is selected.

The bug was caused by setting a variable that counts the number of views to 0, instead of setting it to the correct value. In the AspectJ version, the bug was located in aspect *CountViewsAspect*. In the Java version, the bug was located at a corresponding position in class *MediaUtil*. The other bug descriptions can be found at the project's website.

In addition to these six tasks, we designed a warming up task to let subjects familiarize with the experimental setting. In the AspectJ version, subjects had to count the number of pointcuts of the concern *PhotoAlbum*, in the Java version how often the command *includeFavourites* of the concern *Favourites* occurs. We made sure that the effort for both tasks is comparable (i.e., that the same number of files were opened and had to be looked at and about the same number of occurrences existed). This task is not included in the analysis. For solving the tasks, subjects were instructed to type the answers in an according form, displayed as second window with our tool infrastructure.

E. Experiment Execution

The experiment was conducted in July 2010 instead of a regular lecture session. It was conducted in a lab room with Linux computers and 19" screens. We gave an introduction to all subjects, in which we explained important facets of the experiment and repeated facts from their programming course relevant for the experiment as a reminder. After the introduction, subjects were seated at a computer and could start to work on the tasks on their own. Four experimenters regularly checked that subjects worked as planned. After a subject was finished, she was instructed to raise her arm, so that we can give her the questionnaire to assess her opinion. After completing a questionnaire, subjects were instructed to leave quietly without disturbing the others. The conduction took place without any deviations.

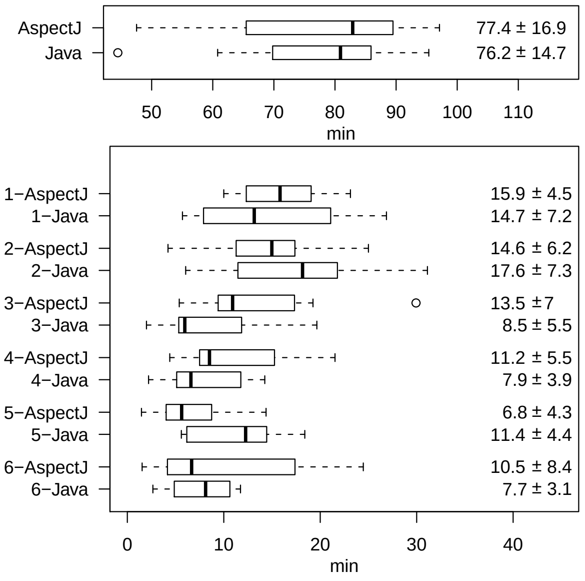

Figure 3. Response time of subjects. Top: for all tasks; bottom: per task. The numbers indicate the mean and standard deviation.

IV. Results

In this section, we present the results of our experiment. We start with statistical analyses, in which we describe the data. Furthermore, we present some anecdotal results

A. Observed Program Comprehension

To assess program comprehension, we measured whether a task was solved correctly and how much time subjects needed to solve the task.

1) Correctness: We present the results regarding correctness of solutions in Figure 2. We can see that for most of the tasks, both groups have about the same number of correct solutions. Only for the sixth task, there is a large difference: Only two subjects of the AspectJ group entered the correct solution.

To evaluate whether there are significant differences in the number of correct solutions, we conducted a χ^2 test [1]. Only for the last task, we found a significant difference in the number of correct solutions (χ^2 = 5.743, p = 0.017). For all other tasks, the χ^2 value is smaller than 1.222, and the p value larger than 0.269.

2) Response Time: In Figure 3, we show the mean response times of subjects for all tasks together (top) and for each task (bottom). We can see that for all tasks, the difference in response time is negligible with 2 % (1.2 minutes, compared to almost 1.5 hours for all tasks). Looking at specific tasks, we see that for the tasks 2 and 5, AspectJ subjects were faster; for the remaining tasks, Java subjects were

Task		1	2	3	4	5	6
Difficulty	U value	-0.953	1.485	-1.409	-0.217	0.000	-3.029
	p value	0.341	0.138	0.159	0.828	1.000	0.002
Motivation	U value	-0.044	-0.567	-0.914	-0.296	-0.404	-3.079
	p value	0.965	0.571	0.361	0.767	0.686	0.002
Other version	U value	0.039	-0.156	-0.124	0.909	0.769	1.150
	p value	0.969	0.876	0.901	0.364	0.442	0.250

Table III
MANN-WHITNEY-U TEST FOR SUBJECTS' OPINION.

faster. The largest differences appear in task 5 in favor of the AspectJ subjects, and in task 3 in favor of the Java subjects.

Before conducting significance tests for response times, we have to consider whether a task was solved correctly or not, because response times differ for correct and incorrect solutions [40]. For example, a subject might enter deliberately a wrong answer, just to be finished with a task, which would bias the response time. Hence, we excluded response times from the analysis, if a subject did not solve a task correctly. Unfortunately, for the last task, this leaves us with only 2 values for response times in the AspectJ group, so we cannot conduct a significance test for this task.

To check whether the observed differences in response time are significant, we conducted a Student's t-test for task 1 and 2 [1], and a Mann-Whitney-U test for tasks 3 to 5, of which the response times are not normally distributed. We found no significant differences in any tasks (all p values are larger than 0.117; largest t value: 1.032, largest U value: 1.567).

Taking both, the results for correctness and response time into account, we found no significant differences for program comprehension between the AspectJ and Java version, except for the correctness of one task. In next sections, we look for an explanation for this difference.

B. *Opinion of Subjects*

For all tasks, we asked subjects to rate their opinion regarding difficulty, motivation, and performance with the other version on a five-point Likert scale. We present the answers of subjects in Figure 4. For difficulty and motivation of the last task, there are large differences, such that AspectJ subjects found this task more difficult and were less motivated to solve it. For all other tasks, the median differed at most by 1.

To check whether the observed differences are significant, we conducted Mann-Whitney-U tests, summarized in Table III. The large differences for the last task (cf. Fig. 4) are significant regarding difficulty and motivation (p values $<$ 0.002). There are no other significant differences.

C. *Anecdotal Results*

In addition to the planned observations, we made some unexpected observations during the experiment. One was that when we told subjects which version they are assigned to, none of the Java subjects complained, but several AspectJ subjects did. This is slightly reflected in the opinion of subjects, in that AspectJ subjects found the last task more difficult and were less motivated to solve it than Java subjects. This could also explain the large performance difference for this task: Only 2 of the AspectJ subjects solved this task correctly. We believe that the opinion of subjects influenced their performance, which is a common phenomenon [30]. Interpreting the results in the context of a bigger picture, we should take into account the opinion of developers regarding the source code they work with, because unsatisfied developers may – consciously or subconsciously – reflect their opinion on their performance.

Furthermore, we analyzed the files AspectJ subjects looked at to solve the tasks. We found that they spent most of their time (means for each task vary from 88 % to 95 %) in those files that implemented a specific concern, and did only occasionally look into the code that is advised. Hence, subjects did not have to look at the base code to understand the implementation of a concern.

V. SOFTWARE MEASURES AND PROGRAM COMPREHENSION

In Section IV, we found that the differences we observed in program comprehension are not significant (except for correctness of the last task). This indicates that both versions are equally comprehensible, which is in contrast to what the software measures suggest (i.e., that the AspectJ version is more comprehensible). To interpret the observed data in terms of our research question, we relate the observed values in program comprehension to software measures.

After evaluating our research question, we go one step further and explore our data for a possible relationship. During this process, we refine the computation of software measures by including the behavior of subjects, such that we can have a detailed look on how software measures and program comprehension could correlate. Data exploration may sound like we are 'fishing for results' (we discuss this in Section VI). However, we did not specify a concrete research hypothesis, but only a research question: Is there a relationship between software measures and program comprehension? Thus, exploring our data is a legitimate step to answer this question. With our exploration, we provide some insights into possible relationships and concrete research hypothesis for future experiments.

A. *Software Measures of Complete System*

Although software measures are often calculated in terms of concerns, for completeness, we start in a general way by comparing the entire application. In Table IV, we present an overview of software measures for the complete system. The AspectJ version has more lines of code, more attributes, and more operations. In contrast, the complexity value is smaller in the AspectJ version. Since we did not observe a significant difference in program comprehension that reflects this difference in software measures, we cannot confirm a relationship between software measures and program comprehension on the level of the complete program. Next, we look at software measures and program comprehension at the concern level.

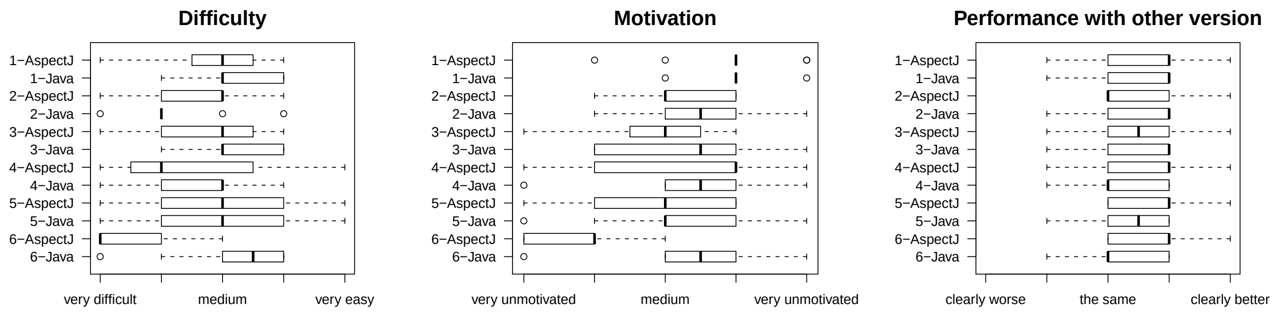

Figure 4. Opinion of subjects. Left: Difficulty, middle: motivation, right: performance with other version.

Version	Complexity	LOC	CA	CO
AspectJ	1.6	6 717	477	182
Java	2.0	5 397	271	165

Table IV
OVERVIEW OF SOFTWARE MEASURES OF MOBILEMEDIA.

B. Software Measures in Terms of Concerns

Software measures for the entire system do not necessarily reflect the subsystem analyzed for a specific task. Hence, we compare the software measures in terms of concerns with program comprehension as we observed it (cf. Table II for an overview of the software measures of MobileMedia in terms of concerns). The software measures for each concern of the AspectJ version are smaller, i.e., suggest better comprehensibility. This means that the AspectJ group should make fewer errors and be faster for every task. However, we could not find such a difference. Only for correctness of the last task, we discovered a significant difference, but in favor of the Java subjects (opposite of what the measures suggest). Hence, we cannot confirm that software measures and program comprehension correlate when considering the concerns subjects worked with.

One might argue that when subjects worked on a task, they did not only look at files that belong to the according concern, but opened other files, as well. Hence, we should compute software measures based on the *files subjects actually looked at*. Since we logged what subjects did during working on a task, including opening files, we are able to compute software measures based on the files subject looked at.

C. Software Measures Related to Files

To compute the software measures related to files for a single task, we determined ‘personal‘ software measures for each subject and computed their mean. We describe this aggregation process for one task. First, we extracted all files a subject looked at and determined the software measure for each file. Second, for each subject, we computed the average complexity value, and the sum for lines of code, concern attributes, and concern operations, respectively. Hence, each subject has her own ‘personal‘ value for complexity, lines of code, concern attributes, and concern operations. As last step, we averaged over all ‘personal‘ software measures.[5]

Task	Version	Complexity	LOC	CA	CO
1	AspectJ	1.12	440.18	8.64	29.64
	Java	2.95	1,290.30	28.50	39.70
2	AspectJ	1.38	409.36	7.36	30.55
	Java	2.24	1,658.20	44.90	66.70
3	AspectJ	1.66	369.18	5.55	25.27
	Java	3.04	1,149,50	27.50	34.80
4	AspectJ	1.02	481.00	15.36	32.91
	Java	3.07	1,210.30	31.70	44.40
5	AspectJ	1.33	382.82	12.27	28.27
	Java	2.58	1,896.30	55.70	74.50
6	AspectJ	1.45	501.27	8.82	33.09
	Java	3.48	1,167.90	24.20	36.30

Table V
OVERVIEW OF SOFTWARE MEASURES PER TASK.

We summarize the results of the adapted software measures in Table V. The difference of software measures between both versions is smaller now for most tasks and measures. This indicates that, if we take into account what subjects actually did, software measures better reflect program comprehension. However, the differences between software measures are still too large, compared to the fact that we did not observe significant differences in program comprehension.

Now, one might argue that subjects looked at one file only for a few seconds, but several minutes at another file. Hence, the *time a subject looked at a file* should also be considered, because the file at which the subject looked longer should have more influence on the software measure. Since we logged the time a subject looked at a file, we can compute weighted software measures.

D. Software Measures Weighted with Response Time

To compute weighted software measures for a single task, we again computed ‘personal‘ software measures for each sub-

[5]The reader may have noticed that if we compute ‘personal‘ software measures, we cannot automatically compute them based only on source code property. We discuss this issue in Section V-E.

Task	Version	Complexity	LOC	CA	CO
1	AspectJ	0.50	210.07	2.26	15.88
	Java	0.67	254.82	3.47	6.35
2	AspectJ	0.28	134.82	3.14	12.15
	Java	0.31	232.33	3.83	7.10
3	AspectJ	0.53	176.31	2.74	14.47
	Java	0.87	293.34	6.56	6.73
4	AspectJ	0.27	145.64	5.13	11.89
	Java	0.87	162.95	10.06	3.96
5	AspectJ	0.60	207.43	6.26	15.88
	Java	0.29	250.16	5.44	6.95
6	AspectJ	0.52	174.77	2.67	13.89
	Java	0.85	310.22	4.67	7.61

Table VI
OVERVIEW OF WEIGHTED SOFTWARE MEASURES PER TASK.

ject, and additionally took into account the time subjects spent with a file. We describe this approach for a single task. First, we divided the time a subject spent with each file by the complete time for a task. Second, we multiplied this value with the according software measures for a file. For example, if a subject looked at a complex file only for a few seconds, the weighted value for complexity is low. Hence, for each file and each subject, we got 'personal' weighted values for complexity, lines of code, concern attributes, and concern operations, respectively. Finally, we proceeded as described in the previous section for the computation of software measures related to files, based on the 'personal' weighted software measures.

In Table VI, we present the mean of weighted software measures for each task. The difference between the software measures of both versions got smaller again, compared to the unweighted values. Especially the complexity values are all smaller than 1, which indicates that complexity can be an appropriate measure if we consider subjects' behavior. The smaller difference for the other software measures aligns better with the results of the experiment we measured, as well.

Another interesting observation for weighted software measures is that the weighted value for concern operations is smaller in the Java version. This is most likely caused by the fact that AspectJ subjects spent most of their time in aspects with a large concern-operations value, because those contained most of the implementation of a concern. Hence, the weighted concern-operations values are more similar to the unweighted concern-operations values. In contrast to the AspectJ group, the Java group looked at more files per task, so the time per file is considerably smaller. Multiplying this small value with the concern operations value results in smaller weighted concern operations values, and thus in a smaller weighted value.

Finally, one might argue that we should also take into account the methods of each file subjects looked at, because a very complex method may be somewhere in a file, where a subject did not even look. Unfortunately, this is difficult to assess reliably in an experimental setting without eye tracker software. We could take code displayed on a screen at any time (start line – end line) as indicator, but this does not allow us to deduce at which method on the screen the subject looked, or whether she just scrolled through the code. Nevertheless, this would be an interesting challenge for future experiments, for example, by using an eye-tracking system.

E. Discussion

So, do software measures and program comprehension correlate? Although we refined software measures, such that they better fit the behavior of subjects, none of the refinements was entirely satisfactory. The values of complexity, lines of code, concern attributes, and concern operations still differed considerably, which is not reflected by program comprehension as we measured it. Except for complexity, we found that the weighted value is similar for both versions (all smaller than one). For all other software measures, we cannot confirm a relationship between software measures and program comprehension, despite our effort.

The reader may have noticed that using the observed data to refine software measures makes it more difficult to determine them. Instead of basing the computation of measures solely on source code, we included what subjects did (by including the visited files and the time how long a subject looked at a file). However, this eliminates the benefit of easy computation of software measures: This approach is not feasible in practice, because we cannot predict which developers work with the source code or how long they look at what file. Hence, adapting software measures is not a practical way to improve the predictive power of software measures. Thus, our refinements of software measures are of little use in practice.

Nevertheless, for initial research on a new concept, plausibility discussions with software measures are helpful to establish research hypotheses regarding benefits and drawbacks. However, such hypotheses should be evaluated empirically eventually. This helps us to discover possible hidden relationships, to describe and evaluate claimed benefits of a concept more easily as well as to gain a more thorough understanding of the relation of software measures and program comprehension. Furthermore, our results and proceedings can act as inspiration to develop and evaluate new software measures that describe comprehensibility of source code better.

VI. LIMITATIONS

A. Threats to Internal Validity

One problem of our study is the experience of our subjects with AspectJ. They were introduced into AspectJ in the course they were enrolled in, whereas they worked with Java since they started to study. To diminish the influence of experience with AspectJ, we made sure that for understanding the cause of bugs, subjects did not need a deep understanding of AspectJ syntax. To further reduce the influence of AspectJ experience, we did not let subjects implement a bug fix, but only explain the problem. This way, subjects did not have to implement AspectJ code. Hence, the experience of subjects with AspectJ was sufficient for our purpose. Furthermore, it is not our intent to assess the understandability of AspectJ, but to assess software measures and their relationship to program comprehension.

Another issue is that we explored the data, which can easily drift off to 'fishing for results'. However, we did not exploit our data until we found interesting results, but made some reasonable, well-defined refinements to the computation of software measures. The data exploration is rather a benefit, because we obtained some insights of a possible relationship of software measures and program comprehension, which should be evaluated in further experiments.

B. Threats to External Validity

For our study, we used only one software system, MobileMedia [14]. However, using MobileMedia has the benefit that our results are comparable with numerous results of other researchers, who also used MobileMedia in their work. Consequently, the generalizability to other research with MobileMedia is given, but not the generalizability to other software systems. Thus, the restriction to MobileMedia is both a benefit and a drawback for external validity.

A further restriction is that we only used four software measures. Our results are only applicable to these software measures. To limit this restriction, we used a representative measure of every category we described in Section II. This allows us to carefully draw conclusions for the categories of software measures. Nevertheless, to be able to state a relationship of other software measures (e.g., coupling and cohesion) to program comprehension, they should also be evaluated in a carefully designed experiment. Here, we showed how a carefully designed setting looks like.

Furthermore, we only evaluated how program comprehension and software measures could correlate. We cannot generalize from program comprehension to other software quality facets, such as maintainability and design stability, and their relationship to software measures, which was the focus of numerous studies (cf. Table I).

VII. Related Work

There is a lot of work concerning software measures. We already mentioned one line of research that develops and tests measures to evaluate quality properties of aspect-oriented software systems [14], [15], [19], [29]. In this work, several software projects are evaluated with the developed software measures. For example, Figueiredo et al. [14] assess the design stability of MobileMedia based on software measures. To this end, MobileMedia was developed in two versions in several scenarios, while with every scenario, the program was extended. Based on software measures, both versions were compared. However, the studies did not include the behavior human subjects to assess quality properties.

On the other hand, there is also empirical research with real subjects regarding program comprehension, in which properties of source code, such as depth of inheritance hierarchies [7], comment style [34], and identifier styles [36] were evaluated regarding their effect on comprehensibility. This is similar to our work, in which we assessed the comprehensibility of two systems. However, we did not evaluate whether several facets of source code influence program comprehension. Instead, we were only interested in whether we could observe a difference in comprehensibility.

Furthermore, we conducted some own research regarding program comprehension [10], [11]. In both experiments, we evaluated how background colors can improve program comprehension. However, we did not work with software measures in any of the experiments.

VIII. Conclusion and Future Work

Software measures are often used to assess facets of software quality, such as comprehensibility. The use of software measures to evaluate benefits and drawbacks of new concepts is popular. We focused on the relationship between software measures and program comprehension. We designed an experiment to evaluate how software measures and program comprehension correlate. Two groups of subjects work with two comparable versions of MobileMedia, one implemented in AspectJ, the other in Java with a preprocessor. Both versions differed considerably with respect to several software measures. The results of our experiment do not indicate a relationship between software measures and program comprehension. Even including subjects' behavior (e.g., files a subject worked with) did not improve our results such that we could show that a relationship between program comprehension and software measures exists. We only could find a relationship of complexity with program comprehension, if we took into account how much time a subject spent with a file. For none of the other software measures, the adaptations toward weighted software measures were satisfactory, such that the results we observed would indicate a relationship to software measures.

Nevertheless, our refinements pointed in the right direction: The differences in software measures became smaller with each step. This fits our observation that we did not encounter significant differences in program comprehension between both versions. Only for one task we found a difference, such that AspectJ subjects made more errors.

Our results show that the relationship of software measures and program comprehension is an open issue. If we take into account how subjects work with source code, we can compute weighted software measures, which reflect program comprehension better. However, this eliminates one benefit of software measures: They cannot be computed solely based on facets of source code. Hence, combining software measures with subjects' behavior is not a feasible way for practical use of software measures. Nevertheless, since we are exploring how software measures and program comprehension correlate, it is legitimate to adapt software measures as we did.

The results of our experiment can be a good starting point for future research: Other software measures can be calculated and compared with our results. Another way is to replicate our experiment, either as it is or with slight changes, such as recruiting AspectJ experts or using other software systems. In fact, we explicitly encourage other researches to look into our experiment and use our results for further research.

Acknowledgments

We thank Norbert Siegmund and Jens Dörre for helpful discussions and their support in preparing and conducting the experiment. Feigenspan's work is supported by BMBF project 01IM10002B. Apel's work is supported by the DFG projects #AP 206/2 and #AP 206/4. Kästner's work is supported in part by ERC (#203099).

References

[1] T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[2] I. Bertoncello et al. Explicit Exception Handling Variability in Component-based Product Line Architectures. In *Proc. Int'l Workshop Exception Handling*, pages 47–54. ACM Press, 2008.

[3] J. Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[4] R. Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering*, pages 196–201. IEEE CS, 1978.

[5] A. Bryant et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, pages 109–121. ACM Press, 2006.

[6] P. Clements and L. Northrop. *Software Product Lines: Practice and Patterns*. Addison Wesley, 2001.

[7] J. Daly et al. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In *Proc. Int'l Conf. Software Maintenance*, pages 20–29. IEEE CS, 1995.

[8] A. Dunsmore and M. Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, 2000.

[9] R. Dyer et al. A Preliminary Study of Quantified, Typed Events. In *AOSD Workshop Empirical Evaluation of Software Composition Techniques*, 2010.

[10] J. Feigenspan et al. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *Proc. Int'l Workshop on Feature-Oriented Software Development*, pages 55–62. ACM Press, 2009.

[11] J. Feigenspan et al. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering*, pages 66–75. Institution of Engineering and Technology, 2011.

[12] J. Feigenspan, N. Siegmund, and J. Fruth. On the Role of Program Comprehension in Embedded Systems. In *Workshop Software-Reengineering*, pages 34–35, 2011.

[13] E. Figueiredo et al. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In *ECOOP Workshop Quantitative Approaches in Object-Oriented Software Engineering*, pages 58–69, 2005.

[14] E. Figueiredo et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering*, pages 261–270. ACM Press, 2008.

[15] E. Figueiredo et al. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proc. Europ. Conf. Software Maintenance and Reengineering*, pages 183–192. IEEE CS, 2008.

[16] E. Figueiredo, J. Whittle, and A. Garcia. ConcernMorph: Metrics-based Detection of Crosscutting Patterns. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering*, pages 299–300. ACM Press, 2009.

[17] I. Galvâo, P. van den Broek, and M. Akşit. A Model for Variability Design Rationale in SPL. In *Proc. Europ. Conf. Software Architecture*, pages 332–335. ACM Press, 2010.

[18] A. Garcia et al. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proc. Int'l Conf. Aspect-Oriented Software Development* , pages 3–14. ACM Press, 2005.

[19] P. Greenwood et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 176–200. Springer, 2007.

[20] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

[21] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the Maintainability of Object-Oriented Software. In *IEEE Region 10 Conf. Computer and Comm. Systems*, pages 404–409. IEEE CS, 1990.

[22] K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. pages 327–353. Springer, 2001.

[24] J. Koenemann and S. Robertson. Expert Problem Solving Strategies for Program Comprehension. In *Proc. Conf. Human Factors in Computing Systems*, pages 125–130. ACM Press, 1991.

[25] U. Kulesza et al. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. Int'l Conf. Software Maintenance*, pages 223–233. IEEE CS, 2006.

[26] B. Lientz and B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[27] R. Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[28] S. McConnell. *Code Complete*. Microsoft Press, second edition, 2004.

[29] A. Molesini et al. On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. In *Proc. Working IEEE/IFIP Conf. on Software Architecture*, pages 29–38. IEEE CS, 2008.

[30] D. Mook. *Motivation: The Organization of Action*. W.W. Norton & Co., second edition, 1996.

[31] B. Morin et al. Taming Dynamically Adaptive Systems using Models and Aspects. In *Proc. Int'l Conf. Software Engineering*, pages 122–132. IEEE CS, 2009.

[32] D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[33] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[34] L. Prechelt et al. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, 2002.

[35] M. Robillard and G. Murphy. Representing Concerns in Source Code. *ACM Trans. Softw. Eng. & Methodology*, 16(1):1–38, 2007.

[36] B. Sharif and J. Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. Int'l Conf. Program Comprehension*, pages 196–205. IEEE CS, 2010.

[37] B. Shneiderman and R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Parallel Programming*, 8(3):219–238, 1979.

[38] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

[39] A. von Mayrhauser and M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[40] J. Yellott. Correction for Fast Guessing and the Speed Accuracy Trade-off in Choice Reaction Time. *Journal of Mathematical Psychology*, 8:159–199, 1971.